Elgg Documentation

Version master

Various

déc. 07, 2020

Table des matières

Elgg (pronunciation) est un framework de développement rapide pour construire des applications web sociales. C'est un excellent choix pour construire tout type d'application où des utilisateurs se connectent et partagent des informations.

CHAPITRE 1

Fonctionnalités

— **Une API bien documentée** qui permet aux développeurs d'accélérer leurs nouveaux projets avec une courbe d'apprentissage simple
— **Composer** est le gestionnaire de paquets qui simplifie grandement l'installation et la maintenance du coeur d'Elgg et des plugins
— **Un système flexible de hooks et d'événements** qui permet aux plugins d'étendre et de modifier la majorité des aspects du fonctionnement et du comportement des fonctionnalités des applications
— **Un système de vues extensible** qui permet aux plugins de collaborer sur la couche de présentation des applications et de construire des thèmes personnalisés complexes
— **Un système d'assets statiques mis en cache** qui permet aux thèmes et aux plugins de servir des images, feuilles de style, polices et scripts sans charger le moteur
— **L'authentication des utilsateurs** s'appuie sur des modules d'authentification enfichables, ce qui permet aux applications d'implémenter des protocoles d'authentification personnalisés
— **La Sécurité** est assurée par une validation anti CSRF native, des filtres XSS stricts, des signatures HMAC, et les dernières approches cryptographiques pour le hachage des mots de passe
— **Une API côté client** motorisée par des modules JavaScript asynchrones chargés via RequireJS et un service Ajax natif pour une communication aisée avec le serveur
— **Un système d'Entités flexible** qui permet aux applications de prototyper de nouveaux types de contenus et d'interactions entre utilisateurs
— **Un modèle de données très bien conçu** avec une couche d'API consolidée qui permet aux développeurs de créer facilement les interfaces avec la base de données
— **Une système de contrôle d'accès** qui permet aux applications de construire des politiques d'accès aux contenus granulaires, comme de créer des réseaux privés et des intranets
— **Groupes** - support d'emblée pour des groupes d'utilisateurs
— **Stockage de fichiers** supporté par une API flexible qui permet aux plugins de stocker les fichiers créés par les utilisateurs et de les servir/streamer sans démarrer le moteur
— **Un Service de Notifications** qui permet aux applications d'abonner des utilisateurs aux notifications sur le site ou par email, et d'implémenter l'intégration avec d'autres services tierce-partie
— **Des Services Web RPC** qui peuvent être utilisés pour des intégrations complexes avec des applications externes et des clietns mobiles
— **L'Internationalisation** et la localisation des applications Elgg est simple et peut être intégrée avec des services tierce-partie tels que Transifex
— **La communauté Elgg** peut aider sur n'importe quelle problématique qui survienne, et héberge un répertoire

de plus de **1000+ plugins open source**

Sous le capot :

- Elgg est un framework modulaire orienté objet (OOP) qui est dirigé par des services DI
- compatible NGINX ou Apache
- Symfony2 HTTP Foundation gère les requêtes et réponses
- RequireJS gère les modules AMD
- Zend Mail gère les emails sortants
- Filtres XSS htmLawed
- DBAL
- Migrations de base de donnée Phinx
- CSS-Crush pour le pré-traitement CSS
- Imagine pour la manipulation d'image
- Cache persistant avec Memcached et/ou Redis
- Gestion des erreurs avec Monolog

CHAPITRE 2

Exemples

Elgg a été utilisé pour construire tous types d'applications sociales :

- réseau sociaux ouverts (similaires à Facebook)
- thématiques (comme la communauté Elgg)
- intranets privés ou professionnels
- rencontres
- éducatifs
- blog d'entreprise

Ceci constitue la documentation officielle du projet Elgg.

CHAPITRE 3

Poursuivre le lecture

3.1 Pour démarrer

Découvrez si Elgg est adapté pour votre communauté.

3.1.1 Fonctionnalités

Présentation de sites : https://elgg.org/showcase

Pour les développeurs

- License permissive
- Framework pour les thèmes
- Internationalisation
- Moteur de templates
- Framework de widgets
- APIs pour les plugins
- Graph social
- API pour les services web
- Framework JS basé sur jQuery
- Gestion des sessions
- Routage d'URL personnalisé

Pour les administrateurs

— Page et image du profil des membres
— Listes de contrôle d'accès (ACL) très détaillées
— Contacts et listes de contacts (à la manière des cercles G+)
— Responsive, design compatible mobile
— Support RSS
— Flux d'activité
— Plugins pour les types de contenus courants tels que des blogs, signets, fichiers, du microblogging, des messages privés, documents, forums, discussions
— Authentification des utilisateurs et administration

Si vous avez besoin de plus de fonctionnalités que ce qu'Elgg propose d'emblée, voici quelques possibilités :
— Ajoutez-en de nouvelles en *installant des plugins* - par exemple des blogs, des forums, des signets
— Développez vos propres fonctionnalités via des plugins
— Engagez quelqu'un pour le faire pour vous

3.1.2 Plugins joints

Elgg est livré avec un jeu de plugins. Ils fournissent les fonctionnalités fondamentales pour votre réseau social

Blog

Un weblog, ou blog, est sans conteste l'un des éléments fondamental d'ADN de tous les types de sites de réseautage social. La forme la plus simple de publication personnelle, qui permet de publier des notes textuelles dans un ordre antéchronologique. Les commentaires sont également une part importante du blogging, en transformant un acte de publication personnel en une conversation.

Le blog d'Elgg étend ce modèle en fournissant des contrôles d'accès pour chaque entrée et un système de tags transverse aux blogs. Vous pouvez contrôler précisément qui peut voir quel article de blog, et identifier des articles écrits par d'autres personnes sur des sujets similaires. Vous pouvez également voir les entrées écrites par vos contacts (auxquels vous avec accès).

Voir aussi :

Blogging sur Wikipedia

Tableau de bord

Le tableau de bord est livré à la fois avec la version complète et noyau seulement d'Elgg. C'est le portail des utilisateurs vers des activités qui leur sont importantes à la fois à l'intérieur du site et depuis des sources externes. En utilisant la puissante API widgets d'Elgg, il est possible de construire des widgets qui retirent du contenu pertinent depuis l'intérieur d'un site Elgg, ainsi que de récupérer des informations depuis des sources tierce-partie telles que Twitter ou Flickr (dès lors que ces widgets existent). Le tableau de bord des utilisateurs n'est pas la même chose que leur profil : tandis que le profil est destiné à être consulté par les autres, le tableau de bord est un espace strictement personnel que les utilisateurs utilisent pour leurs propres besoins.

Fig. 1 – Un tableau de bord Elgg typique

Diagnostics

Pour l'utilisateur techniquement averti, les diagnostics système permettent d'évaluer rapidement l'environnement serveur, le code d'Elgg, et les plugins d'une installation Elgg. Les diagnostics sont un plugin système du noyau qui est activé par défaut avec Elgg. Pour télécharger le fichier de diagnostics, suivez les étapes ci-dessous. Le fichier est un vidage de tous types d'informations utiles.

Pour l'utiliser :

- Connectez-vous en tant qu'administrateur
- Rendez-vous dans Administration -> Administrer -> Utilitaires ->Diagnostics Système
- Cliquez sur "Télécharger"

Contenu du fichier de vidage des diagnostics système

- Liste de tous les fichiers d'Elgg ainsi qu'un hash pour chaque fichier
- Liste de tous les plugins
- Superglobales PHP
- Paramètres PHP
- Paramètres Apache
- **Valeurs de Elgg CONFIG**
  - chaînes de traduction
  - paramètres du site
  - paramètres de la base de données
  - hooks des plugins
  - actions
  - vues
  - gestionnaires de pages
  - beaucoup plus

Répertoire de fichiers

Le répertoire de fichiers permet à des membres de charger n'importe quel type de fichier. Comme avec tout dans un système Elgg, vous pouvez aisément filtrer les fichiers par tags et restreindre l'accès de sorte qu'ils soient visibles seulement par les personnes que vous souhaitez. Chaque fichier peut également avoir des commentaires associés.

Il existe plusieurs types d'usages différents pour cette fonctionnalité

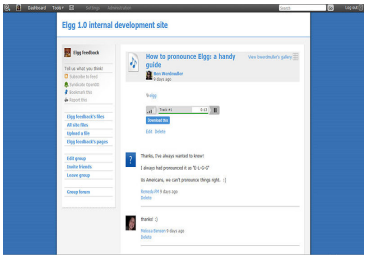

Fig. 2 – Un fichier dans un répertoire de fichiers Elgg

Galerie de photo

Quand un utilisateur charge des photographies ou d'autres images, elles sont automatiquement rassemblées dans une galerie de photos Elgg dans laquelle il est possible de naviguer. Les utilisateurs peuvent aussi voir les photos que leurs contacts ont chargées, ou voir des images attachées à un groupe. Une version plus grande de la photo apparaît en cliquant sur l'un des fichiers.

Podcasts

Un répertoire de fichiers Elgg est doublé automatiquement par un flux RSS, de sorte que vous pouvez vous abonner à du nouveau contenu audio en utilisant des programmes tels que iTunes.

Contenus spéciaux

Il est possible pour d'autres plugins d'ajouter de nouveaux lecteurs et visionneuses pour différents types de contenus. L'auteur d'un plugin peut par exemple intégrer de cette manière une visionneuse pour des documents Word.

Note pour les développeurs

Pour ajouter un lecteur pour un nouveau type de contenu, créez un plugin avec des vues de la forme `file/specialcontent/mime/type`. Par exemple, pour créer une visionneuse pour des documents Word, vous pouvez créer une vue nommée `file/specialcontent/application/msword`, puisque `application/msword` est le type MIME pour les documents Word. A l'intérieur de cette vue, la version `ElggEntity` du fichier sera référencée en tant que `$vars['entity']`. Dès lors, l'URL du fichier téléchargeable est :

```
echo $vars['entity']->getDownloadURL();
```

En utilisant cela, il devrait être possible de développer la plupart des types de visionnneuses embarquables.

Groupes

Une fois que vous avez trouvé d'autres personnes avec des centres d'intérêt similaires - ou peut-être que vous faites partie d'un groupe de recherche ou d'un cours - vous pourriez souhaiter disposer d'un environnement plus structuré pour partager du contenu et discuter d'idées. C'est là qu'interviennent les puissantes fonctionnalités d'Elgg en terme de construction de groupes. Vous pouvez créer et modérer autant de groupes que vous le désirez.

- Vous pouvez conserver toute l'activité du groupe privée pour le groupe, ou utiliser l'option "rendre public" pour disséminer ses travaux auprès d'un public plus large.
- Chaque groupe produit des flux RSS granulaires, de sorte qu'il est aisé de suivre les développements des groupes
- Chaque groupe dispose de sa propre URL et page de profil
- Chaque groupe dispose d'un répertoire de fichiers *Répertoire de fichiers*, d'un forum, de pages et d'un panneau d'affichage

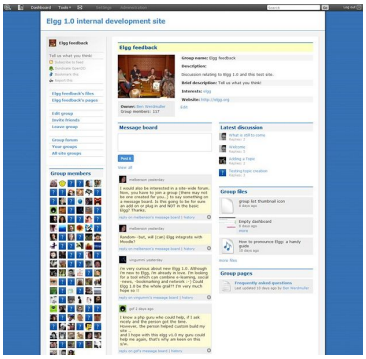

Fig. 3 – Un profil de groupe typique

Messageboard

Le panneau d'affichage - similaire au "Mur" de Facebook ou à un mur de commentaires dans d'autres réseaux - est un plugin qui permet aux membres de placer un widget panneau d'affichage sur leur profil. Les autres membres peuvent alors publier des messages qui vont apparaître sur ce panneau d'affichage. Vous pouvez ensuite répondre directement à n'importe quel message et voir l'historique entre vous et la personne qui a publié le message.

Messages

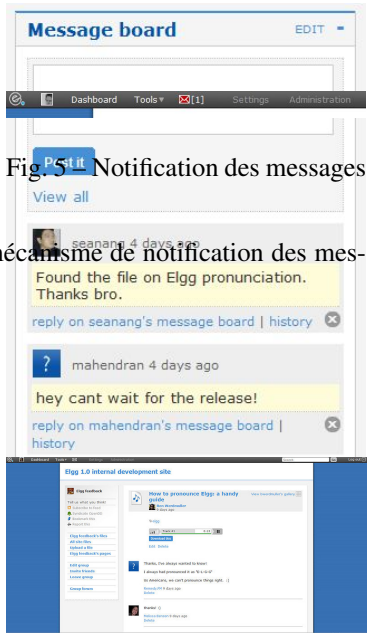

Fig. 5 – Notification des messages

Des messages privés peuvent être envoyés aux memrbes en cliquant sur leur avatar ou le lien vers leur profil. Puis, en utilisant l'"*éditeur WYSIWYG* natif, il est possible de formater le message. Chaque membre dispose de sa propre boîte de réception et d'envoi. Il est possible d'être notifié par email des nouveaux messages.

Quand les membres s'identifient, il sont notifiés de tout nouveau message par le mécanisme de notification des messages dans leur barre d'outil supérieure.

Pages

Fig. 4 – Un exemple de panneau d'affichage placé sur le profil

Fig. 6 – Une page Elgg

Le plugin pages permet d'enregistrer et de conserver d'une manière organisée et hiérarchisée des pages de texte, et d'en restreindre à la fois les accès en lecture et en écriture. Ceci signifie que vous pouvez créer collaborativement un ensemble de documents avec une liste de personnes choisies, participer à un processus d'écriture au sein d'un groupe formalisé, ou simplement utiliser cette fonctionnalité pour créer un document que vous seul pouvez voir, jusqu'à ce que vous choisissiez de le partager une fois terminé. Le menu de navigation simple vous permet de voir la structure de l'ensemble de documents depuis n'importe quelle page. Vous pouvez créer autant de ces structures que vous le souhaitez ; chaque page individuelle dispose de ses propres contrôles d'accès, de sorte que vous pouvez révéler des portions de la structure tout en conservant les autres invisibles. Dans la ligne des autres éléments dans Elgg, vous pouvez aisément ajouter des commentaires sur une page, ou rechercher des pages par tag.

Usage

Les pages expriment vraiment leur potentiel dans deux domaines, d'abord comme moyen pour les utilisateurs de construire des choses telles qu'un CV, un portfolio, une documentation réflexive, et autres constructions de ce type. La seconde est dans le domaine de la collaboration, tout particulièrement dans le cadre d'un groupe. Avec les puissants outils de contrôle d'accès à la fois en lecture et en écriture, ce plugin est idéal pour la création collaborative de documents.

Note : Les développeurs devraient tenir compte du fait qu'il existe en fait 2 types de pages :

1. Pages racine (avec le sous-type `page_top`)
2. Pages normales (avec le sous-type `page`)

Profil

Le plugin profile est livré à la fois avec la version complète et noyau seulement d'Elgg. L'objectif est qu'il puisse être désactivé et remplacé par un autre plugin de profil si vous le souhaitez. Il fournit un certain nombre d'éléments de fonctionnalités que beaucoup considèrent comme fondamentales pour le concept d'un site de réseautage social, et est unique parmi les plugins parce que l'icône de profil qu'il définit est référencée comme un standard à travers l'ensemble du système.

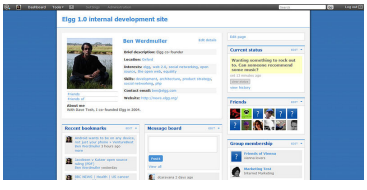

Fig. 7 – Un profil Elgg

Informations utilisateur

Ceci fournit des informations à propos d'un utilisateur, qui est configurable à partir du fichier start.php du plugin. Vous pouvez changer les champs de profil disponibles depuis le panneau d'administration. Chaque champ de profil dispose de son propre niveau d'accès, de sorte que les membres puissent choisir exactement qui peut voir quel élément précis. Certains des champs contiennent des tags (par exemple *compétences*) : restreindre l'accès à ce champ va également restreindre les personnes qui peuvent vous trouver via ce tag.

Avatar utilisateur

L'avatar utilisateur représente un utilisateur (ou un groupe) à travers le site. Par défaut, l'avatar intègre un menu contextuel sensitif qui vous permet d'effectuer des actions sur l'utilisateur dès lors que vous voyez son avatar. Par exemple vous pouvez l'ajouter comme contact, lui envoyer un message privé, et plus. Chaque plugin peut ajouter des entrées à ce menu contextuel, de sorte que son contenu final va beaucoup dépendre des fonctionnalités actives dans le site Elgg actuel.

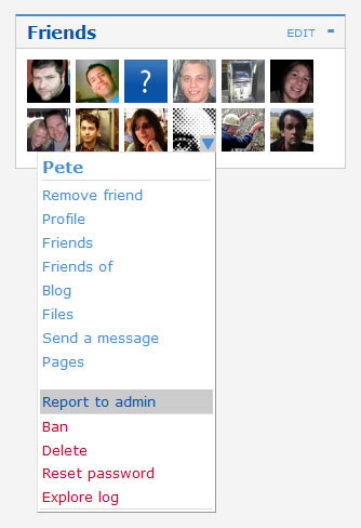

Fig. 8 – Le menu contextuel Elgg

Notes pour les développeurs

Utiliser une icône de profil différente Pour remplacer l'icône de profil, ou fournir plus de contenu, étendre la vue `icon/user/default`.

Ajouter des éléments au menu contextuel Le menu contextuel peut être étendu en enregistrant un *hook de plugin* pour "register" "menu :user_hover", les sections suivantes ont une signification spéciale :
— **default** pour les liens non actifs (par ex. pour lire un blog)
— **admin** pour les liens accessibles seulement par les administrateurs
Dans tous les cas, l'utilisateur en question sera passé en tant que `$params['entity']`.

Le Fil

Le plugin de câble d'Elgg « The Wire » (Le Fil) est un plugin de micro-blogging à la Twitter, qui permet aux utilisateurs de publier des notes vers le fil.

Validation des comptes utilisateur via l'email

Le plugin uservalidationbyemail ajoute une étape au processus d'inscription des utilisateurs. Après que l'utilisateur se soit enregistré sur le site, un email est envoyé à son adresse email afin de valider que cette adresse email appartient bien à l'utilisateur. L'utilisateur ne pourra se connecter qu'après avoir cliqué sur le lien de vérification contenu dans l'email.

Déroulement pour l'utilisateur

1. L'utilisateur crée un compte en se rendant sur la page d'inscription de votre site
2. Une fois le compte créé l'utilisateur arrive sur une page qui indique de vérifier la présence de l'email de validation sa boîte de messagerie
3. Dans l'email de validation, un lien permet de confirmer son adresse email
4. Après avoir cliqué sur le lien, le compte est validé
5. Si possible l'utilisateur est connecté

Si l'utilisateur tente de s'identifier avant d'avoir validé son compte, une erreur indique que l'utilisateur doit d'abord vérifier sa boite mail. De plus, un nouveau mail de validation est envoyé.

Options pour les administrateurs du site

Un administrateur du site peut effectuer certaines actions sur les comptes non validés. Une liste des membres non validés se trouve dans Administration -> Utilisateurs -> Non validés. L'administrateur peut manuellement valider ou supprimer l'utilisateur. Il existe également une option pour renvoyer l'email de validation.

Les plugins suivants sont également livrés avec Elgg, mais ne sont pas (encore) documentés
- activity (activité)
- signets
- ckeditor
- custom_index (index personnalisé)
- developers (développeurs)
- discussions
- embed
- externalpages
- friends (contacts)
- friends_collections
- garbagecollector
- invitefriends
- likes
- members
- notifications
- reportedcontent
- search
- site_notifications
- system_log
- tagcloud
- web_services

3.1.3 License

MIT ou GPLv2

Un package Elgg complet comprenant le framework et un noyau de plugins est disponible sous la ersion 2 de la licence GNU General Public License (GPLv2). Nous distribbuons également le framework (sans les plugins) sous la licence MIT.

FAQ

Les réponses suivantes vous sont proposées à titre informatif ; elles ne constituent pas un conseil légal. Consultez un juriste si vous voulez être sûr(e) des réponses à ces questions. La Fondation Elgg ne peut pas être tenues pour responsable des décisions que vous prendrez sur la base de ce que vous lisez sur cette page.

Pour les questions qui n'ont pas réponse ici, veuillez vous référer à la FAQ officielle pour GPLv2 : official FAQ for the GPLv2.

Combien coûte Elgg ?

Elgg peut être téléchargé, installé et utilisé gratuitement. Si vous souhaitez faire un don, nous apprécions d'avoir des partenaires financiers financial supporters !

Puis-je enlever les liens / le branding Elgg ?

Oui.

Puis-je modifier le code source ?

Oui, mais en général nous vous recommandons de faire vos modifications sous forme de plugins de sorte que lorsque une nouvelle version d'Elgg est publiée le processus de mise à niveau reste aussi simple que possible.

Puis-je demander des frais d'adhésion à mes utilisateurs ?

Oui.

Si je modifie Elgg, dois-je rendre mes modifications disponibles ?

Non, si vous utilisez Elgg pour fournir un service, vous n'avez pas à rendre les sources disponibles. Si vous distribuez une version modifiée d'Elgg, alors vous devez inclure le code sources des modifications.

Si j'utilise Elgg pour héberger un réseau, est-ce que la Fondation Elgg a quelques droits que ce soit sur mon réseau ?

Non.

Quelle est la différence entre la version MIT et la version GPL ?

Les plugins ne sont pas inclus avec la version MIT.

Vous pouvez distribuer un produit commercial construit avec Elgg en utilisant la version MIT sans devoir rendre disponibles vos modifications.

Avec la version sous licence GPL, vous devez rendre vos modifications du framework publiques si vous redistribuez le framework.

Pourquoi certains plugins sont-ils absents de la version MIT ?

Les plugins ont été développés sous licence GPL, et ne peuvent donc pas être diffusés sous une licence MIT. De plus, certains plugins font appel à des dépendances externes qui ne sont pas compatibles avec la licence MIT.

Ai-je le droit de distribuer un plugin pour Elgg sous une licence commerciale ?

Nous pensons que vous le pouvez, puisque les plugins ne dépendent habituellement que du cœur du framework (core) et que le framework est disponible sous la licence MIT. Ceci dit, nous vous conseillons vraiment de consulter un juriste sur ce point particulier si vous voulez être absolument sûr(e).

Notez que les plugins distribués via la communauté doivent être publiés sous une licence compatible GPLv2. Ils n'ont pas à être publiés sous licence GPLv2, seulement sous une licence compatible (comme la MIT).

Pouvons-nous construire notre propore outil utilisant Elgg et vendre cet outil à nos clients ?

Oui, mais dans ce cas vos clients seront libres de redistribuer cet outil sous les termes de la licence GPLv2.

3.1.4 Installation

Ayez votre propre instance d'Elgg opérationnelle en un rien de temps.

Contents

Pré-requis

- MySQL 5.5.3+ (5.0.0+ si vous mettez à niveau une installation existante)
- PHP 7.0+ avec les extensions suivantes :
  - GD (pour les opérations graphiques)
  - PDO (pour la connexion à la base de données)
  - JSON (pour les réponses AJAX, etc.)
  - XML (pour lire fichiers manifest des plugins, etc.)
  - Multibyte String support (pour i18n)
  - La configuration appropriée et la capacité d'envoyer des emails via un MTA (Mail Transport Agent)
- Serveur web avec le support de la réécriture d'URL (URL rewriting)

Le support officiel est fourni pour les configurations suivantes :

- **Serveur Apache**
  - Apache avec le module rewrite module activé
  - PHP exécuté comme un module Apache
- **Serveur Nginx**
  - Nginx avec PHP-FPM utilisant FastCGI

Par « support officiel », nous entendons que :

- La plupart des développements et des tests sont effectués avec ces configurations

— Une bonne part de la documentation a été écrite en partant du principe qu'Apache ou Nginx est utilisé
— La priorité sur les rapports de bug est donnée aux utilisateurs d'Apache et Nginx si le bug est propre au serveur web (mais ces cas sont rares).

Politique de support des navigateurs

Les branches de fonctionnalités supportent les 2 dernières versions de tous les principaux navigateurs disponibles au moment de la première publication d'une version stable pour cette branche.

Les versions de correction de bugs ne modifient pas le support des navigateurs, même si une nouvelle version du navigateur a été publiée depuis.

Les principaux navigateurs signifie ici les navigateurs suivants, ainsi que leurs homologues mobiles :
— Navigateur Android
— Chrome
— Firefox
— IE
— Safari
« Support » peut signifier que nous prenons avantage des technologies nouvelles et non implémentées, mais fournissons un polyfill JavaScript pour les navigateurs qui en ont besoin.

Il se peut qu'Elgg fonctionne avec des navigateurs non supportés, cependant la compatibilité peut être perdue à tout moment, y compris lors de la publication d'une correction de bug.

Vue d'ensemble

Chargez Elgg

Avec Composer (recommandé si vous êtes à l'aise avec la ligne de commande) :

```
composer self-update
composer create-project elgg/starter-project:dev-master ./path/to/project/root
cd ./path/to/project/root
composer install
composer install # 2nd call is currently required
vendor/bin/elgg-cli install # follow the questions to provide installation details
```

A partir du fichier ZIP (recommandé si vous n'êtes pas à l'aise avec la ligne de commande) :
— Télécharger la dernière version d'Elgg
— Chargez le fichier ZPI avec un client FTP sur votre serveur
— Dézippez les fichiers dans la racine web de votre domaine.

Créer un fichier pour les données

Elgg a besoin d'un répertoire particulier pour stocker les fichiers chargés, y compris les images du profil et les photos. Vous devez créer ce répertoire.

Attention : Pour des raisons de sécurité, ce dossier **DOIT** être conservé hors de la racine de vos documents (DocumentRoot). Si vous l'avez créé dans le dossier `/www/` ou `/public_html/`, ce n'est pas la bonne manière de faire.

Une fois que ce répertoire a été créé, vous devez vous assurer que le serveur web sur lequel tourne Elgg a le droit d'écrire et de créer des répertoires à l'intérieur. Ceci ne devrait pas être un problème sur les serveurs basés sur Windows, mais si votre serveur utilise Linux, Mac OS X ou une variante d'UNIX, vous aurez besoin de définir les droits d'accès du répertoire.

Si vous utilisez un client FTP graphique pour charger les fichiers, vous pouvez habituellement définir les droits en faisant un clic droit sur le dossier et en sélectionnant « propriétés » ou « Informations ».

Note : Les répertoires doivent pouvoir être lus et écrits. Les permissions suggérées dépendent de votre serveur et de la configuration des utilisateurs. Si le répertoire de données a pour propriétaire l'utilisateur du serveur web, les droits d'accès recommandés sont `750`.

Avertissement : Définir les droits de votre répertoire de données à `777` va fonctionner, mais c'est dangereux et n'est pas recommandé. Si vous hésitez sur la bonne configuration à choisir pour les permissions, contactez votre hébergeur pour plus d'informations.

Créer une base de données MySQL

En utilisant l'outil d'administration de base de données de votre choix (si vous ne savez pas lequel, demandez à votre administrateur système), créez une nouvelle base de données MySQL pour Elgg. Vous pouvez créer une base de données MySQL avec n'importe lequel des outils suivants :

Assurez-vous d'ajouter un utilisateur à la base de données avec tous les privilèges et notez le nom de la base de données, l'identifiant et le mot de passe. Vous aurez besoin de ces informations pour installer Elgg.

Mettre en place le Cron

Elgg utilise des requêtes paramétrées pour effectuer des tâches de fond telles qu'envoyer des notifications ou effectuer des travaux de nettoyage de la base de données. Vous devez configurer le *cron* pour pouvoir utiliser ce type de fonctionnalités.

Visiter votre site Elgg

Une fois ces étapes terminées, visitez votre site Elgg dans votre navigateur web. A partir de là, Elgg vous guidera à travers le reste du processus d'installation. Le premier compte que vous créerez à la fin de l'installation sera le compte administrateur.

Une note sur settings.php et .htaccess

L'installeur d'Elgg va essayer de créer deux fichiers pour vous :

- `elgg-config/settings.php`, qui contient la configuration de votre environnement local pour votre installation
- `.htaccess`, qui permet à Elgg de générer des URLs dynamiques

Si ces fichiers ne peuvent pas être générés automatiquement, par exemple parce que le serveur web n'a pas les droits d'accès en écriture dans les répertoires, Elgg vous dira comment les créer. Vous pouvez aussi modifier temporairement les droits d'accès sur la racine du répertoire et le répertoire engine. Définissez les droits d'accès sur ces deux répertoires de sorte que le serveur web puisse écrire ces deux fichiers, terminez le processus d'installation, et modifiez à nouveau

les droits d'accès pour rétablir les droits d'origine. Si, pour quelque raison que ce soit, ceci ne fonctionne pas, vous devrez :

— Depuis `elgg-config/`, copiez `settings.example.php` vers `settings.php`, puis ouvrez-le dans un éditeur de texte et complétez les informations de la base de données
— Sur le serveur Apache, copiez `install/config/htaccess.dist` vers `.htaccess`
— SUr un serveur Nginx copiez `install/config/nginx.dist` vers `/etc/nginx/sites-enabled` et modifiez son contenu

Autres configurations

— Cloud9
— Homestead
— EasyPHP
— IIS
— MAMP
— MariaDB
— Nginx
— Ubuntu
— Hôtes virtuels
— XAMPP

Dépannage

Au secours ! J'ai des soucis pour installer Elgg

D'abord :

— Re-vérifiez que votre serveur répond bien aux pré-requis pour Elgg.
— Suivez si besoin les instructions spécifiques à un environnement
— Avez-vous vérifié que `mod_rewrite` est bien chargé ?
— Est-ce que le module mysql de apache est bien chargé ?

Conservez des notes sur ce que vous faites pour résoudre les problèmes d'installation. Parfois la modification d'un paramètre ou d'un fichier pour essayer de résoudre un problème peut être à l'origine d'un autre problème plus tard. Si vous devez recommencer depuis le début, supprimez simplement tous les fichiers, videz votre base de données, et commencez à nouveau.

Je ne peux pas enregistrer mon fichier de configuration sur une installation (j'ai une erreur 404 lors de l'enregistrement du fichier de configuration)

Elgg dépend de l'extension Apache `mod_rewrite` pour simuler certaines URLs. Par exemple à chaque fois que vous effectuez une action dans Elgg, or lorsque vous visitez la page de profil d'un utilisateur, l'URL est translatée par le serveur en quelque chose qu'Elgg comprend en interne. Ceci est fait en utilisant des règles définies dans un fichier `.htaccess`, qui est le moyen standard d'Apache pour définir des éléments de configuration supplémentaires pour un site.

Cette erreur suggère que les règles de ``mod_rewrite``ne sont pas traitées correctement. Ceci peut arriver pour différentes raisons. Si vous n'êtes pas à l'aise pour mettre en œuvre les solutions indiquées ci-dessous, nous vous recommandons vivement de contacter votre administrateur système ou le support technique et de leur faire suivre cette page.

Le `.htaccess`, s'il n'est pas créé automatiquement (cela se produit quand vous avez un problème avec `mod_rewrite`), vous pouvez le créer en renommant le fichier `install/config/htaccess.dist` distribué avec Elgg en `.htaccess`. Par ailleurs, si vous trouvez un fichier `.htaccess` dans le répertoire d'installation, mais

avez toujours une erreur 404, vérifiez que le contenu de `.htaccess` est identique à celui de `install/config/htaccess.dist`.

``mod_rewrite`` n'est pas installé.

Contrôlez votre `httpd.conf` pour vérifier que ce module est bien chargé par Apache. Vous pouvez avoir besoin de redémarrer Apache pour qu'il tienne compte de tout changement de configuration. Vous pouvez également utiliser PHP info pour contrôler que le module est chargé.

Les règles définies dans ``.htaccess`` ne sont pas respectées.

Dans les paramètres de configuration de votre hôte virtuel (qui peut être intégrée dans `httpd.conf`), modifiez la paramètre AllowOverride pour qu'il ressemble à :

```
AllowOverride all
```

Ceci va indiquer à Apache de prendre les règles de `mod_rewrite` depuis `.htaccess`.

Elgg n'est pas installé à la racine de votre répertoire web (« Elgg is not installed in the root of your web directory ») (par ex. : http ://example.org/elgg/ au lieu de http ://example.org/)

Le script d'installation me redirige vers « action » alors que cela devrait être vers « actions »

Il s'agit d'un problème avec votre configuration de `mod_rewrite`.

> **Attention :** NE CHANGEZ PAS, RÉPÉTEZ, NE CHANGEZ PAS quelque nom de répertoire que ce soit !

J'ai installé Elgg dans un sous-répertoire et mon action d'installation ne fonctionne pas !

Si vous installez Elgg de sorte qu'il soit accessible avec une adresse comme http://example.org/monsite/ plutôt que http://example.org/, il existe une faible possibilité que les règles de réécriture dans .htaccess ne soient pas traitées correctement. Ceci est généralement lié à l'utilisation d'un alias dans Apache. Vous pouvez avoir besoin d'indiquer à mod_rewrite où se situe votre installation Elgg.

- Ouvrez .htaccess dans un éditeur de texte
- Lorsqu'on vous y invite, ajoutez une ligne comme `RewriteBase /chemin/vers/votre/installation/elgg/` (N'oubliez pas le slash final)
- Enregistrez le fichier et rafraîchissez votre navigateur.

Veuillez noter que le chemin que vous utilisez est le chemin **web**, moins l'hôte.

Par exemple, si votre installation Elgg s'affiche sur http://example.org/elgg/, vous devriez définir la base comme ceci :

```
RewriteBase /elgg/
```

Veuillez noter qu'installer dans un sous-répertoire ne nécessite pas d'utiliser RewriteBase. Il y a seulement quelques rares circonstances dans lesquelles c'est imposé par la configuration du serveur.

J'ai tout fait ! mod_rewrite fonctionne correctement, mais j'ai toujours l'erreur 404

Il y a peut-être un problème avec le fichier .htaccess. Parfois la routine d'installation d'Elgg n'arrive pas à en créer un ni à vous le signaler. Si vous en êtes à ce stade et avez essayé tout ce qui est indiqué ci-dessus :

— vérifiez qu'il s'agit bien du .htaccess créé par Elgg (et pas d'un fichier d'exemple fourni par le fournisseur du serveur)
— s'il ne s'agit pas du fichier htaccess fourni par Elgg, utilisez htaccess_dist (renommez-le en .htaccess)

J'ai un message d'erreur indiquant que le test de réécriture a échoué après la page de vérification des pré-requis

J'ai les messages suivants après l'étape de vérification des pré-requis (étape 2) de l'installation :

> Nous pensons que votre serveur utilise un serveur web Apache.
>
> Le test de réécriture a échoué et la cause al plus probable est que AllowOverride n'est pas définie à All pour le répertoire d'Elgg. Ceci empêche Apache de traiter le fichier .htaccess qui contient les règles de réécriture.
>
> Une cause moins probable est qu'Apache est configuré avec un alais pour votre répertoire Elgg et que vous deviez définir RewriteBase dans votre .htaccess. Vous trouverez des instructions complémentaires dans le fichier .htaccess de votre répertoire Elgg.

Après cette erreur, toute interaction avec l'interface web produit une erreur 500 (Erreur interne du serveur - Internal Server Error)

Ceci est probablement causé par le non-chargement du module de filtre en dé-commentant la ligne

> #LoadModule filter_module modules/mod_filter.so

ligne dans le fichier « httpd.conf ».

le fichier de journal d'Apache « error.log » va contenir une entrée similaire à :

> . . . .htaccess : Invalid command "AddOutputFilterByType", perhaps misspelled or defined by a module not included in the server configuration (Commande "AddOutputFilterByType" invalide, peut-être mal écrite ou définie par un module qui n'est pas inclus dans la configuration du serveur)

Il y a une page blanche après que j'ai soumis ma configuration de base de données

Vérifiez que le module d'Apache mysql est installé et est bien chargé.

J'obtiens une erreur 404 avec une URL très longue

Si vous voyez une erreur 404 pendant l'installation ou lors de la création du premier compte utilisateur, avec une URL telle que : `http://example.com/homepages/26/d147515119/htdocs/elgg/action/register` ceci signifie que l'URL de votre site est incorrecte dans la table sites_entity table de votre base de données. Cette valeur a été définie par vous lors de la deuxième étape de l'installation. Elgg essaie de deviner la bonne valeur mais a des difficultés avec les hébergements mutualisés. Utilisez phpMyAdmin pour modifier cette valeur avec la bonne URL de base du site.

J'ai des difficultés pour définir le chemin vers le répertoire de données

Ceci est fortement spécifique au serveur utilisé aussi il est difficile de donner des conseils spécifiques. Si vous avez créé un répertoire pour les données, vérifiez que votre serveur HTTP arrive bien à y accéder. Le moyen le plus simple (mais le moins sécurisé) d'y parvenir est de lui donner les droits d'accès `777`. Il est largement préférable de donner au serveur web la propriété du répertoire et de limiter les droits d'accès.

> **Avertissement :** Définir les permissions du répertoire à `777` permet à **L'INTÉGRALITÉ** d'Internet de placer des fichiers dans la structure de votre répertoire et ainsi d'infecter votre serveur avec des malwares et autres virus. Définir les permissions à `750` devrait être plus que suffisant.

L'origine la plus probable de ce problème est que PHP est configuré pour interdire l'accès à la plupart des répertoires qui utilisent open_basedir. Vous pouvez souhaiter vérifier ce point avec votre fournisseur d'hébergement.

Assurez-vous que le chemin est correct et se termine par un /. Vous pouvez vérifier le chemin dans la table config de votre base de données.

Si vous n'avez qu'un accès FTP à votre serveur et avez créé un répertoire mais n'en connaissez pas le chemin, vous devriez pouvoir le trouver à partir du chemin www défini dans la table config de votre base de données. A ce stade il est recommandé de demander de l'aide à l'équipe support de votre fournisseur d'hébergement.

Je ne peux pas valider mon compte admin car je n'ai pas de serveur d'email !

Bien qu'il soit exact que les comptes utilisateur normaux (à l'exception de ceux créés depuis le panneau d'administration) nécessitent que leur adresse email soit authentifiée avant de pouvoir se connecter, ceci n'est pas nécessaire pour le compte administrateur.

Une fois que vous avez créé votre premier compte utilisateur vous pouvez vous connecter en utilisant les accès que vous avez fourni !

J'ai essayé toutes ces suggestions et je n'arrive toujours pas à installer Elgg

Il est possible que quelque chose d'autre ait été cassé lors du débogage de l'installation. Essayez de faire une nouvelle installation propre :

- supprimez votre base de données Elgg
- supprimez votre répertoire de données
- supprimez les fichiers source d'Elgg
- recommencez l'installation

SI cette méthode ne fonctionne pas, demandez de l'aide à la communauté Elgg. Pensez à indiquer quelle version d'Elgg vous essayez d'installer, à fournir des détails sur le serveur et la plateforme utilisés, ainsi que tous les messages d'erreur que vous pouvez avoir reçus, y compris ceux du journal d'erreur de votre serveur.

3.1.5 Vue d'ensemble pour les développeurs

Cette page constitue une rapide introduction à Elgg pour les développeurs. Elle couvre les approches de base pour travailler avec Elgg en tant que framework, et mentionne certains des termes et des technologies utilisées.

Voyez *Developer Guides* pour des tutoriels ou *Docs de conception* pour une discussion approfondie sur le design.

Base de données et persistence

Elgg utilise MySQL 5.5 ou supérieur pour la persistance des données, et les données de la base sont rranscrites (mappées) dans des Entités (une représentation d'une unité d'information atomique) et des Extenseurs (Extenders) (des informations additionnelles et des descriptions à propos des Entités). Elgg supporte des informations additionnelles telles que des relations entre des Entités, des flux d'activités, et divers types de réglages.

Plugins

Les Plugins modifient le comportement ou l'apparence d'Elgg en surchargeant les vues, ou en gérant des événements et des hooks des plugins. Toutes les modifications d'un site Elgg devraient être implémentées via des plugins pour garantir que la mise à niveau du coeur reste simple.

Actions

Les actions sont le premier moyen pour les utilisateurs d'interagir avec un site Elgg. Les actions sont définies par les plugins.

Événements et hooks du plugin

Les Événements et les Hooks des plugins sont utilisés dans les plugins Elgg pour interagir avec le moteur d'Elgg dans certaines circonstances. Les Événements et les Hooks sont activés à des moments stratégiques lors des processus de démarrage et d'exécution d'Elgg, et permettent aux plugins de modifier ou d'annuler le comportement par défaut.

Vues

Les vues sont la première couche de présentation pour Elgg. Les vues peuvent être surchargées ou étendues par les Plugins. Les vues sont des catégories d'un type de vue (Viewtype), qui définit quel type de sortie devrait être généré par la vue.

JavaScript

Elgg utilise un système JavaScript compatible AMD fourni par RequireJs. jQuery, jQuery UI, jQuery Form, et jQuery UI Autocomplete sont livrés avec Elgg.

Les plugins peuvent charger leurs propres bibliothèques JS.

Internationalisation

L'interface d'Elgg supporte de multiples langues, et utilise Transifex pour la traduction.

Mise en cache

Elgg utilise deux caches pour améliorer les performances : un cache système et SimpleCache.

Bibliothèques tierce-partie

L'utilisation de bibliothèques tierce-partie est gérée par le gestionnaire de dépendances Composer. jQuery, RequireJs ou Zend mail sont des exemples de bibliothèques tierce-partie.

Pour une liste complète des dépendances d'Elgg, consultez la page Packagist d'Elgg.

Génération de données de test

Elgg fournit un mécanisme de génération de données (ensemencement de base de donénes, ou database seeding) pour remplir la base de données avec des entités à des fins de test.

Vous pouvez exécuter les commandes suivantes pour générer des données de test et les supprimer de la base de données.

```
# seed the database
vendor/bin/elgg-cli database:seed

# unseed the database
vendor/bin/elgg-cli database:unseed
```

Les plugins peuvent enregistrer leurs propres graines via le hook `'seeds', 'database'`. Le gestionnaire doit retourner le nom de la classe de la graine, qui doit étendre la classe `\Elgg\Database\Seeder\Seed`.

3.1.6 Elgg CLI

Contents

elgg-cli outils en ligne de commande

Selon la manière dont vous avez installé Elgg et la configuraiton de votre serveur vous pouvez accéder aux binaires de `elgg-cli` de l'une des manières suivantes depuis la racine de votre installation Elgg :

```
php ./elgg-cli list
./elgg-cli list
php ./vendor/bin/elgg-cli list
./vendor/bin/elgg-cli list
```

Commandes disponibles

```
cd /path/to/elgg/

# Get help
vendor/bin/elgg-cli --help

# List all commands
vendor/bin/elgg-cli list

# Install Elgg
vendor/bin/elgg-cli install [-c|--config CONFIG]

# Run Simpletest test suite
vendor/bin/elgg-cli simpletest [-c|--config CONFIG] [-p|--plugins PLUGINS] [-f|--
↪filter FILTER]

# Seed the database with fake entities
vendor/bin/elgg-cli database:seed [-l|--limit LIMIT]

# Remove seeded faked entities
vendor/bin/elgg-cli database:unseed

# Optimize database tables
# Request garbagecollector plugin
vendor/bin/elgg-cli database:optimize

# Run cron jobs
vendor/bin/elgg-cli cron [-i|--interval INTERVAL] [-q|--quiet]

# Flush caches
vendor/bin/elgg-cli flush

# System upgrade
# -v|-vv|-vvv control verbosity of the command (helpful for debugging upgrade scripts)
vendor/bin/elgg-cli upgrade [-v]

# Upgrade and execute all async upgrades
vendor/bin/elgg-cli upgrade async [-v]

# List all, active or inactive plugins
# STATUS = all | active | inactive
vendor/bin/elgg-cli plugins:list [-s|--status STATUS]

# Activate plugins
# List plugin ids separating them with spaces: vendor/bin/elgg-cli plugins:activate␣
↪activity blog
# use -f flag to resolve conflicts and dependencies
vendor/bin/elgg-cli plugins:activate [<plugins>] [-f|--force]

# Deactivate plugins
# List plugin ids separating them with spaces: vendor/bin/elgg-cli plugins:deactivate␣
↪activity blog
# use -f flag to also disable dependents
vendor/bin/elgg-cli plugins:deactivate [<plugins>] [-f|--force]
```

Ajouter des commandes personnalisées

Les plugins peuvent ajouter leurs propres commandes à l'application CLI, en ajoutant le nom de classe de la commande via le hook `'commands','cli'`. La classe de la commande doit étendre `\Elgg\CLI\Command`.

```
class MyCommand extends \Elgg\li\Command {

}

elgg_register_plugin_hook_handler('commands', 'cli', function($hook, $type, $return) {

    $return[] = MyCommand::class;

    return $return;

});
```

3.2 Guides pour les administrateurs

Bonnes pratiques pour gérer efficacement un site construit avec Elgg.

3.2.1 Prise en main

Vous avez installé Elgg et réglé toutes les potentiels problèmes initiaux. Et maintenant ? Voici quelques suggestions pour vous aider à vous familiariser avec Elgg.

Concentrez-vous d'abord sur les fonctionnalités principales

Quand vous démarrez avec Elgg, le mieux est de commencer par explorer les fonctionnalités de base du noyau et des plugins livrés conjointement avant d'installer des plugins tierce-partie. Il est tentant d'installer chaque plugin intéressant du site communautaire, mais explorer les fonctionnalités du noyau vous permet de vous familiariser avec le comportement habituel d'Elgg, et évite d'introduire des bugs ou de la confusion dans votre nouveau réseau Elgg.

Elgg s'installe avec un jeu de plugins de réseau activés : des blogs, des marque-pages partagés, des fichiers, des groupes, des « likes », des messages, des pages de type wiki, des profils utilisateurs, et du microblogging. Pour changer les plugins activés, connectez-vous avec un compte admin, puis utilisez le menu supérieur pour accéder à Administration, puis à Plugins dans la barre latérale droite.

Note : L'utilisateur que vous créez pendant l'installation est un utilisateur admin.

Créez les utilisateurs de test

Les utilisateurs peuvent être créés de deux manières dans un Elgg standard :

1. Terminez le processus d'inscription en utlisant une autre adresse email et un autre nom d'utilisateur. (Déconnectez-vous d'abord ou utilisez un autre navigateur !)
2. Ajoutez un utilisateur dans la section Admin en naviguant vers Administration -> Utilisateurs -> Ajouter un nouvel utilisateur.

Note : Les utilisateurs qui s'inscrivent eux-même doivent valider leur compte avant de pouvoir se connecter. Les utilisateurs qu'un admin crée sont déjà validés.

Explorez les fonctionnalités de l'utilisateur

Utilisez vos utilisateurs de test pour créer des articles de blog, ajouter des widgets à votre profil ou à votre tableau de bord, publier sur le Fil (« The Wire », microblogging) et créer des pages (création de pages de type wiki). Examinez les Paramètres (« Settings ») dans la barre de menu supérieure. C'est à cet endroit qu'un utilisateur définit ses préférences et configure ses outils (qui peuvent être vides parce qu'aucun des plugins par défaut n'ajoute de contrôle à cet endroit).

Explorez les fonctionnalités de l'administration

Tous les outils d'administration sont accessibles en cliquant sur Administration dans le menu supérieur. L'administrateur a un tableau de bord avec un widget qui explique les diverses sections. Changez les options dans le menu Configurer pour changer l'apparence et le comportement de Elgg.

Étendre les fonctionnalités d'Elgg

Après avoir exploré ce qu'Elgg peut faire « out of the box », installez quelques thèmes et plugins. Vous pouvez trouver de nombreux plugins et thèmes sur le site de la communauté Elgg qui ont été développés par d'autres personnes. Ces plugins font tout, de la modification de chaînes de traductions à un re-design complet de l'interface d'Elgg, en passant par l'ajout d'un chat, etc. Puisque ces plugins ne sont pas officiels, assurez-vous de vérifier les commentaires pour vous assurez que vous installez des plugins bien écrits par des développeurs de haute qualité.

3.2.2 Installation avec Composer

Le moyen le plus simple pour garder votre site Elgg à jour est d'utiliser Composer. Composer va s'occuper d'installer toutes les dépendances requises pour tous les plugins et pour Elgg, tout en conservant également ces dépendances à jour sans avoir de conflit.

Contents

Installer Composer

https://getcomposer.org/download/

Installer Elgg en tant que projet Composer

```
composer self-update
composer create-project elgg/starter-project:dev-master ./path/to/my/project
cd ./path/to/my/project
composer install
```

Ceci va créer un fichier composer.json sur la base du projet de démarrage Elgg Elgg starter project qui a les bases pour installer Elgg.

Ouvrez votre navigateur

Allez sur votre navigateur et installez Elgg via l'interface d'installation

Mettre en place le contrôle de version

Cette étape est optionnelle mais vivement recommandée. Elle vous permettra de gérer simplement l'installation du même plugin entre différents environnements (développement/tests/production).

```
cd ./path/to/my/project
git init
git add .
git commit -a -m 'Initial commit'
git remote add origin <git repository url>
git push -u origin master
```

Installer les plugins

Installez les plugins sous forme de dépendances Composer. Ceci suppose qu'un plugin a été enregistré sur Packagist

```
composer require hypejunction/hypefeed
composer require hypejunction/hypeinteractions
# whatever else you need
```

Commitez

Assurez-vous que `composer.lock` n'est pas ignoré dans `.gitignore`

```
git add .
git commit -a -m 'Add new plugins'
git push origin master
```

Déployer en production

Déploiement initial

```
cd ./path/to/www

# you can also use git clone
git init
git remote add origin <git repository url>
git pull origin master

composer install
```

Déploiements subséquents

```
cd ./path/to/www
git pull origin master

# never run composer update in production
composer install
```

3.2.3 Mettre à niveau Elgg

Ce document va vous guider à travers les étapes nécessaires pour mettre à niveau votre installation Elgg vers la dernière version.

Si vous avez écrit des plugins personnalisés, vous devriez également lire les guides de développement pour les *informations sur la mise à jour du code des plugins* pour la dernière version d'Elgg.

Contents

Conseil

- **Sauvegardez votre base de données, votre répertoire de données et le code source**
- Faites attentions aux commentaires spécifiques pour certaines versions ci-dessous
- Les versions inférieures à 2.0 sont invitées à n'effectuer qu'**une seule mise à niveau mineure à la fois**
- Vous pouvez mettre à niveau depuis n'importe quelle version mineure vers n'importe quelle version mineure supérieure au sein de la même version majeure (2.0 -> 2.1 oo 2.0 -> 2.3)
- Vous pouvez mettre à niveau uniquement la dernière version mineure de la version majeure précédente vers n'importe quelle version mineure de la version majeure suivante (2.3 -> 3.0 ou 2.3 -> 3.2, mais pas 2.2 -> 3.x).
- A partir de Elgg 2.3.* vous pouvez mettre à niveau vers n'importe quelle version ultérieure de Elgg sans devoir passer par chacune des versions mineures (par ex. vous pouvez mettre à niveau directement de la 2.3.8 vers la 3.2.5, sans devoir mettre à niveau vers les versions 3.0 et 3.1)
- Essayez la nouvelle version sur un site de test avant d'effectuer une mise à niveau
- Signalez tout problème dans les plugins aux auteurs de plugins
- Si vous êtes auteur de plugins, vous pouvez signaler tout problème de rétro-compatibilité sur GitHub

De 2.3 à 3.0

1. Mettez à niveau `composer.json`

Si vous avez utilisé le projet de démarrage Elgg pour installer Elgg 2.3, vous pouvez avoir besoin de mettre à niveau votre `composer.json` :

- modifiez les pré-requis de la plateforme vers PHP >= 7.0
- facultativement, définiez les paramètres d'optimisation de l'autoloader
- facultativement, désactivez le plugin fxp-asset au bénéfice de asset-packagist

Votre `composer.json` devrait ressembler à quelque chose comme ceci (en fonction des changements que vous avez vous-même introduits) :

```
{
        "type": "project",
        "name": "elgg/starter-project",
        "require": {
                "elgg/elgg": "3.*"
        },
        "config": {
                "process-timeout": 0,
                "platform": {
                        "php": "7.0"
                },
                "fxp-asset": {
                        "enabled": false
                },
                "optimize-autoloader": true,
                "apcu-autoloader": true
        },
        "repositories": [
                {
                        "type": "composer",
                        "url": "https://asset-packagist.org"
          }
        ]
}
```

2. Mettez à jour `.htaccess`

Trouvez la ligne :

```
RewriteRule ^(.*)$ index.php?__elgg_uri=$1 [QSA,L]
```

Et rempalcez-la par :

```
RewriteRule ^(.*)$ index.php [QSA,L]
```

3a. Mise à niveau via Composer (recommandé)

Si vous aviez installé votre projet 2.3 avec composer, vous pouvez suivre la séquence suivante :

Sauvegardez votre base de données, votre répertoire de données, et le code

```
composer self-update

cd ./path/to/project/root
composer require elgg/elgg:~3.0.0
composer update
vendor/bin/elgg-cli upgrade async -v
```

3b. Mise à niveau manuelle (approche traditionnelle)

Les mises à niveau manuelles sont une opération pénible pour les administrateurs de sites. Nous vous déconseillons de maintenir un site Elgg en utilisant des archives ZIP. Economisez-vous du temps en apprenant comment utiliser `composer` et les systèmes de contrôle de version, tels que `git`. Cette tâche sera également plus complexe si vous utilisez des plugins tierce-partie et/ou si vous avez modifié les fichiers du noyau !

1. **Sauvegardez votre base de données, votre répertoire de données, et le code**
2. Identifiez-vous sur votre site en tant qu'administrateur
3. Téléchargez la nouvelle version de Elgg depuis http://elgg.org
4. **Mettez à jour les fichiers**
   - Si vous mettez à niveau vers une version majeure, vous devrez replacer tous les fichiers du noyau et supprimer tous les fichiers qui ont été retirés du noyau d'Elgg, car ils peuvent interférer avec le bon fonctionnement de votre site.
   - Si vous mettez à jour vers une version mineure ou un patch, vous devez remplacer tous les fichiers du code.
5. **Fusionnez tous les nouveaux changements des règles de réécriture (rewrite rules)**
   - Pour Apache depuis `install/config/htaccess.dist` vers `.htaccess`
   - Pour Nginx depuis `install/config/nginx.dist` vers la configuration de votre serveur (habituellement dans `/etc/nginx/sites-enabled`)
6. Visitez http://your-elgg-site.com/upgrade.php
7. Exécutez les mises à niveau asynchrones via http://your-elgg-site.com/admin/upgrades

---

Note : Toute modification devrait avoir été faite au sein de plugins, de sorte qu'elles ne soient pas perdues lors de l'écrasement des fichiers. Si ce n'est pas le cas, prenez soin de maintenir vos modifications.

Note : Si vous n'arrivez pas à accéder au script `upgrade.php` et recevez une erreur, ajoutez `$CONFIG->security_protect_upgrade = false;` à votre ficher `settings.php` et supprimez-le après avoir terminé toutes les étapes de l'installation.

Note : Si vous rencontrez des soucis avec des plugins lors du processus de mise à niveau, ajoutez un fichier nommé `disabled` dans votre dossier `/mod/`. Ceci va désactiver tous les plugins, de sorte que vous puissiez terminer la mise à niveau du noyau. Vous pouvez ensuite gérer les mises à niveau de chacun des plugins l'un après l'autre.

Si vous avez installé Elgg en utilisant une archive d'installation mais souhaitez maintenant basculer vers composer :

- Mettez à niveau votre installation en utilisant la Méthode manuelle
- Déplacez votre code vers un emplacement provisoire
- Créez un nouveau projet composer en utilisant le projet Elgg de démarrage en suivant les *instruction d'installation* dans le répertoire racine de votre installation actuelle
- Copiez les plugins tierce-partie depuis votre ancienne installation vers le répertoire `/mod`
- Exécutez l'installateur de Elgg en utilisant votre navigateur ou l'outil `elgg-cli`
- Quand vous arrivez à l'étape de la base de données, indiquez les mêmes accès que vous avez utilisés pour l'installation manuelle, Elgg va comprendre qu'il y a une installation existante et ne remplacera aucune valeur de la base de données
- Facultativement, commitez votre nouveau projet vers un suivi de version

Applying a patch using Composer

The definition of a patch can be found in the *Release policy*.

Your `composer.json` requirement for Elgg should be `~3.y.0` (where `y` is the minor version `0`, `1`, etc. you wish to have installed). This will make sure you can easily install patches without the risk of installing the next minor release.

```
{
        "require": {
                "elgg/elgg": "~3.0.0"
        }
}
```

Just to be sure you can first verify what will be installed / upgraded by executing the folowing command

```
# to get a full list of all packages which can be upgraded
composer update --dry-run

# or if you only wish to check for Elgg
composer update elgg/elgg --dry-run
```

To upgrade Elgg simply execute

```
# to upgrade all packages
composer update

# or to only upgrade Elgg
composer update elgg/elgg
```

Versions précédentes

Check Elgg documentation that corresponds to the Elgg version you want to upgrade to, by switching the documentation version in the lower left corner of *Upgrading docs*

3.2.4 Plugins

Les plugins peuvent modifier le comportement d'Elgg et ajouter de nouvelles fonctionnalités.

Contents

Où trouver des plugins

Des plugins peuvent être obtenus depuis :
- *La Communauté Elgg*
- Github
- Sites tierce-partie (généralement pour un certain prix)

Si aucun plugin existant ne correspond à vos besoins, vous pouvez solliciter un développeur hire a developer ou créer le vôtre *create your own*.

La Communauté Elgg

Trouver des plugins

Ordre de tri basé sur la popularité

Sur la page des plugins de la communauté, vous pouvez trier par date de mise en ligne (Filtre : Newest) ou par nombre de téléchargements (Filtre : Most downloads). Trier par nombre de téléchargement est une bonne idée si vous débutez avec Elgg et voulez voir quels plugins sont fréquemment utilisés par d'autres administrateurs. Ceux-ci sont souvent (mais pas toujours) des plugins de meilleure qualité qui offrent des possibilités significatives.

Utilisez la recherche de plugin par tag

Une boite de recherche se trouve à côté des options de filtrage de la page des plugins. Elle vous permet de rechercher des plugins par tags. Les auteurs des plugins définissent les tags.

Identifiez les auteurs de plugins particuliers

La qualité des plugins varie de manière substantielle. Si vous trouvez un plugin qui fonctionne bien sur votre site, vous pouvez regarder les autres réalisations que l'auteur du plugin a développées en cliquant sur son nom lorsque vous êtes sur la page d'un plugin.

Évaluer les Plugins

Regardez les commentaires et les notations

Avant de télécharger et d'utiliser un plugin, c'est toujours une bonne idée de lire les commentaires que les autres ont laissé. Si vous voyez des personnes qui se plaignent que le plugin ne fonctionne pas ou qu'il rend leur site instable, vous devriez probablement éviter ce plugin. Néanmoins,c ertaines utilisateurs ignorent les instructions d'installation ou installent un plugin de manière incorrecte puis laissent un feedback négatif. De plus, certains auteurs de plugins choisissent de ne pas autoriser les commentaires.

Installer sur une site de test

Si vous essayez un plugin pour la première fois, c'est une mauvaise idée de l'installer sur un site de production. Vous devriez maintenir un site de test séparé pour évaluer les plugins. C'est une bonne idée de mettre en place progressivement les plugins sur votre site de production même après qu'ils aient passé l'évaluation sur un site de test. Ceci vous permet d'isoler les problèmes potentiels introduits par un nouveau plugin.

Types de plugins

Thèmes

Les thèmes sont des plugins qui modifient l'apparence et le comportement (look-and-feel) de votre site. Ils comprennent généralement des feuilles de styles, des scripts côté client et des vues qui modifient la présentation et le comportement par défaut d'Elgg.

Packs de langues

La packs de langue (ou de traduction) sont des plugins qui fournissent le support d'autres langues.

Les packs de traduction peuvent étendre et inclure des traductions pour les chaînes de traduction du noyau, des plugins du noyau et/ou de plugins tierces-parties.

Quelques packs de langues sont déjà compris dans le noyau, et se trouvent dans le dossier `languages` du répertoire racine d'Elgg. Les plugins individuels ont leurs traductions dans le répertoire `languages` de la racine du plugin.

Cette structure facilite la création de nouveaux packs de langues qui remplacent les chaines de traduction existantes ou ajoutent le support de nouvelles langues.

Installation

Tous les plugins sont dans le dossier `mod` de votre installation Elgg.

Pour installer un nouveau plugin :

- décompresser (unzip) le contenu du package de distribution du plugin
- copiez/transférez via FTP le dossier décompressé dans le dossier `mod` de votre installation Elgg, en vous assurant que `manifest.xml` est directement placé dans le dossier du plugin (par ex. si vous alliez installer un plugin appelé `my_elgg_plugin`, le manifest du plugin aurait besoin de se trouver à `mod/my_elgg_plugin/manifest.xml`)
- activez le plugin depuis votre panneau d'administration

Pour activer un plugin :

- Connectez-vous à votre site Elgg avec votre compte administrateur
- Allez dans Administration -> Configurer -> Plugins
- Trouvez votre plugin dans la liste des plugins installés et cliquez sur le bouton "activer".

Ordre des plugins

Les plugins sont chargés selon l'ordre dans lequel ils sont listés sur la page Plugins. L'ordre initial après une installation est plus ou moins aléatoire. AU fur et à mesure que de nouveaux plugins sont ajoutés par un administrateur, ils sont placés en fin de liste.

Quelques règles générales pour l'ordre des plugins :

- Un plugin de thème devrait être placé en dernier ou au moins proche du bas de la liste
- Un plugin qui modifie le comportement d'un autre plugin devrait être placé plus bas dans la liste des plugins

3.2.5 Performance

Faites que votre site fonctionne sans heurt et de manière aussi réactive que possible.

Contents

Est-ce Elgg peut passer à l'échelle de X millions d'utilisateurs ?

Les gens demandent souvent si Elgg convient pour de grands sites.

Tout d'abord, nous pourrions vous arrêter pour demander : « où et comment prévoyez-vous d'obtenir tous ces utilisateurs ? » Sérieusement, c'est un problème très intéressant. Faire évoluer Elgg est avant tout une question d'ingénierie technique. C'est intéressant, et plus ou moins un problème résolu. L'informatique ne fonctionne pas différemment pour Elgg que pour Google, par exemple. Obtenir des millions d'utilisateurs ? C'est comme le Saint Graal de toute l'industrie de la technologie.

Deuxièmement, comme avec la plupart des choses dans la vie, la réponse est «ça dépend» :

— Que font vos utilisateurs ?
— Sur quelle machine et avec quelle configuration fonctionne Elgg ?
— Est-ce que vos plugins se comportent bien ?

Améliorer l'efficacité du moteur Elgg est un projet en cours, bien qu'il y ait des limites à la quantité que n'importe quel script peut faire.

Si vous êtes sérieux au sujet de l'évolutivité, vous aurez probablement envie de considérer un certain nombre de choses vous-même.

Mesurez d'abord

Il ne sert à rien de jeter des ressources sur un problème si vous ne connaissez pas :

— quel est le problème
— de quelles ressources ce problème a besoin
— où ces ressources sont nécessaires

Investissez dans une sorte de profilage afin de vous faire connaître où votre goulot d'étranglement se situe, surtout si vous envisagez d'injecter de l'argent important sur un problème.

Régler MySQL

Elgg fait un usage intensif de la base de données en back-end, faisant, à chaque chargement de page, de nombreux appels et allers et retours entre le serveur web et la base de données. Ceci est parfaitement normal et un serveur de base de données bien configuré sera capable de faire face à des milliers de requêtes par seconde.

Voici quelques astuces de configuration qui pourraient aider :

— Assurez-vous que MySQL utilise un fichier de configuration my.cnf adapté à la taille de votre site web.
— Augmentez la quantité de mémoire disponible pour PHP et de même pour MySQL (vous aurez dans tous les cas à augmenter la quantité de mémoire disponible pour le processus de php)

Activer la mise en cache

En règle générale, si un programme est lent, c'est parce qu'elle effectue à plusieurs reprises un calcul ou une opération coûteuse. La mise en cache permet au système d'éviter de faire ce travail encore et encore à chaque demande grâce à la mise en cache des résultats précédents. En utilisant la mémoire pour stocker les résultats, il est alors facile de récupérer en mémoire le résultat de la demande et le passer aux demandes suivantes, économisant ainsi à chaque demande tout le travail de calcul. Ci-dessous, nous discutons de plusieurs solutions de mise en cache généralement disponibles et pertinentes pour Elgg.

Cache simple (Simplecache)

Par défaut, les vues sont mises en cache dans le répertoire de données de Elgg pour une période de temps donnée. Cela supprime la nécessité de régénérer chaque vue à chaque chargement de la page.

Ceci peut être désactivé en définissant `$CONFIG->simplecache_enabled = false;` Pour de meilleures performances, assurez-vous que cette valeur est bien définie sur `true`.

Cela conduit à des artefacts pendant le développement/la programmation si par exemple vous modifiez un thème dans votre plugin puisque la version mise en cache sera utilisée de préférence à la nouvelle fournie par votre plugin.

Simple cache peut être désactivée via le menu d'administration. Il est recommandé de le faire sur votre plate-forme de développement si vous écrivez des plugins Elgg.

Ce cache est automatiquement vidé à chaque fois qu'un plugin est activé, désactivé ou réorganisé, ou quand le script upgrade.php est exécuté.

Pour de meilleures performances, vous pouvez également créer un lien symbolique depuis `/cache/` dans votre répertoire racine web vers le répertoire `assetroot` spécifié dans votre configuration (par défaut il est situé dans `/path/to/dataroot/caches/views_simplecache/` :

```
cd /path/to/wwwroot/
ln -s /path/to/dataroot/caches/views_simplecache/ cache
```

Si votre serveur web supporte les liens symboliques (« symlinks »), ceci va servir les fichiers directement depuis le disque sans démarrer PHP à chaque fois.

Pour des raisons de sécurité, quelques serveurs web (par ex. Apache dans la version 2.4) pourraient par défaut ne suivre les liens symboliques que si le propriétaire de la source et de la cible du lien correspondent. Si le lien symbolique du cache ne fonctionne pas sur votre serveur web, vous pouvez changer le propriétaire du lien symbolique du cache lui-même (et non du répertoire `/views_simplecache/`) avec

```
cd /path/to/wwwroot/
chown -h wwwrun:www cache
```

Dans cet exemple on prend l'hypothèse que le répertoire `/views_simplecache/` du répertoire de données appartient au compte wwwrun qui fait partie du groupe www. Si ce n'est pas le cas sur votre serveur, vous devez modifier la commande chown en conséquence.

Cache système (System cache)

L'emplacement des vues est mis en cache afin de retrouver rapidement les vues (en effet, le profilage a indiqué que le temps de chargement d'une page prend une quantité non-linéaire du temps plus il y a de plugins activés, ceci en raison de la recherche des vues). Elgg met également en cache des informations telles que la cartographie de la langue et de la carte des classes.

Ceci peut être désactivé en définissant `$CONFIG->system_cache_enabled = false;` Pour de meilleures performances, assurez-vous que cette valeur est bien définie sur `true`.

Les emplacements des vues sont actuellement stockés dans des fichiers placés dans votre dossier de données (toutefois des versions ultérieures de Elgg peuvent utiliser memcache). Comme avec le cache simple, le cache système est automatiquement vidé chaque fois qu'un plugin est activé, désactivé ou réorganisé, ou quand le script upgrade.php est exécuté.

Le cache du système peut être désactivé via le menu d'administration, et il est recommandé de le faire sur votre plate-forme de développement si vous écrivez des plugins Elgg.

Cache de démarrage

Elgg a la capacité de mettre en cache de nombreuses ressources créées et récupérées pendant le processus de démarrage. Pour configurer la durée de validité de ce cache vous devez définir la valeur du TTL dans votre fichier `settings.php` : `$CONFIG->boot_cache_ttl = 3600;`

Regardez les documentation de Stash pour plus d'informations sur le TTL.

Cache des requêtes de base de données

Pour toute la durée d'exécution d'une page donnée, un cache de toutes les requêtes `SELECT` est conservé. Ceci signifie que pour le chargement d'une page donnée une requête select donnée sera envoyée à la base de données une seule et unique fois, même si elle est exécutée à plusieurs reprises. Toute écriture sur la base de données va vider ce cache, aussi il est conseillé que sur des pages complexes vous différiez les écriture sur la base de données après la fin de la page, ou utilisiez la fonctionnalité `execute_delayed_*`. Ce cache sera automatiquement effacé à la fin du chargement de la page.

Vous pouvez rencontrer des problèmes de mémoire si vous utilisez le framework Elgg comme bibliothèque dans un script PHP CLI. Ceci peut être désactivé en définissant `$CONFIG->db_disable_query_cache = true;`

Etags et headers Expires

Ces technologies indiquent aux navigateurs de vos utilisateurs de mettre en cache des ressources statiques (CSS, JS, images) localement. Avoir ceci activé réduit fortement la charge du serveur et améliore la performance perçue par les utilisateurs.

Utilisez le plugin Firefox yslow ou les Outils de Développement de Chrome DevTools Audits pour confirmer les technologies utilisées actuellement sur votre serveur.

Si les ressources statiques ne sont pas mises en cache :

- Vérifiez que vous avez ces extensions et activées sur votre hébergement
- Mettez à jour votre fichier .htaccess, si vous faites une mise à niveau depuis une version précédente d'Elgg
- Activez *Simplecache*, qui transforme les vues sélectionnées en ressources (assets) pouvant être mis en cache par le navigateur

Memcached

Libmemcached a été créé par Brian Aker et a été conçu dès le départ pour offrir la meilleure performance possible aux utilisateurs de Memcached.

Voir aussi :

http://libmemcached.org/About.html and https://secure.php.net/manual/en/book.memcached.php

Pré-requis pour l'installation :

- php-memcached
- libmemcached
- memcached

Configuration :

Dé-commentez et renseignez les sections suivantes dans le fichier `settings.php`

```
$CONFIG->memcache = true;

$CONFIG->memcache_servers = array (
    array('server1', 11211),
    array('server2', 11211)
);
```

De manière optionnelle, si vous avez plusieurs installations Elgg mais n'utilisez qu'un seul serveur Memcache, vous pouvez vouloir ajouter un préfixe d'espace de nom (namespace prefix). Afin de faire cela, décommentez la ligne suivante

```
$CONFIG->memcache_namespace_prefix = '';
```

Squid

Nous avons obtenu de bons résultats en utilisant Squid pour mettre en cache les images.

Mise en cache du Bytecode

Il existe de nombreux caches de code disponibles sur le marché. Ceux-ci accélèrent votre site en mettant en cache le byte code compilé depuis votre script, ce qui signifie que votre serveur n'a pas à compiler le code PHP à chaque fois qu'il est exécuté.

Servir directement les fichiers

Si votre serveur peut être configuré pour supporter les headers X-Sendfile ou X-Accel, vous pouvez configurer leur utilisation dans `settings.php`. Ceci permet à votre serveur web de diffuser directement les fichiers au client au lieu d'utiliser la focntion PHP `readfile()`.

Composer Autoloader Optimization

The Composer autoloader is responsible for loading classes provided by dependencies of Elgg. The way the autoloader works is it searches for a classname in the installed dependencies. While this is mostly a fast process it can be optimized.

You can optimize the autoloader 2 different ways. The first is in the commandline, the other is in the `composer.json` of your project.

If you want to optimize the autoloader using the commandline use the `-o` flag. The disadvantage is you have to add the `-o` flag every time you run Composer.

```
# During the installation
composer install -o

# Or during the upgrade process
composer upgrade -o
```

The second option is to add the optimization to your `composer.json` file, that way you never forget it.

```
{
        "config": {
                "optimize-autoloader": true,
                "apcu-autoloader": true
        }
}
```

Voir aussi :

Check out the Autoloader Optimization page for more information about how to optimize the Composer autoloader.

Note : As of Elgg 3.0 all the downloads of Elgg from the website have the optimized autoloader.

Hébergement

N'espérez pas de faire tourner un site pour des millions d'utilisateurs sur un hébergement mutualisé à bas prix. Vous aurez besoin de votre propre serveur d'hébergement et d'avoir la main sur la configuration, ainsi que de beaucoup de bande passante et de mémoire disponibles.

Mémoire, CPU et bande passante

De par la nature de la mise en cache, toutes les solutions de mise en cache auront besoin de mémoire. C'est un investissement plutôt économique d'augmenter la mémoire et le CPU.

Sur un matériel puissant il est probable que la bande passante soit le goulet d'étranglement (bottleneck) avant le serveur lui-même. Assurez-vous que votre hébergement peut supporter le trafic que vous attendez.

Configuration

Enfin, jetez un coup d'œil à votre configuration car il y a quelques points d'attention qui peuvent surprendre les gens.

Par exemple, Apache peut d'emblée gérer une charge plutôt élevée. Cependant, la majorité des distributions Linux sont livrées avec mysql configuré pour de petits sites. Ceci peut peut donner lieu à des processus Apache bloqués qui attendent de pouvoir parler à un processus MySQL très surchargé.

Vérifiez les plugins au mauvais comportement

Des plugins peuvent être développés d'une manière très naïve et ceci peut ralentir l'ensemble du site.

Essayez de désactiver quelques plugins pour voir si cela améliore notablement les performances. Une fois que vous avez trouvé un responsable potentiel, rendez-vous sur la page de l'auteur du plugin et signalez vos résultats.

Utilisez du HTML rendu côté client

Nous avons constaté qu'à un certain niveau, une bonne partie du temps passé sur le serveur est simplement le temps de construction du HTML de la page avec le système de vues d'Elgg.

Il est très difficile de mettre en cache les sorties des modèles (templates) dans la mesure où elles prennent généralement des entrées arbitraires. Au lieu d'essayer mettre en cache la sortie de certaines pages ou vues, la suggestion est de basculer vers un système de modèles basé sur HTML de sorte que le navigateur de l'utilisateur puisse lui-même mettre en cache les modèles. Puis demandez à l'ordinateur de l'utilisateur de générer le résultat en appliquant des données JSON à ces modèles.

Ceci peut être très efficace, mais présente l'inconvénient de demander un coup de développement supplémentaire significatif. L'équipe d'Elgg envisage d'intégrer cette stratégie directemetn dans Elgg, dans la mesure où ceci est aussi efficace, tout particulièrement avec les pages avec du contenu répété ou caché.

3.2.6 Cron

Contents

Qu'est-ce que ça fait ?

`Cron`__ est un programme disponible sur les systèmes d'exploitation basés sur Unix qui permet aux utilisateurs d'exécuter des commandes et des scripts à des intervales particuliers ou à des heures spécifiques.

Le gestionnaire de cron d'Elgg permet aux administrateurs et aux développeurs de plugins de mettre en place des tâches qui ont besoin d'être exécutées à des intervalles définis.

Les exemples les plus courants de tâches cron dans Elgg incluent :
— l'envoi des notifications en file d'attente
— la rotation des journaux système dans la base de données
— le nettoyage des déchets dans la base de données (compactage de la base de données par la suppression des entrées qui ne sont plus requises)

Les plugins peuvent ajouter des tâches en enregistrant un gestionnaire de hook plugin pour l'un des intervalles de cron suivants :
— `minute` - Exécuté toutes les minutes
— `fiveminute` - Exécuté toutes les 5 minutes
— `fifteenmin` - Exécuté toutes les 15 minutes
— `halfhour` - Exécuté toutes les 30 minutes
— `hourly` - Exécuté toutes les heures
— `daily` - Exécuté tous les jours
— `weekly` - Exécuté toutes les semaines
— `monthly` - Exécuté tous les mois
— `yearly` - Exécuté tous les ans

```
elgg_register_plugin_hook_handler('cron', 'hourly', function() {

    $events = my_plugin_get_upcoming_events();

    foreach ($events as $event) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $attendees = $event->getAttendees();

        // notify
    }
});
```

Comment ça marche ?

`crontab` doit être configuré de telle manière qu'il active le gestionnaire de cron d'Elgg chaque minute, ou à un intervalle spécifique. Une fois que l'entrée du cron active le travail du cron, Elgg exécute chaque gestionnaire de hook attaché à cet intervalle.

Si vous avez un accès SSH à vos serveurs Linux, tapez `crontab -e` et ajoutez votre configuration crontab.

```
* * * * * path/to/phpbin path/to/elgg/elgg-cli cron -q
```

La commande suivante va s'exécuter chaque minute et activer toutes les tâches du cron attendues.

Facultativement, vous pouvez activer les gestionnaire pour un intervalle spécifique :

```
0 * * * * path/to/phpbin path/to/elgg/elgg-cli cron -i hourly -q
```

Vous pouvez trouver plus d'informations sur cron sur :

3.2.7 Sauvegarde et Restauration

Contents

Introduction

Pourquoi

Les fournisseurs d'hébergements mutualisés ne fournissent généralement pas de moyen automatisé de sauvegarder votre installation Elgg. Cet article propose une méthode permettant d'accomplir cette tâche.

Dans l'IT il existe souvent de nombreuses manières d'accomplir la même chose. Gardez cela à l'esprit. Cet article va expliquer une méthode pour sauvegarder et restaurer votre installation Elgg sur un hébergement mutualisé qui utilise l'application CPanel. Cependant, les idées présentées ici peuvent être ajustées pour d'autres applications également. Ce qui suit sont des situations typiques qui peuvent exiger une procédure telle que celle-ci :

- Récupération après une catastrophe
- Déplacer votre site Elgg vers un nouvel hôte
- Dupliquer une installation

Quoi

Sujets traités :

- Des backups complets des répertoires d'Elgg et des bases de données MySQL sont effectués tous les jours (automatisé)
- Les backups sont envoyés sur un autre site via FTP (automatisé)
- Les backups locaux sont supprimés après un transfert réussi vers le site distant (automatisé)
- Cinq jours de backups seront conservés (automatisé)
- Restauration des données sur le nouvel hôte (manuel)

Ce processus a été composé à partir d'articles d'assistance précédemment publiés dans le wiki de documentation d'Elgg.

Hypothèses

Les hypothèses suivantes ont été faites :

- Le répertoire du programme Elgg est `/home/userx/public_html`
- Le répertoire de données Elgg est `/home/userx/elggdata`
- Vous avez créé un répertoire local pour vos backups à `/home/userx/sitebackups`
- Vous disposez d'un server FTP distant auquel envoyer les fichiers de backup
- Le répertoire dans lequel vous allez enregistrer les backups distants est `/home/usery/sitebackups/`
- Vous allez restaurer le site chez un deuxième fournisseur d'hébergement mutualisé dans le répertoire `/home/usery/public_html`

Important : Assurez-vous de remplacer `userx`, `usery`, `http://mynewdomain.com` et tous les mots de passe par des valeurs qui correspondent à votre propre installation !

Créer un backup utilisable - automatiquement

Personnalisez le script de sauvegarde automatique (backup)

Le script que vous allez utiliser peut être trouvé *ici* .

Copiez simplement le script dans un fichier texte et renommez le fichier avec une extension .pl. Vous pouvez utiliser n'importe quel éditeur de texte pour modifier le fichier.

Modifiez les valeurs suivantes pour qu'elles correspondent à votre structure de répertoires :

```
# ENTER THE PATH TO THE DIRECTORY YOU WANT TO BACKUP, NO TRAILING SLASH
$directory_to_backup = '/home/userx/public_html';
$directory_to_backup2 = '/home/userx/elggdata';
# ENTER THE PATH TO THE DIRECTORY YOU WISH TO SAVE THE BACKUP FILE TO, NO TRAILING␣
↪SLASH
$backup_dest_dir = '/home/userx/sitebackups';
```

Modifiez les valeurs suivantes pour qu'elles correspondent à vos paramètres de base de données :

```
# MYSQL BACKUP PARAMETERS
$dbhost = 'localhost';
$dbuser = 'userx_elgg';
$dbpwd = 'dbpassword';
# ENTER DATABASE NAME
$database_names_elgg = 'userx_elgg';
```

Modifiez les valeurs suivantes pour qu'elles correspondent aux paramètres de votre serveur FTP distant :

```
# FTP PARAMETERS
$ftp_host = "FTP HOSTNAME/IP";
$ftp_user = "ftpuser";
$ftp_pwd = "ftppassword";
$ftp_dir = "/";
```

Enregistrez le fichier avec l'extension `.pl` (dans le cadre de cet article nous appellerons ce fichier : `elgg-ftp-backup-script.pl`) et chargez-le dans le répertoire `/home/userx/sitebackups`

Soyez conscient(e) que vous pouvez désactiver le FTP et modifier un peu le script de sorte qu'il ne supprime pas le fichier de backup local dans le cas où vous ne voulez pas utiliser de stockage distant pour vos backups.

Configurez la tâche périodique (cron) de backup

Connectez-vous à votre application CPanel et cliquez sur le lien « Cron Jobs ». Dans le menu déroulant Paramètres communs (Common Settings), choisissez « Une fois par jour » (« Once a day ») et tapez ce qui suit dans le champ de commande `/usr/bin/perl /home/userx/sitebackups/elgg-ftp-backup-script.pl`

Cliquez sur le bouton « Add New Cron Job ». Des sauvegardes quotidiennes complètes sont désormais planifiées et seront transférées hors du site.

Configurez la tâche périodique (cron) de nettoyage

Si vous envoyez vos sauvegardes, via FTP, à un autre hébergeur mutualisé qui utilise l'application CPanel ou si vous avez totalement désactivé FTP, vous pouvez configurer votre conservation des données comme suit.

Connectez-vous à votre application CPanel pour votre site FTP, ou localement si vous n'utilisez pas FTP, et cliquez sur le lien « Cron Jobs ». Dans le menu déroulant Paramètres communs (Common Settings) choisissez Une fois par jour (« Once a day ») et saisissez ce qui suit dans le champ de commande `find /home/usery/sitebackups/full_* -mtime +4 -exec rm {} \;`

Le paramètre `-mtime X` va définir le nombre de jours de rétention des sauvegardes. Tous les fichiers plus anciens que le nombre de jour `x` sera supprimé. Cliquez sur le bouton « Add New Cron Job ». Vous avez maintenant configuré la durée de rétention de vos sauvegardes.

Restauration depuis un backup

Préparez vos fichiers de backup

L'hypothèse est que vous restaurez votre site vers un autre prestataire d'hébergement mutualisé avec CPanel.

Lorsque le script a sauvegardé les fichiers, la structure originelle des répertoires est conservée dans le fichier zip. Nous devons faire un petit nettoyage. Effectuer ce qui suit :

- Téléchargez le fichier de backup à partir duquel vous souhaitez effectuer la restauration
- Décompressez le contenu du fichier de backup
- **Naviguez pour trouver la sauvegarde de votre site et la sauvegarde SQL. Extrayez les deux archives. Vous aurez alors :**
  - un dump MySQL avec une extension « .sql »
  - **une autre structure de répertoire avec pour contenu :**
    - « /home/userx/public_html »
    - `/home/userx/elggdata`
- **Re-compressez le contenu du répertoire `/home/userx/public_html` sous forme de fichier zip de sorte que les fichier**
  - La raison de cette démarche est simple. Il est beaucoup plus efficace de télécharger un seul fichier zip que d'envoyer par FTP sur votre nouvel hôte tout le contenu du répertoire `/ home/userx/public_html`.
- Re-compressez le contenu du répertoire /home/userx/elggdata directory sous forme de fichier zip de sorte que les fichiers soient à la racine du fichier zip

Vous devriez maintenant avoir les fichiers suivants :

- le fichier « .sql »
- le fichier zip avec le contenu de `/home/userx/public_html` à la racine
- le fichier zip avec le contenu de */home/userx/elggdata*` à la racine

Restaurez les fichiers

Ceci est écrit dans l'hypothèse que vous restaurez vers un hébergement différent mais conservez la même structure de répertoires. Faites comme suit :

- Connectez-vous à l'application CPanel de l'hébergement sur lequel vous souhaitez restaurer le site et ouvrez le gestionnaire de fichiers (File Manager).
- **Naviguez vers `/home/usery/public_html`**
  - Chargez le fichier zip qui contient les fichiers de `/home/userx/public_html`
  - **Décompressez le fichier zip** Vous devriez maintenant voir tous les fichiers dans `/home/usery/public_html`
  - Supprimez le fichier zip

- **Naviguez vers `/home/usery/elggdata`**
  - Chargez le fichier zip qui contient les fichiers de `/home/userx/elggdata`
  - **Décompressez le fichier zip** Vous devriez maintenant voir tous les fichiers dans /home/usery/elggdata
  - Supprimez le fichier zip

La restauration des fichiers du programme et des données est terminée

Restaurez la base de données MySQL

Note : A nouveau, l'hypothèse ici est que vous restaurez votre installation Elgg vers un autre fournisseur d'hébergement mutualisé. Chaque fournisseur d'hébergement mutualisé ajoute le nom du compte du détenteur comme préfixe aux bases de données associées avec ce compte. Par exemple, le nom d'utilisateur de notre premier hébergement est `userx` de sorte que l'hébergeur va ajouter le préfixe `userx_` pour nous donner un nom de base de données de `userx_elgg`. Quand nous restaurons vers notre second fournisseur d'hébergement mutualisé nous le faisons avec un nom d'utilisateur de `usery` de sorte que le nom de notre base de données sera `usery_elgg`. Les fournisseurs d'hébergement ne vous permettent pas de modifier ce comportement. De sorte que le processus ici n'est pas aussi simple que simplement restaurer la base de données depuis le backup vers le compte usery. Néanmoins, une fois ceci dit, ce n'est pas si difficile non plus.

Modifiez le backup MySQL

Ouvrez le fichier``.sql`` que vous avez extrait de votre backup dans votre éditeur de texte favori. Commentez les lignes suivantes avec un dièse :

```
#CREATE DATABASE /*!32312 IF NOT EXISTS*/ `userx_elgg` /*!40100 DEFAULT CHARACTER SET␣
↪latin1 */;
#USE `userx_elgg`;
```

Enregistrez le fichier.

Créez la nouvelle base de données

Effectuez les actions suivantes :

- **Connectez-vous à l'application CPanel du nouvel hôte et cliquez sur l'icône « Bases de données MySQL (« MySQL Datal**
  - Renseignez le nom de la base de données et cliquez sur le bouton créer (« create »). Pour notre exemple, nous allons conserver `elgg` ce qui nous donne pour nom de base de données `usery_elgg`
  - **Vous pouvez associer un utilisateur existant à la nouvelle base de données, ou pour créer un nouvel utilisateur vou**
    - Allez dans la section « Ajouter Nouvel utilisateur » de la page « Bases de données MySQL »
    - Saisissez le nom d'utilisateur et le mot de passe. Pour notre exemple nous allons rester simples et conserver à nouveau `elgg`. Ce qui va nous donner un nom d'utilisateur `usery_elgg`
  - **Associez le nouvel utilisateur avec la nouvelle base de données**
    - Allez dans la section Ajouter un nouvel utilisateur à la base de données (« Add User To Database ») de la page Bases de données MySQL (« MySQL Databases »). Ajoutez l'utilisateur `usery_elgg` à la base de données `usery_elgg`
    - Sélectionnez « Tous les Privilèges » (« All Privileges ») et cliquez sur le bouton « Effectuer les changements » (« Make Changes »)

Restaurez la base de données de production

Maintenant il est temps de restaurer le fichier de backup MySQL en l'important dans notre nouvelle base de données nommée « usery_elgg ».

- **Connectez-vous à l'application CPanel du nouvel hôte et cliquez sur l'« icône phpMyAdmin**
  - Choisissez la base de données `usery_elgg` dans la colonne de gauche
  - Cliquez sur l'onglet « import » en haut de la page
  - Naviguez jusqu'au backup `.sql` dans votre arborescence locale et sélectionnez-le
  - Cliquez sur le bouton « Go » en bas à droite de la page

Vous devriez maintenant voir un message indiquant que l'opération avec succès

Combiner l'ensemble

L'installation elgg restaurée ne connait **rien** du nom de la nouvelle base de données, du nom d'utilisateur, de la structure des répertoires, etc. C'est ce dont nous allons traiter ici.

Modifiez `/public_html/elgg-config/settings.php` sur le nouvel hébergement pour reproduire les informations de la base de données que vous venez de créer.

```
// Database username
$CONFIG->dbuser = 'usery_elgg';

// Database password
$CONFIG->dbpass = 'dbpassword';

// Database name
$CONFIG->dbname = 'usery_elgg';

// Database server
// (For most configurations, you can leave this as 'localhost')
$CONFIG->dbhost = 'localhost';

$CONFIG->wwwroot = 'http://your.website.com/'
```

Chargez à nouveau le fichier `settings.php` vers le nouvel hôte - en remplaçant le fichier existant.

Ouvrez l'outil phpMyAdmin sur le nouvel hôte depuis CPanel. Sélectionnez la base de données `usery_elgg` sur la gauche, et cliquez sur l'onglet SQL en haut de la page. Exécutez les requêtes SQL suivantes sur la base de données `usery_elgg` :

Modifiez le chemin d'installation

```
UPDATE `elgg_config` SET `value` = REPLACE(`value`, "/home/userx/public_html/grid/",
↪"/home/usery/public_html/grid/") WHERE `name` = "path";
```

Modifiez le répertoire de données

```
UPDATE `elgg_config` SET `value` = REPLACE(`value`, "/home/userx/elggdata/", "/home/
↪usery/elggdata/") WHERE `name` = "dataroot";
```

Modifiez le répertoire de données de stockage des fichiers

```
UPDATE elgg_metadata set value = '/home/usery/elggdata/' WHERE name = 'filestore::dir_
↪root';
```

Finaliser la nouvelle installation

Exécutez le script de mise à niveau en visitant l'URL suivante : `http://mynewdomain.com/upgrade.php` . Effectuez cette étape deux fois - depuis le début.

Mettez à jour vos enregistrements DNS de sorte que votre nom d'hôte se résolve toujours vers l'adresse IP du nouvel hôte si c'est un déplacement permanent.

Félicitations !

Si vous avez suivi les étapes soulignées ici vous devriez maintenant avoir une copie totalement fonctionnelle de votre installation Elgg d'origine.

Connexe

Script de backup FTP

Voici un script automatisé pour sauvegarder une installation Elgg.

```
#!/usr/bin/perl -w

# FTP Backup

use Net::FTP;

# DELETE BACKUP AFTER FTP UPLOAD (0 = no, 1 = yes)
$delete_backup = 1;

# ENTER THE PATH TO THE DIRECTORY YOU WANT TO BACKUP, NO TRAILING SLASH
$directory_to_backup = '/home/userx/public_html';
$directory_to_backup2 = '/home/userx/elggdata';

# ENTER THE PATH TO THE DIRECTORY YOU WISH TO SAVE THE BACKUP FILE TO, NO TRAILING␣
↪SLASH
$backup_dest_dir = '/home/userx/sitebackups';

# BACKUP FILE NAME OPTIONS
($a,$d,$d,$day,$month,$yearoffset,$r,$u,$o) = localtime();
$year = 1900 + $yearoffset;
$site_backup_file = "$backup_dest_dir/site_backup-$day-$month-$year.tar.gz";
$full_backup_file = "$backup_dest_dir/full_site_backup-$day-$month-$year.tar.gz";

# MYSQL BACKUP PARAMETERS
$dbhost = 'localhost';
$dbuser = 'userx_elgg';
$dbpwd = 'dbpassword';
$mysql_backup_file_elgg = "$backup_dest_dir/mysql_elgg-$day-$month-$year.sql.gz";

# ENTER DATABASE NAME
$database_names_elgg = 'userx_elgg';

# FTP PARAMETERS
$ftp_backup = 1;
$ftp_host = "FTP HOSTNAME/IP";
$ftp_user = "ftpuser";
```

(suite sur la page suivante)

(suite de la page précédente)

```
$ftp_pwd = "ftppassword";
$ftp_dir = "/";

# SYSTEM COMMANDS
$cmd_mysqldump = '/usr/bin/mysqldump';
$cmd_gzip = '/usr/bin/gzip';

# CURRENT DATE / TIME
($a,$d,$d,$day,$month,$yearoffset,$r,$u,$o) = localtime();
$year = 1900 + $yearoffset;

# BACKUP FILES
$syscmd = "tar --exclude $backup_dest_dir" . "/* -czf $site_backup_file $directory_to_
↪backup $directory_to_backup2";

# elgg DATABASE BACKUP
system($syscmd);
$syscmd = "$cmd_mysqldump --host=$dbhost --user=$dbuser --password=$dbpwd --add-drop-
↪table --databases $database_names_elgg -c -l | $cmd_gzip > $mysql_backup_file_elgg";

system($syscmd);

# CREATING FULL SITE BACKUP FILE
$syscmd = "tar -czf $full_backup_file $mysql_backup_file_elgg $site_backup_file";
system($syscmd);

# DELETING SITE AND MYSQL BACKUP FILES
unlink($mysql_backup_file_elgg);
unlink($site_backup_file);

# UPLOADING FULL SITE BACKUP TO REMOTE FTP SERVER
if($ftp_backup == 1)
{
   my $ftp = Net::FTP->new($ftp_host, Debug => 0)
      or die "Cannot connect to server: $@";

   $ftp->login($ftp_user, $ftp_pwd)
      or die "Cannot login ", $ftp->message;

   $ftp->cwd($ftp_dir)
      or die "Can't CWD to remote FTP directory ", $ftp->message;

   $ftp->binary();

   $ftp->put($full_backup_file)
      or warn "Upload failed ", $ftp->message;

   $ftp->quit();
}

# DELETING FULL SITE BACKUP
if($delete_backup = 1)
{
   unlink($full_backup_file);
}
```

Dupliquer une installation

Contents

Introduction

Pourquoi dupliquer une installation Elgg ?

Il existe beaucoup de raisons pour lesquelles vous pouvez vouloir dupliquer une installation Elgg : déplacer le site vers un autre serveur, créer un serveur de test ou de développement, et créer des backups fonctionnels sont les plus habituelles. Pour créer un double fonctionnel d'un site Elgg, 3 choses doivent être copiées :

- Base de données
- Données du répertoire de données
- Code

Il faut modifier au moins 5 éléments d'information pour une installation copiée :

- le fichier `elgg-config/settings.php` qui peut aussi se trouver à l'emplacement `engine/settings.php` pour les versions pré-2.0
- le fichier `.htaccess` (Apache) ou la configuration de Nginx en fonction du serveur utilisé
- l'entrée de la base de donnée pour l'entité de votre site
- l'entrée de la base de donnée pour le chemin d'installation
- l'entrée de la base de donnée pour le chemin des données

Ce qui n'est pas traité dans ce tutoriel

Ce tutoriel nécessite une connaissance minimale d'Apache, MySQL, et des commandes Linux. A ce titre, certaines choses ne seront pas couvertes dans ce tutoriel. Ceci comprend :

- Comment sauvegarder (backup) et restaurer des bases de données MySQL
- Comment configurer Apache pour fonctionner avec Elgg
- Comment transférer des fichiers vers et depuis votre serveur de production

Avant de commencer

Avant de commencer, assurez-vous que votre installation Elgg est pleinement fonctionnelle. Vous aurez aussi besoin des éléments suivants :

— Un backup de la base de données de votre site Elgg actif
— Un emplacement pour copier la base de données active
— **Un serveur qui convienne pour installer le site Elgg dupliqué** (Qui peut être le même serveur que celui de votre installation Elgg en production)

Les backups de la base de données peuvent être obtenus de diverses manières, y compris via phpMyAdmin, l'interface officielle de MySQL, et la ligne de commande. Contactez votre hébergeur pour des informations sur comment sauvegarder et restaurer des bases de données, ou utilisez un moteur de recherche pour trouver de l'information à ce sujet.

Au cours de ce tutoriel, nous allons faire les hypothèses suivante à propos du site Elgg en production :

— L'URL est `http://www.myelgg.org/`
— Le chemin d'installation est `/var/www/elgg/`
— Le chemin du répertoire de données est `/var/data/elgg/`
— L'hôte de la base de données est `localhost`
— Le nom de la base de données est `production_elgg`
— L'utilisateur de la base de données est `db_user`
— Le mot de passe de la base de données est `db_password`
— Le préfixe des tables de la base de données est `elgg`

A la fin du tutoriel, les renseignements de notre installation Elgg de test seront :

— L'URL est `http://test.myelgg.org/`
— Le chemin d'installation est `/var/www/elgg_test/`
— Le répertoire de données est `/var/data/elgg_test/`
— L'hôte de la base de données est `localhost`
— Le nom de la base de données est `test_elgg`
— L'utilisateur de la base de données est `db_user`
— Le mot de passe de la base de données est `db_password`
— Le préfixe des tables de la base de données est `elgg`

Copiez le code d'Elgg vers le serveur de test

La toute première étape est de dupliquer le code du site Elgg de production. Dans notre exemple, c'est aussi simple que de copier `/var/www/elgg/` vers `/var/www/elgg_test/`.

```
cp -a /var/www/elgg/ /var/www/elgg_test/
```

Copiez les données d'Elgg vers le serveur de test

Dans cet exemple, c'est aussi simple que copier `/var/data/elgg/` vers `/var/data/elgg_test/`.

```
cp -a /var/data/elgg/ /var/data/elgg_test/
```

Si vous n'avez pas d'accès console (shell) à votre serveur et devez transférer les données par FTP, vous pouvez avoir besoin de modifier le propriétaire et les permissions des fichiers.

Note : Vous devez aussi supprimer les dossier de cache du disque. Ceux-ci correspondent aux dossiers `cacheroot` et `assetroot` dans votre configuration.

Editez settings.php

Le fichier `elgg-config/settings.php` contient les renseignement de configuration de la base de données. Ceux-ci doivent être ajustés pour votre nouvelle installation Elgg de test. Dans notre exemple, nous allons regarder dans `/var/www/elgg_test/elgg-config/settings.php` et trouver les lignes qui ressemblent à ceci :

```
// Database username
$CONFIG->dbuser = 'db_user';

// Database password
$CONFIG->dbpass = 'db_password';

// Database name
$CONFIG->dbname = 'elgg_production';

// Database server
// (For most configurations, you can leave this as 'localhost')
$CONFIG->dbhost = 'localhost';

// Database table prefix
// If you're sharing a database with other applications, you will want to use this
// to differentiate Elgg's tables.
$CONFIG->dbprefix = 'elgg';
```

Nous devons modifier ces lignes pour qu'elles correspondent à notre nouvelle installation :

```
// Database username
$CONFIG->dbuser = 'db_user';

// Database password
$CONFIG->dbpass = 'db_password';

// Database name
$CONFIG->dbname = 'elgg_test';

// Database server
// (For most configurations, you can leave this as 'localhost')
$CONFIG->dbhost = 'localhost';

// Database table prefix
// If you're sharing a database with other applications, you will want to use this
// to differentiate Elgg's tables.
$CONFIG->dbprefix = 'elgg';

$CONFIG->wwwroot = 'http://your.website.com/'
```

Note : Notez que `$CONFIG->dbname` a changé pour correspondre à notre nouvelle base de données.

Copier la base de données d'Elgg

Maintenant la base de données doit être copiée depuis `elgg_production` vers `elgg_test`. Voyez la documentation de votre gestionnaire MySQL préféré pour savoir comment dupliquer une base de données. Vous allez généralement exporter les tables de la base de donnée actuelle vers un fichier, créer la nouvelle base de données, puis importer les tables que vous avez précédemment exportées.

Vous avez deux possibilités pour mettre à jour les valeurs dans la base de données. Vous pourriez changer les valeurs dans le fichier exporté, ou vous pourriez modifier ces valeurs par des requêtes sur la base de données. Un avantage de modifier directement le fichier exporté est que vous pouvez aussi modifier les liens que les utilisateurs ont créé vers du contenu au sein de votre site. Par exemple, si des personnes ont mis en marque-page des pages avec le plugin bookmarks, les marque-pages vont pointer vers l'ancien site tant que vous n'aurez pas mis à jour leurs URLs.

Entrées de la base de données

Nous devons maintenant modifier 4 entrées dans la base de données. Ceci peut être fait aisément en exécutant 4 commandes SQL simples :

Modifiez le chemin d'installation

```
UPDATE `elgg_config` SET `value` = REPLACE(`value`, "/var/www/elgg_production/", "/
↪var/www/elgg_test/") WHERE `name` = "path";
```

Modifiez le répertoire de données

```
UPDATE `elgg_config` SET `value` = REPLACE(`value`, "/var/data/elgg_production/", "/
↪var/data/elgg_test/") WHERE `name` = "dataroot";
```

Vérifier le fichier .htaccess

Si vous aviez apporté des modifications au fichier .htaccess qui modifient des chemins, assurez-vous que vous les mettez à jour dans l'installation de test.

Mettez à jour la configuration du serveur web

Pour cet exemple, vous devez modifier la config Apache pour activer un sous-domaine avec pour racine des documents (document root) `/var/www/elgg_test/`. Si vous prévoyez d'installer dans un sous-répertoire de votre racine des documents, cette étape n'est pas nécessaire.

Si vous utilisez Nginx, vous aurez besoin de mettre à jour la config du serveur pour correspondre aux nouveaux chemins basés sur `install/config/nginx.dist`.

Exécutez upgrade.php

Pour régénérer les données mises en cache, assurez-vous d'exécuter `http://test.myelgg.org/upgrade.php`

Astuces

C'est une bonne idée de garder un serveur de test à portée de main pour expérimenter l'installation de nouveaux plugins et faire du développement. Si vous automatisez les restaurations de la base de données `elgg_test`, modifier les valeurs de `$CONFIG` et ajouter les lignes suivantes à la fin du fichier `elgg_test/elgg-config/settings.php` va permettre de réécrire sans couture les entrées de la base de données MySQL.

```
$con = mysql_connect($CONFIG->dbhost, $CONFIG->dbuser, $CONFIG->dbpass);
mysql_select_db($CONFIG->dbname, $con);

$sql = "UPDATE {$CONFIG->dbprefix}config
   SET value = REPLACE(`value`, "/var/www/elgg_production/", "/var/www/elgg_test/")
   WHERE name = 'path'";
mysql_query($sql);
print mysql_error();

$sql = "UPDATE {$CONFIG->dbprefix}config
   SET value = REPLACE(`value`, "/var/data/elgg_production/", "/var/data/elgg_test/")
   WHERE name = 'dataroot'";
mysql_query($sql);
print mysql_error();
```

Connexe

Voir aussi :

Sauvegarde et Restauration

3.2.8 Trouver de l'aide

Vous avez un problème avec Elgg ? Le meilleur moyen d'obtenir de l'aide est de demander sur le Site de la Communité. Ce site communautaire est animé par un grand nombre de volontaires. Voici quelques astuces pour vous aider à trouver l'aide dont vous avez besoin.

Contents

Trouver de l'aide

Ne soyez pas un `Vampire de l'Aide`_

Nous avons tous été des débutants à un moment, mais nous pouvons apprendre. Le fait de ne pas montrer que vous faites des tentatives pour apprendre par vous-même ou faites vos propres recherches est décourageant pour ceux qui aident. De plus, il est quasiment impossible de répondre à des questions très génériques comme « Comment je construis un forum ? ».

Faites d'abord des recherches

Assurez-vous de faire des recherches dans la documentation (ce site), sur le Site de la Communauté, et dans un moteur de recherche avant de poser une question. Les nouveaux utilisateurs d'Elgg se posent souvent les mêmes questions, aussi merci de faire des recherches. Les membres sont moins enclins à répondre à une demande à laquelle il a déjà été répondu maintes fois ou à laquelle on peut facilement trouver une réponse via un moteur de recherche.

Demandez une seule fois

Poser les mêmes questions à plusieurs endroits rend plus difficile le fait de vous répondre. Posez votre questions à un seul endroit. Les questions dupliquées peuvent être modérées.

Indiquez la version d'Elgg

Les différentes versions d'Elgg ont différentes fonctionnalités (et différents bugs). Préciser la version d'Elgg que vous utilisez va aider les personnes qui aident.

Ayez un profil raisonnable

Les profils qui ressemblent à du spam ou ont des noms loufoques seront souvent ignorés. La jovialité est bienvenue, mais les membres ont plus de chance d'aider Michael que 1337elggHax0r.

Publiez dans le forum approprié

Assurez-vous de publier dans le bon forum. Si vous avez une question sur comment créer un plugin, n'écrivez pas dans le forum Elgg Feedback. Si vous avez besoin d'aide pour installer Elgg, écrivez dans le groupe Technical Support (support technique) et pas dans le groupe Theming (thèmes).

Utilisez un titre de sujet descriptif

De bons titres de sujets décrivent de manière concise votre problème ou question. De mauvais titres sont vagues, ne contiennent que les lettres capitales, et une ponctuation excessive.

Bon titre : « Écran blanc après la mise à niveau vers 1.7.4 »

Mauvais titre : « URGENT ! ! ! ! site cassé ;-(je perds de l'argent à l'aide ! ! ! ! ! ! ! ! ! ! »

Soyez détaillé

Ajoutez autant de détails que possible à propos de votre problème. Si vous avez un site en ligne, incluez un lien. Soyez accueillant si des membres de la communauté vous demandent plus d'informations. Nous ne pouvons pas vous aider si vous ne donnez pas de détails !

Gardez cela public

C'est un forum public pour le bien du projet Elgg. Gardez les sujets publics. Il n'y a aucune raison pour qui que ce soit de vous envoyer des messages privés ou des emails. De même, il n'y a aucune raison de demander à qui que ce soit de vous envoyer un email privé. Publiez en public.

Recommandations

En plus des mentions légales et de la charte du site, suivre des recommandations aident à garder notre site communautaire utile et sûr pour tout le monde.

Contenu

Tous les contenus doivent être tous publics : « PG » (Parental Guidance - classification des contenus adultes) aux USA et au Royaume-Uni. Si votre site Elgg a du contenu adulte et qu'on vous a demandé de publier un lien, veuillez le marquer « NSFW » (Not Safe For Work) de sorte que les personnes le sachent.

L'utilisation excessive de jurons dans n'importe quelle langue ne sera pas tolérée.

Humeur

Travailler avec des problèmes techniques peut être frustrant. Veuillez laisser le site de la communauté libre de frustration. Si vous ressentez de l'anxiété, passez le cap et faites autre chose. Menacer ou attaquer des membres de la communauté, des développeurs du noyau, ou des développeurs de plugins n'aidera pas à résoudre votre problème et risque très probablement de vous faire bannir.

Publicité

Les annonces publicitaires ne sont pas autorisées. La publication de tout type d'annonce sera modérée.

Demander de l'argent / Offrir de payer

Ne demandez pas d'argent sur le site de la communauté. De même , ne proposez pas de payer pour des réponses. Si vous recherchez des développements sur mesure, publiez dans le groupe « Professional Services » (Services Professionnels). Les publications qui demandent de l'argent ou recommandent un plugin commercial peuvent être modérées.

Liens

Si vous avez un problème avec un site en ligne, merci de fournir un lien vers ce site.

Ceci dit, la communauté n'est pas un service de backlinks ni un outil SEO. Un excès de liens sera modéré et votre compte peut être banni.

Signatures

Il y a une raison pour laquelle Elgg n'a pas d'option pour les signatures : elles causent du désordre et détournent l'attention de la conversation. Il est recommandé aux utilisateurs de ne pas utiliser de signature sur le site communautaire, et les signatures avec des liens ou de la publicité seront retirées.

Relancer, +1, moi aussi

Ne faites pas cela. Si votre question n'a pas reçu de réponse, voyez le début de ce document pour des astuces. Ces types de réponses n'ajoutent rien à la conversation et peuvent être modérées.

Publier du Code

De longs extraits de code rendent la lecture confuse dans le cadre d'un forum. Veuillez utiliser http://elgg.pastebin.com pour publier de longs extraits de code et donnez un lien Paste Bin au lieu de publier directement le code/

Bonnes idées

Pas de règles strictes, mais des bonnes idées.

Dites merci

Est-ce que quelqu'un vous a aidé ? Assurez-vous de remercier cette personne : le site de la communauté est animé par des volontaires. Personne n'a la devoir de vous aider avec votre problème. Assurez-vous de montrer votre reconnaissance !

Donnez en retour

Vous avez une astuce pour Elgg ? Vous voyez quelqu'un qui a le même problème que vous aviez eu ? Vous êtes passé par là et pouvez les aider, alors donnez-leur un coup de main !

3.2.9 Sécurité

A partir de Elgg 3.0 plusieurs paramètres de sécurisation ont été ajoutés à Elgg. Vous pouvez activer/désactiver ces paramètres comme vous le souhaitez.

Contents

Protection des mises à niveau

L'URL de http://your-elgg-site.com/upgrade.php peut être protégée par un jeton unique. Ceci évite que des utilisateurs aléatoires aient la possibilité d'exécuter ce fichier. Le jeton n'est pas nécessaire pour les administrateurs du site connectés.

Protection du Cron

Les URLs du *cron* peuvent être protégées par un jeton unique. Ceci évite que des utilisateurs aléatoires puissent exécuter le cron. Le jeton n'est pas nécessaire que le cron est lancé depuis la ligne de commande du serveur.

Désactiver l'autocomplétion du mot de passe

Les données saisies dans ces champs seront mises en cache par le navigateur. Un attaquant qui a accès au navigateur de la victime pourrait subtiliser ces informations. Ceci est particulièrement important si l'application est utilisée habituellement depuis des ordinateurs partagés tels que des cybercafés ou des terminaux d'aéroports. Si vous désactivez ceci, les outils de gestion des mots de passe ne pourront plus renseigner automatiquement ces champs. Le support pour l'attribut « autocomplete » peut dépendre des navigateurs.

Le changement d'adresse email requiert le mot de passe

Quand un utilisateur souhaite changer l'adresse email associée à son compte, il doit également donner son mot de passe actuel.

Session bound icons

Entity icons can be session bound by default. This means the URLs generated also contain information about the current session. Having icons session bound makes icon urls not shareable between sessions. The side effect is that caching of these urls will only help the active session.

Notification aux administrateurs du site

Quand un nouvel administrateur de site est ajouté ou quand un administrateur du site est retiré tous les administrateurs du site reçoivent une notification à ce sujet.

Notifications à l'utilisateur

Administrateur du site

Quand le rôle d'administrateur du site est ajouté ou retiré à un compte, envoie une notification à l'utilisateur dont le compte est affecté.

Bannir/Réintégrer

Quand le compte d'un utilisateur est banni ou réintégré, permet à l'utilisateur affecté d'être informé de cette action.

3.2.10 Validation des utilisateurs

Les plugins peuvent influencer la manières dont les utilisateurs sont validés avant de pouvoir utiliser le site.

Contents

Liste des utilisateurs non validés

Dans la section Admin du site vous trouverez une liste des utilisateurs non validés. Plusieurs actions peuvent être effectuées sur les utilisateurs, telles que les supprimer du système ou les valider.

Les plugins ont la possibilité d'ajouter de nouvelles fonctionnalités à cette liste.

Voir aussi :

Un exemple de ceci est le plugin *Validation des comptes utilisateur via l'email* qui ne permet pas aux utilisateurs de se connecter sur le site tant que leur adresse email n'est pas validée.

3.3 Developer Guides

Customize Elgg's behavior with plugins.

3.3.1 Don't Modify Core

Avertissement : In general, you shouldn't modify non-config files that come with third-party software like Elgg.

The best way to customize the behavior of Elgg is to *install Elgg as a composer dependency* and use the root directory to store modifications specific to your application, and alter behavior through the rich Elgg plugin API.

If you'd like to share customizations between sites or even publish your changes as a reusable package for the community, create a *plugin* using the same plugin APIs and file structure.

It makes it hard to get help

When you don't share the same codebase as everyone else, it's impossible for others to know what is going on in your system and whether your changes are to blame. This can frustrate those who offer help because it can add considerable noise to the support process.

It makes upgrading tricky and potentially disastrous

You will certainly want or need to upgrade Elgg to take advantage of
- security patches
- new features
- new plugin APIs
- new stability improvements
- performance improvements

If you've modified core files, then you must be very careful when upgrading that your changes are not overwritten and that they are compatible with the new Elgg code. If your changes are lost or incompatible, then the upgrade may remove features you've added and even completely break your site.

This can also be a slippery slope. Lots of modifications can lead you to an upgrade process so complex that it's practically impossible. There are lots of sites stuck running old versions software due to taking this path.

It may break plugins

You may not realize until much later that your « quick fix » broke seemingly unrelated functionality that plugins depended on.

Summary

- **Resist the temptation** Editing existing files is quick and easy, but doing so heavily risks the maintainability, security, and stability of your site.
- When receiving advice, consider if the person telling you to modify core will be around to rescue you if you run into trouble later !
- **Apply these principle to software in general.** If you can avoid it, don't modify third party plugins either, for the same reasons : Plugin authors release new versions, too, and you will want those updates.

3.3.2 Access Control Lists

An Access Control List (or ACL) can grant one or more users access to an entity or annotation in the database.

Contents

Voir aussi :

Database Access Control

Creating an ACL

An access collection can be create by using the function *create_access_collection()*.

```
$owner_guid = elgg_get_logged_in_user_guid();

$acl = create_access_collection("Sample name", $owner_guid, 'collection_subtype');
```

ACL subtypes

ACLs can have a subtype, this is to help differentiate between the usage of the ACL. It's higly recommended to set a subtype for an ACL.

Elgg core has three examples of subtype usage

- `group_acl` an ACL owned by an `ElggGroup` which grants group members access to content shared with the group
- `friends` an ACL owned by an `ElggUser` which grant friends of a user access to content shared with friends
- `friends_collection` an ACL owned by an `ElggUser` which grant specific friends access to content shared with the ACL

Adding users to an ACL

If you have an ACL you still need to add users to it in order to grant those users access to content with the *access_id* of the ACLs *id*.

```
// creating an ACL
$owner_guid = elgg_get_logged_in_user_guid();

$acl_id = create_access_collection("Sample name", $owner_guid, 'collection_subtype');

// add other user (procedural style)
add_user_to_access_collection($some_user_guid, $acl_id);

// add other user (object oriented style)
/* @var $acl ElggAccessCollection */
```

(suite sur la page suivante)

(suite de la page précédente)

```
$acl = get_access_collection($acl_id);

$acl->addMember($some_other_user_guid);
```

Removing users from an ACL

If you no longer wish to allow access for a given user in an ACL you can easily remove that user from the list.

```
// remove a user from an ACL (procedural style)
remove_user_from_access_collection($user_guid_to_be_removed, $acl_id);

// remove a user from an ACL (object oriented style)
/* @var $acl ElggAccessCollection */
$acl = get_access_collection($acl_id);

$acl->removeMember(user_guid_to_be_removed);
```

Retrieving an ACL

In order to manage an ACL, or add the ID of an ACL to an access list there are several functions available to retrieve an ACL from the database.

```
// get ACL based on known id
$acl = get_access_collection($acl_id);

// get all ACLs of an owner (procedural style)
$acls = elgg_get_access_collections([
        'owner_guid' => $some_owner_guid,
]);

// get all ACLs of an owner (object oriented style)
$acls = $some_owner_entity->getOwnedAccessCollections();

// add a filter for ACL subtype
// get all ACLs of an owner (procedural style)
$acls = elgg_get_access_collections([
        'owner_guid' => $some_owner_guid,
        'subtype' => 'some_subtype',
]);

// get all ACLs of an owner (object oriented style)
$acls = $some_owner_entity->getOwnedAccessCollections([
        'subtype' => 'some_subtype',
]);

// get one ACL of an owner (object oriented style)
// for example the group_acl of an ElggGroup
// Returns the first ACL owned by the entity with a given subtype
$acl = $group_entity->getOwnedAccessCollection('group_acl');
```

Read access

The access system of Elgg automaticly adds all the ACLs a user is a member of to the access checks. For example a user is a member of a group and is friends with 3 other users, all the corresponding ACLs are added in order to check access to entities when retrieving them (eg. listing all blogs).

3.3.3 Accessibility

This page aims to list and document accessibility rules and best practices, to help core and plugins developpers to make Elgg the most accessible social engine framework that everyone dreams of.

Note : This is an ongoing work, please contribute on Github if you have some skills in this field !

Resources + references

- Official WCAG Accessibility Guidelines Overview
- Official WCAG Accessibility Guidelines
- Resources for planning and implementing for accessibility
- Practical tips from the W3C for improving accessibility
- Preliminary review of websites for accessibility
- Tools for checking the accessibility of websites
- List of practical techniques for implementing accessibility (It would be great if someone could go through this and filter out all the ones that are relevant to Elgg)

Tips for implementing accessibility

- All accessibility-related tickets reported to trac should be tagged with « a11y », short for « accessibility »
- Use core views such as `output/*`, and `input/*` to generate markup, since we can bake a11y concerns into these views
- All images should have a descriptive `alt` attribute. Spacer or purely decorative graphics should have blank `alt` attributes
- All `<a>` tags should have text or an accessible image inside. Otherwise screen readers will have to read the URL, which is a poor experience `<a>` tags should contain descriptive text, if possible, as opposed to generic text like « Click here »
- Markup should be valid
- Themes should not reset « outline » to nothing. `:focus` deserves a special visual treatment so that handicapped users can know where they are

Tips for testing accessibility

- Use the tools linked to from the resources section. Example report for community.elgg.org on June 16, 2012
- Try different font-size/zoom settings in your browser and make sure the theme remains usable
- Turn off css to make sure the sequential order of the page makes sense

Documentation objectives and principles

- Main accessibility rules
- collect and document best practices
- Provide code examples
- Keep the document simple and usable
- Make it usable for both beginner developpers and experts (from most common and easiest changes to elaborate techniques)

3.3.4 Forms + Actions

Create, update, or delete content.

Elgg forms submit to actions. Actions define the behavior for form submission.

This guide assumes basic familiarity with :

- *Plugins*
- *Views*
- *Internationalization*

Contents

Registering actions

Actions must be registered before use.

There are two ways to register actions :

Using `elgg_register_action`

```
elgg_register_action("example", __DIR__ . "/actions/example.php");
```

The `mod/example/actions/example.php` script will now be run whenever a form is submitted to `http://localhost/elgg/action/example`.

Use `elgg-plugin.php`

```
return [
   'actions' => [
      // defaults to using an action file in /actions/myplugin/action_a.php
      'myplugin/action_a' => [
         'access' => 'public',
      ],

      // define custom action path
      'myplugin/action_b' => [
         'access' => 'admin',
         'filename' => __DIR__ . '/actions/action.php'
      ],

      // define a controller
      'myplugin/action_c' => [
         'controller' => \MyPlugin\Actions\ActionC::class,
      ],
   ],
];
```

Avertissement : A stumbling point for many new developers is the URL for actions. The URL always uses `/action/` (singular) and never `/actions/` (plural). However, action script files are usually saved under the directory `/actions/` (plural) and always have an extension. Use `elgg_generate_action_url()` to avoid confusion.

Registering actions using plugin config file

You can also register actions via the *elgg-plugin* config file. To do this you need to provide an action section in the config file. The location of the action files are assumed to be in the plugin folder `/actions`.

```
<?php

return [
        'actions' => [
            'blog/save' => [], // all defaults
            'blog/delete' => [ // all custom
                  'access' => 'admin',
                  'filename' => __DIR__ . 'actions/blog/remove.php',
            ],
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ],
];
```

Permissions

By default, actions are only available to logged in users.

To make an action available to logged out users, pass `"public"` as the third parameter :

```
elgg_register_action("example", $filepath, "public");
```

To restrict an action to only administrators, pass `"admin"` for the last parameter :

```
elgg_register_action("example", $filepath, "admin");
```

Writing action files

Use the `get_input` function to get access to request parameters :

```
$field = get_input('input_field_name', 'default_value');
```

You can then use the *Database* api to load entities and perform actions on them accordingly.

To indicate a successful action, use `elgg_ok_response()`. This function accepts data that you want to make available to the client for XHR calls (this data will be ignored for non-XHR calls)

```
$user = get_entity($guid);
// do something

$action_data = [
   'entity' => $user,
   'stats' => [
       'friends' => $user->getFriends(['count' => true]);
   ],
];

return elgg_ok_response($action_data, 'Action was successful', 'url/to/forward/to');
```

To indicate an error, use `elgg_error_response()`

```
$user = elgg_get_logged_in_user_entity();
if (!$user) {
   // show an error and forward the user to the referring page
   // send 404 error code on AJAX calls
   return elgg_error_response('User not found', REFERRER, ELGG_HTTP_NOT_FOUND);
}

if (!$user->canEdit()) {
   // show an error and forward to user's profile
   // send 403 error code on AJAX calls
   return elgg_error_response('You are not allowed to perform this action', $user->
↪getURL(), ELGG_HTTP_FORBIDDEN);
}
```

Customizing actions

Before executing any action, Elgg triggers a hook :

```
$result = elgg_trigger_plugin_hook('action', $action, null, true);
```

Where `$action` is the action being called. If the hook returns `false` then the action will not be executed.

Example : Captcha

The captcha module uses this to intercept the `register` and `user/requestnewpassword` actions and redirect them to a function which checks the captcha code. This check returns `true` if valid or `false` if not (which prevents the associated action from executing).

This is done as follows :

```
elgg_register_plugin_hook_handler("action", "register", "captcha_verify_action_hook");
elgg_register_plugin_hook_handler("action", "user/requestnewpassword", "captcha_
↪verify_action_hook");

...

function captcha_verify_action_hook($hook, $entity_type, $returnvalue, $params) {
  $token = get_input('captcha_token');
  $input = get_input('captcha_input');

  if (($token) && (captcha_verify_captcha($input, $token))) {
    return true;
  }

  register_error(elgg_echo('captcha:captchafail'));

  return false;
}
```

This lets a plugin extend an existing action without the need to replace the whole action. In the case of the captcha plugin it allows the plugin to provide captcha support in a very loosely coupled way.

Actions available in core

entity/delete

If your plugin does not implement any custom logic when deleting an entity, you can use bundled delete action

```
$guid = 123;
// You can provide optional forward path as a URL query parameter
$forward_url = 'path/to/forward/to';
echo elgg_view('output/url', array(
   'text' => elgg_echo('delete'),
   'href' => "action/entity/delete?guid=$guid&forward_url=$forward_url",
   'confirm' => true,
));
```

You can customize the success message keys for your entity type and subtype, using `"entity:delete:$type:$subtype:success"` and `"entity:delete:$type:success"` keys.

```
// to add a custom message when a blog post or file is deleted
// add the translations keys in your language files
return array(
   'entity:delete:object:blog:success' => 'Blog post has been deleted,
   'entity:delete:object:file:success' => 'File titled %s has been deleted',
);
```

Forms

To output a form, use the elgg_view_form function like so :

```
echo elgg_view_form('example');
```

Doing this generates something like the following markup :

```
<form action="http://localhost/elgg/action/example">
  <fieldset>
    <input type="hidden" name="__elgg_ts" value="1234567890" />
    <input type="hidden" name="__elgg_token" value="3874acfc283d90e34" />
  </fieldset>
</form>
```

Elgg does some things automatically for you when you generate forms this way :

1. It sets the action to the appropriate URL based on the name of the action you pass to it
2. It adds some anti-csrf tokens (`__elgg_ts` and `__elgg_token`) to help keep your actions secure
3. It automatically looks for the body of the form in the `forms/example` view.

Put the content of your form in your plugin's `forms/example` view :

```
// /mod/example/views/default/forms/example.php
echo elgg_view('input/text', array('name' => 'example'));

// defer form footer rendering
// this will allow other plugins to extend forms/example view
elgg_set_form_footer(elgg_view('input/submit'));
```

Now when you call `elgg_view_form('example')`, Elgg will produce :

```
<form action="http://localhost/elgg/action/example">
  <fieldset>
    <input type="hidden" name="__elgg_ts" value="...">
    <input type="hidden" name="__elgg_token" value="...">

    <input type="text" class="elgg-input-text" name="example">
    <div class="elgg-foot elgg-form-footer">
        <input type="submit" class="elgg-button elgg-button-submit" value="Submit">
    </div>
  </fieldset>
</form>
```

Inputs

To render a form input, use one of the bundled input views, which cover all standard HTML input elements. See individual view files for a list of accepted parameters.

```
echo elgg_view('input/select', array(
   'required' => true,
   'name' => 'status',
   'options_values' => array(
      'draft' => elgg_echo('status:draft'),
      'published' => elgg_echo('status:published'),
   ),
   // most input views will render additional parameters passed to the view
   // as tag attributes
   'data-rel' => 'blog',
));
```

The above example will render a dropdown select input :

```
<select required="required" name="status" data-rel="blog" class="elgg-input-select">
   <option value="draft">Draft</option>
   <option value="published">Published</option>
</select>
```

To ensure consistency in field markup, use `elgg_view_field()`, which accepts all the parameters of the input being rendered, as well as `#label` and `#help` parameters (both of which are optional and accept HTML or text).

```
echo elgg_view_field(array(
   '#type' => 'select',
   '#label' => elgg_echo('blog:status:label'),
   '#help' => elgg_view_icon('help') . elgg_echo('blog:status:help'),
   'required' => true,
   'name' => 'status',
   'options_values' => array(
      'draft' => elgg_echo('status:draft'),
      'published' => elgg_echo('status:published'),
   ),
   'data-rel' => 'blog',
));
```

The above will generate the following markup :

```
<div class="elgg-field elgg-field-required">
   <label for="elgg-field-1" class="elgg-field-label">Blog status<span title="Required
↪" class="elgg-required-indicator">*</span></label>
   <div class="elgg-field-input">
      <select required="required" name="status" data-rel="blog" id="elgg-field-1"␣
↪class="elgg-input-select">
         <option value="draft">Draft</option>
         <option value="published">Published</option>
      </select>
   </div>
   <div class="elgg-field-help elgg-text-help">
      <span class="elgg-icon-help elgg-icon"></span>This indicates whether or not the␣
↪blog is visible in the feed
   </div>
</div>
```

Input types

A list of bundled input types/views :

- `input/text` - renders a text input `<input type="text">`
- `input/plaintext` - renders a textarea `<textarea></textarea>`
- `input/longtext` - renders a WYSIWYG text input
- `input/url` - renders a url input `<input type="url">`
- `input/email` - renders an email input `<input type="email">`
- `input/checkbox` - renders a single checkbox `<input type="checkbox">`
- `input/checkboxes` - renders a set of checkboxes with the same name
- `input/radio` - renders one or more radio buttons `<input type="radio">`
- `input/submit` - renders a submit button `<input type="submit">`
- `input/button` - renders a button `<button></button>`
- `input/file` - renders a file input `<input type="file">`
- `input/select` - renders a select input `<select></select>`
- `input/hidden` - renders a hidden input `<input type="hidden">`
- `input/password` - renders a password input `<input type="password">`
- `input/number` - renders a number input `<input type="number">`
- `input/date` - renders a jQuery datepicker
- `input/access` - renders an Elgg access level select
- `input/tags` - renders an Elgg tags input
- `input/autocomplete` - renders an Elgg entity autocomplete
- `input/captcha` - placeholder view for plugins to extend
- `input/friendspicker` - renders an Elgg friend autocomplete
- `input/userpicker` - renders an Elgg user autocomplete
- `input/location` renders an Elgg location input

Files and images

Use the input/file view in your form's content view.

```
// /mod/example/views/default/forms/example.php
echo elgg_view('input/file', array('name' => 'icon'));
```

If you wish to upload an icon for entity you can use the helper view `entity/edit/icon`. This view shows a file input for uploading a new icon for the entity, an thumbnail of the current icon and the option to remove the current icon.

The view supports some variables to control the output

- `entity` - the entity to add/remove the icon for. If provided based on this entity the thumbnail and remove option wil be shown
- `entity_type` - the entity type for which the icon will be uploaded. Plugins could find this usefull, maybe to validate icon sizes
- `entity_subtype` - the entity subtype for which the icon will be uploaded. Plugins could find this usefull, maybe to validate icon sizes
- `icon_type` - the type of the icon (default : icon)
- `name` - name of the input/file (default : icon)
- `remove_name` - name of the remove icon toggle (default : $vars["name"] . "_remove")
- `required` - is icon upload required (default : false)
- `show_remove` - show the remove icon option (default : true)
- `show_thumb` - show the thumb of the entity if available (default : true)
- `thumb_size` - the icon size to use as the thumb (default : medium)

If using the helper view you can use the following code in you action to save the icon to the entity or remove the current icon.

```
if (get_input('icon_remove')) {
   $entity->deleteIcon();
} else {
   $entity->saveIconFromUploadedFile('icon');
}
```

Set the enctype of the form to multipart/form-data :

```
echo elgg_view_form('example', array(
  'enctype' => 'multipart/form-data'
));
```

Note : The `enctype` of all forms that use the method `POST` defaults to `multipart/form-data`.

In your action file, use `elgg_get_uploaded_file('your-input-name')` to access the uploaded file :

```
$icon = elgg_get_uploaded_file('icon');
```

Sticky forms

Sticky forms are forms that retain user input if saving fails. They are « sticky » because the user's data « sticks » in the form after submitting, though it was never saved to the database. This greatly improves the user experience by minimizing data loss. Elgg includes helper functions so you can make any form sticky.

Helper functions

Sticky forms are implemented in Elgg by the following functions :

`elgg_make_sticky_form($name)` Tells the engine to make all input on a form sticky.

`elgg_clear_sticky_form($name)` Tells the engine to discard all sticky input on a form.

`elgg_is_sticky_form($name)` Checks if $name is a valid sticky form.

`elgg_get_sticky_values($name)` Returns all sticky values saved for $name by elgg_make_sticky_form().

Overview

The basic flow of using sticky forms is : Call `elgg_make_sticky_form($name)` at the top of actions for forms you want to be sticky. Use `elgg_is_sticky_form($name)` and `elgg_get_sticky_values($name)` to get sticky values when rendering a form view. Call `elgg_clear_sticky_form($name)` after the action has completed successfully or after data has been loaded by `elgg_get_sticky_values($name)`.

Example : User registration

Simple sticky forms require little logic to determine the input values for the form. This logic is placed at the top of the form body view itself.

The registration form view first sets default values for inputs, then checks if there are sticky values. If so, it loads the sticky values before clearing the sticky form :

```
// views/default/forms/register.php
$password = $password2 = '';
$username = get_input('u');
$email = get_input('e');
$name = get_input('n');

if (elgg_is_sticky_form('register')) {
   extract(elgg_get_sticky_values('register'));
   elgg_clear_sticky_form('register');
}
```

The registration action sets creates the sticky form and clears it once the action is completed :

```
// actions/register.php
elgg_make_sticky_form('register');

...

$guid = register_user($username, $password, $name, $email, false, $friend_guid,
↪$invitecode);

if ($guid) {
   elgg_clear_sticky_form('register');
   ....
}
```

Example : Bookmarks

The bundled plugin Bookmarks" save form and action is an example of a complex sticky form.

The form view for the save bookmark action uses `elgg_extract()` to pull values from the `$vars` array :

```
// mod/bookmarks/views/default/forms/bookmarks/save.php
$title = elgg_extract('title', $vars, '');
$desc = elgg_extract('description', $vars, '');
$address = elgg_extract('address', $vars, '');
$tags = elgg_extract('tags', $vars, '');
$access_id = elgg_extract('access_id', $vars, ACCESS_DEFAULT);
$container_guid = elgg_extract('container_guid', $vars);
$guid = elgg_extract('guid', $vars, null);
$shares = elgg_extract('shares', $vars, array());
```

The page handler scripts prepares the form variables and calls `elgg_view_form()` passing the correct values :

```
// mod/bookmarks/pages/add.php
$vars = bookmarks_prepare_form_vars();
$content = elgg_view_form('bookmarks/save', array(), $vars);
```

Similarly, `mod/bookmarks/pages/edit.php` uses the same function, but passes the entity that is being edited as an argument :

```
$bookmark_guid = get_input('guid');
$bookmark = get_entity($bookmark_guid);

...

$vars = bookmarks_prepare_form_vars($bookmark);
$content = elgg_view_form('bookmarks/save', array(), $vars);
```

The library file defines `bookmarks_prepare_form_vars()`. This function accepts an `ElggEntity` as an argument and does 3 things :

1. Defines the input names and default values for form inputs.
2. Extracts the values from a bookmark object if it's passed.
3. Extracts the values from a sticky form if it exists.

TODO : Include directly from lib/bookmarks.php

```
// mod/bookmarks/lib/bookmarks.php
function bookmarks_prepare_form_vars($bookmark = null) {
   // input names => defaults
   $values = array(
      'title' => get_input('title', ''), // bookmarklet support
      'address' => get_input('address', ''),
      'description' => '',
      'access_id' => ACCESS_DEFAULT,
      'tags' => '',
      'shares' => array(),
      'container_guid' => elgg_get_page_owner_guid(),
      'guid' => null,
      'entity' => $bookmark,
   );

   if ($bookmark) {
      foreach (array_keys($values) as $field) {
         if (isset($bookmark->$field)) {
            $values[$field] = $bookmark->$field;
         }
      }
   }

   if (elgg_is_sticky_form('bookmarks')) {
      $sticky_values = elgg_get_sticky_values('bookmarks');
      foreach ($sticky_values as $key => $value) {
         $values[$key] = $value;
      }
   }

   elgg_clear_sticky_form('bookmarks');

   return $values;
}
```

The save action checks the input, then clears the sticky form upon success :

```
// mod/bookmarks/actions/bookmarks/save.php
elgg_make_sticky_form('bookmarks');
...

if ($bookmark->save()) {
   elgg_clear_sticky_form('bookmarks');
}
```

Ajax

See the *Ajax guide* for instructions on calling actions from JavaScript.

Security

For enhanced security, all actions require an CSRF token. Calls to action URLs that do not include security tokens will be ignored and a warning will be generated.

A few views and functions automatically generate security tokens :

```
elgg_view('output/url', array('is_action' => TRUE));
elgg_view('input/securitytoken');
$url = elgg_add_action_tokens_to_url("http://localhost/elgg/action/example");
```

In rare cases, you may need to generate tokens manually :

```
$__elgg_ts = time();
$__elgg_token = generate_action_token($__elgg_ts);
```

You can also access the tokens from javascript :

```
elgg.security.token.__elgg_ts;
elgg.security.token.__elgg_token;
```

These are refreshed periodically so should always be up-to-date.

Security Tokens

On occasion we need to pass data through an untrusted party or generate an « unguessable token » based on some data. The industry-standard HMAC algorithm is the right tool for this. It allows us to verify that received data were generated by our site, and were not tampered with. Note that even strong hash functions like SHA-2 should *not* be used without HMAC for these tasks.

Elgg provides `elgg_build_hmac()` to generate and validate HMAC message authentication codes that are unguessable without the site's private key.

```
// generate a querystring such that $a and $b can't be altered
$a = 1234;
$b = "hello";
$query = http_build_query([
    'a' => $a,
    'b' => $b,
    'mac' => elgg_build_hmac([$a, $b])->getToken(),
]);
$url = "action/foo?$query";
```

(suite sur la page suivante)

(suite de la page précédente)

```
// validate the querystring
$a = (int) get_input('a', '', false);
$b = (string) get_input('b', '', false);
$mac = get_input('mac', '', false);

if (elgg_build_hmac([$a, $b])->matchesToken($mac)) {
    // $a and $b have not been altered
}
```

Note : If you use a non-string as HMAC data, you must use types consistently. Consider the following :

```
$mac = elgg_build_hmac([123, 456])->getToken();

// type of first array element differs
elgg_build_hmac(["123", 456])->matchesToken($mac); // false

// types identical to original
elgg_build_hmac([123, 456])->matchesToken($mac); // true
```

Signed URLs

Signed URLs offer a limited level of security for situations where action tokens are not suitable, for example when sending a confirmation link via email. URL signatures verify that the URL has been generated by your Elgg installation (using site secret) and that the URL query elements were not tampered with.

URLs a signed with an unguessable SHA-256 HMAC key. See *Security Tokens* for more details.

```
$url = elgg_http_add_url_query_element(elgg_normalize_url('confirm'), [
   'user_guid' => $user_guid,
]);

$url = elgg_http_get_signed_url($url);

notify_user($user_guid, $site->guid, 'Confirm', "Please confirm by clicking this
↪link: $url");
```

Avertissement : Signed URLs do not offer CSRF protection and should not be used instead of action tokens.

3.3.5 Ajax

The `elgg/Ajax` AMD module (introduced in Elgg 2.1) provides a set of methods for communicating with the server in a concise and uniform way, which allows plugins to collaborate on the request data, the server response, and the returned client-side data.

Client and server code written for the legacy API should not need modification.

Contents

Overview

All the ajax methods perform the following :

1. Client-side, the `data` option (if given as an object) is filtered by the hook `ajax_request_data`.
2. The request is made to the server, either rendering a view or a form, calling an action, or loading a path.
3. The method returns a `jqXHR` object, which can be used as a Promise.
4. Server-echoed content is turned into a response object (`Elgg\Services\AjaxResponse`) containing a string (or a JSON-parsed value).
5. The response object is filtered by the hook `ajax_response`.
6. The response object is used to create the HTTP response.
7. Client-side, the response data is filtered by the hook `ajax_response_data`.
8. The `jqXHR` promise is resolved and any `success` callbacks are called.

More notes :

— All hooks have a type depending on the method and first argument. See below.
— By default the `elgg/spinner` module is automatically used during requests.
— User messages generated by `system_message()` and `register_error()` are collected and displayed on the client.
— Elgg gives you a default error handler that shows a generic message if output fails.
— PHP exceptions or denied resource return HTTP error codes, resulting in use of the client-side error handler.
— The default HTTP method is `POST` for actions, otherwise `GET`. You can set it via `options.method`.
— If a non-empty `options.data` is given, the default method is always `POST`.
— For client caching, set `options.method` to `"GET"` and `options.data.elgg_response_ttl` to the max-age you want in seconds.
— To save system messages for the next page load, set `options.data.elgg_fetch_messages = 0`. You may want to do this if you intent to redirect the user based on the response.
— To stop client-side API from requiring AMD modules required server-side with `elgg_require_js()`, set `options.data.elgg_fetch_deps = 0`.
— All methods accept a query string in the first argument. This is passed on to the fetch URL, but does not appear in the hook types.

Performing actions

Consider this action :

```
// in myplugin/actions/do_math.php

elgg_ajax_gatekeeper();

$arg1 = (int)get_input('arg1');
$arg2 = (int)get_input('arg2');

// will be rendered client-side
system_message('We did it!');

echo json_encode([
    'sum' => $arg1 + $arg2,
    'product' => $arg1 * $arg2,
]);
```

To execute it, use `ajax.action('<action_name>', options)` :

```
var Ajax = require('elgg/Ajax');
var ajax = new Ajax();

ajax.action('do_math', {
    data: {
        arg1: 1,
        arg2: 2
    },
}).done(function (output, statusText, jqXHR) {
    if (jqXHR.AjaxData.status == -1) {
        return;
    }

    alert(output.sum);
    alert(output.product);
});
```

Notes for actions :

- **All hooks have type `action:<action_name>`. So in this case, three hooks will be triggered :**
  - client-side `"ajax_request_data", "action:do_math"` to filter the request data (before it's sent)
  - server-side `"ajax_response", "action:do_math"` to filter the response (after the action runs)
  - client-side `"ajax_response_data", "action:do_math"` to filter the response data (before the calling code receives it)
- CSRF tokens are added to the request data.
- The default method is `POST`.
- An absolute action URL can be given in place of the action name.
- Using `forward()` in an action simply sends the response. The URL given in not returned to the client.

Note : When setting `data`, use `ajax.objectify($form)` instead of `$form.serialize()`. Doing so allows the `ajax_request_data` plugin hook to fire and other plugins to alter/piggyback on the request.

Fetching data

Consider this PHP script that runs at `http://example.org/myplugin_time`.

```
// in myplugin/elgg-plugin.php
return [
    'routes' => [
        'default:myplugin:time' => [
            'path' => '/myplugin_time',
            'resource' => 'myplugin/time',
        ],
    ],
];

// in myplugin/views/default/resources/myplugin/time.php
elgg_ajax_gatekeeper();

echo json_encode([
    'rfc2822' => date(DATE_RFC2822),
    'day' => date('l'),
]);

return true;
```

To fetch its output, use `ajax.path('<url_path>', options)`.

```
var Ajax = require('elgg/Ajax');
var ajax = new Ajax();

ajax.path('myplugin_time').done(function (output, statusText, jqXHR) {
    if (jqXHR.AjaxData.status == -1) {
        return;
    }

    alert(output.rfc2822);
    alert(output.day);
});
```

Notes for paths :

- The 3 hooks (see Actions above) will have type `path:<url_path>`. In this case, « path :myplugin_time ».
- If the page handler echoes a regular web page, `output` will be a string containing the HTML.
- An absolute URL can be given in place of the path name.

Fetching views

Consider this view :

```
// in myplugin/views/default/myplugin/get_link.php

if (empty($vars['entity']) || !$vars['entity'] instanceof ElggObject) {
    return;
}

$object = $vars['entity'];
/* @var ElggObject $object */
```

(suite sur la page suivante)

(suite de la page précédente)

```
echo elgg_view('output/url', [
    'text' => $object->getDisplayName(),
    'href' => $object->getUrl(),
    'is_trusted' => true,
]);
```

Since it's a PHP file, we must register it for Ajax first :

```
// in myplugin_init()
elgg_register_ajax_view('myplugin/get_link');
```

To fetch the view, use `ajax.view('<view_name>', options)` :

```
var Ajax = require('elgg/Ajax');
var ajax = new Ajax();

ajax.view('myplugin/get_link', {
    data: {
        guid: 123 // querystring
    },
}).done(function (output, statusText, jqXHR) {
    if (jqXHR.AjaxData.status == -1) {
        return;
    }

    $('.myplugin-link').html(output);
});
```

Notes for views :

- The 3 hooks (see Actions above) will have type `view:<view_name>`. In this case, « view :myplugin/get_link ».
- `output` will be a string with the rendered view.
- The request data are injected into `$vars` in the view.
- If the request data contains `guid`, the system sets `$vars['entity']` to the corresponding entity or `false` if it can't be loaded.

Avertissement : In ajax views and forms, note that `$vars` can be populated by client input. The data is filtered like `get_input()`, but may not be the type you're expecting or may have unexpected keys.

Fetching forms

Consider we have a form view. We register it for Ajax :

```
// in myplugin_init()
elgg_register_ajax_view('forms/myplugin/add');
```

To fetch this using `ajax.form('<action_name>', options)`.

```
var Ajax = require('elgg/Ajax');
var ajax = new Ajax();

ajax.form('myplugin/add').done(function (output, statusText, jqXHR) {
    if (jqXHR.AjaxData.status == -1) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return;
    }

    $('.myplugin-form-container').html(output);
});
```

Notes for forms :

- The 3 hooks (see Actions above) will have type `form:<action_name>`. In this case, « form :myplugin/add ».
- `output` will be a string with the rendered view.
- The request data are injected into `$vars` in your form view.
- If the request data contains `guid`, the system sets `$vars['entity']` to the corresponding entity or `false` if it can't be loaded.

Note : Only the request data are passed to the requested form view (i.e. as a third parameter accepted by `elgg_view_form()`). If you need to pass attributes or parameters of the form element rendered by the `input/form` view (i.e. normally passed as a second parameter to `elgg_view_form()`), use the server-side hook `view_vars, input/form`.

Avertissement : In ajax views and forms, note that `$vars` can be populated by client input. The data is filtered like `get_input()`, but may not be the type you're expecting or may have unexpected keys.

Submitting forms

To submit a form using Ajax, simply pass `ajax` parameter with form variables :

```
echo elgg_view_form('login', ['ajax' => true]);
```

Redirects

Use `ajax.forward()` to start a spinner and redirect the user to a new destination.

```
var Ajax = require('elgg/Ajax');
var ajax = new Ajax();
ajax.forward('/activity');
```

Piggybacking on an Ajax request

The client-side `ajax_request_data` hook can be used to append or filter data being sent by an `elgg/Ajax` request.

Let's say when the view `foo` is fetched, we want to also send the server some data :

```
// in your boot module
var Ajax = require('elgg/Ajax');
var elgg = require('elgg');

var ajax = new Ajax();
```

(suite sur la page suivante)

(suite de la page précédente)

```
elgg.register_hook_handler(Ajax.REQUEST_DATA_HOOK, 'view:foo', function (name, type,␣
↪params, data) {
    // send some data back
    data.bar = 1;
    return data;
});
```

This data can be read server-side via `get_input('bar');`.

Note : If data was given as a string (e.g. `$form.serialize()`), the request hooks are not triggered.

Note : The form will be objectified as `FormData`, and the request content type will be determined accordingly. Effectively this allows plugins to submit multipart form data without using `jquery.form` plugin and other `iframe` hacks.

Piggybacking on an Ajax response

The server-side `ajax_response` hook can be used to append or filter response data (or metadata).

Let's say when the view `foo` is fetched, we want to also send the client some additional data :

```
use Elgg\Services\AjaxResponse;

function myplugin_append_ajax($hook, $type, AjaxResponse $response, $params) {

    // alter the value being returned
    $response->getData()->value .= " hello";

    // send some metadata back. Only client-side "ajax_response" hooks can see this!
    $response->getData()->myplugin_alert = 'Listen to me!';

    return $response;
}

// in myplugin_init()
elgg_register_plugin_hook_handler(AjaxResponse::RESPONSE_HOOK, 'view:foo', 'myplugin_
↪append_ajax');
```

To capture the metadata send back to the client, we use the client-side `ajax_response` hook :

```
// in your boot module
var Ajax = require('elgg/Ajax');
var elgg = require('elgg');

elgg.register_hook_handler(Ajax.RESPONSE_DATA_HOOK, 'view:foo', function (name, type,␣
↪params, data) {

    // the return value is data.value

    // the rest is metadata
```

(suite sur la page suivante)

(suite de la page précédente)

```
    alert(data.myplugin_alert);

    return data;
});
```

Note : Only `data.value` is returned to the `success` function or available via the *Deferred* interface.

Note : Elgg uses these same hooks to deliver system messages over `elgg/Ajax` responses.

Handling errors

Responses basically fall into three categories :

1. HTTP success (200) with status `0`. No `register_error()` calls were made on the server.
2. HTTP success (200) with status `-1`. `register_error()` was called.
3. HTTP error (4xx/5xx). E.g. calling an action with stale tokens, or a server exception. In this case the `done` and `success` callbacks are not called.

You may need only worry about the 2nd case. We can do this by looking at `jqXHR.AjaxData.status` :

```
ajax.action('entity/delete?guid=123').done(function (value, statusText, jqXHR) {
    if (jqXHR.AjaxData.status == -1) {
        // a server error was already displayed
        return;
    }

    // remove element from the page
});
```

Requiring AMD modules

Each response from an Ajax service will contain a list of AMD modules required server side with *elgg_require_js()*. When response data is unwrapped, these modules will be loaded asynchronously - plugins should not expect these modules to be loaded in their *$.done()* and *$.then()* handlers and must use *require()* for any modules they depend on. Additionally AMD modules should not expect the DOM to have been altered by an Ajax request when they are loaded - DOM events should be delegated and manipulations on DOM elements should be delayed until all Ajax requests have been resolved.

Legacy elgg.ajax APIs

Elgg 1.8 introduced `elgg.action`, `elgg.get`, `elgg.getJSON`, and other methods which behave less consistently both client-side and server-side.

Legacy elgg.action

Differences :

- you must manually pull the `output` from the returned wrapper
- the `success` handler will fire even if the action is prevented
- the `success` handler will receive a wrapper object. You must look for `wrapper.output`
- no ajax hooks

```
elgg.action('do_math', {
    data: {
        arg1: 1,
        arg2: 2
    },
    success: function (wrapper) {
        if (wrapper.output) {
            alert(wrapper.output.sum);
            alert(wrapper.output.product);
        } else {
            // the system prevented the action from running, but we really don't
            // know why
            elgg.ajax.handleAjaxError();
        }
    }
});
```

elgg.action notes

- It's best to echo a non-empty string, as this is easy to validate in the `success` function. If the action was not allowed to run for some reason, `wrapper.output` will be an empty string.
- You may want to use the *elgg/spinner* module.
- Elgg does not use `wrapper.status` for anything, but a call to `register_error()` causes it to be set to `-1`.
- If the action echoes a non-JSON string, `wrapper.output` will contain that string.
- `elgg.action` is based on `jQuery.ajax` and returns a `jqXHR` object (like a Promise), if you should want to use it.
- After the PHP action completes, other plugins can alter the wrapper via the plugin hook `'output', 'ajax'`, which filters the wrapper as an array (not a JSON string).
- A `forward()` call forces the action to be processed and output immediately, with the `wrapper.forward_url` value set to the normalized location given.
- To make sure Ajax actions can only be executed via XHR, use `elgg_ajax_gatekeeper()`.

elgg.action JSON response wrapper

```
{
    current_url: {String} "http://example.org/action/example/math", // not very useful
    forward_url: {String} "http://example.org/foo", ...if forward('foo') was called
    output: {String|Object} from echo in action
    status: {Number} 0 = success. -1 = an error was registered.
    system_messages: {Object}
}
```

Avertissement : It's probably best to rely only on the `output` key, and validate it in case the PHP action could not run for some reason, e.g. the user was logged out or a CSRF attack did not provide tokens.

Avertissement : If `forward()` is used in response to a legacy ajax request (e.g. `elgg.ajax`), Elgg will *always* respond with this wrapper, **even if not in an action**.

Legacy view fetching

A plugin can use a view script to handle XHR `GET` requests. Here's a simple example of a view that returns a link to an object given by its GUID :

```
// in myplugin_init()
elgg_register_ajax_view('myplugin/get_link');
```

```
// in myplugin/views/default/myplugin/get_link.php

if (empty($vars['entity']) || !$vars['entity'] instanceof ElggObject) {
    return;
}

$object = $vars['entity'];
/* @var ElggObject $object */

echo elgg_view('output/url', [
    'text' => $object->getDisplayName(),
    'href' => $object->getUrl(),
    'is_trusted' => true,
]);
```

```
elgg.get('ajax/view/myplugin/get_link', {
    data: {
        guid: 123 // querystring
    },
    success: function (output) {
        $('.myplugin-link').html(output);
    }
});
```

The Ajax view system works significantly differently than the action system.

- There are no access controls based on session status.
- Non-XHR requests are automatically rejected.
- GET vars are injected into `$vars` in the view.
- If the request contains `$_GET['guid']`, the system sets `$vars['entity']` to the corresponding entity or `false` if it can't be loaded.
- There's no « wrapper » object placed around the view output.
- System messages/errors shouldn't be used, as they don't display until the user loads another page.
- Depending on the view's suffix (.js, .html, .css, etc.), a corresponding Content-Type header is added.

Avertissement :

In ajax views and forms, note that `$vars` can be populated by client input. The data is filtered like `get_input()`, but may not be the type you're expecting or may have unexpected keys.

Returning JSON from a view

If the view outputs encoded JSON, you must use `elgg.getJSON` to fetch it (or use some other method to set jQuery's ajax option `dataType` to `json`). Your `success` function will be passed the decoded Object.

Here's an example of fetching a view that returns a JSON-encoded array of times :

```
elgg.getJSON('ajax/view/myplugin/get_times', {
    success: function (data) {
        alert('The time is ' + data.friendly_time);
    }
});
```

Legacy form fetching

If you register a form view (name starting with `forms/`), you can fetch it pre-rendered with `elgg_view_form()`. Simply use `ajax/form/<action>` (instead of `ajax/view/<view_name>`) :

```
// in myplugin_init()
elgg_register_ajax_view('forms/myplugin/add');
```

```
elgg.get('ajax/form/myplugin/add', {
    success: function (output) {
        $('.myplugin-form-container').html(output);
    }
});
```

Only the request data are passed to the requested form view (i.e. as a third parameter accepted by `elgg_view_form()`). If you need to pass attributes or parameters of the form element rendered by the `input/form` view (i.e. normally passed as a second parameter to `elgg_view_form()`), use the server-side hook `view_vars, input/form`.

Avertissement :

In ajax views and forms, note that `$vars` can be populated by client input. The data is filtered like `get_input()`, but may not be the type you're expecting or may have unexpected keys.

Legacy helper functions

These functions extend jQuery's native Ajax features.

`elgg.get()` is a wrapper for jQuery's `$.ajax()`, but forces `GET` and does URL normalization.

```
// normalizes the url to the current <site_url>/activity
elgg.get('/activity', {
    success: function(resultText, success, xhr) {
        console.log(resultText);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
});
```

`elgg.post()` is a wrapper for jQuery's `$.ajax()`, but forces `POST` and does URL normalization.

3.3.6 Authentication

Elgg provides everything needed to authenticate users via username/email and password out of the box, including :

- remember-me cookies for persistent login
- password reset logic
- secure storage of passwords
- logout
- UIs for accomplishing all of the above

All that's left for you to do as a developer is to use the built-in authentication functions to secure your pages and actions.

Working with the logged in user

Check whether the current user is logged in with `elgg_is_logged_in()` :

```
if (elgg_is_logged_in()) {
  // do something just for logged-in users
}
```

Check if the current user is an admin with `elgg_is_admin_logged_in()` :

```
if (elgg_is_admin_logged_in()) {
  // do something just for admins
}
```

Get the currently logged in user with `elgg_get_logged_in_user_entity()` :

```
$user = elgg_get_logged_in_user_entity();
```

The returned object is an `ElggUser` so you can use all the methods and properties of that class to access information about the user. If the user is not logged in, this will return `null`, so be sure to check for that first.

Gatekeepers

Gatekeeper functions allow you to manage how code gets executed by applying access control rules.

Forward a user to the front page if they are not logged in with `elgg_gatekeeper()` :

```
elgg_gatekeeper();

echo "Information for logged-in users only";
```

Forward a user to the front page unless they are an admin with `elgg_admin_gatekeeper()` :

```
elgg_admin_gatekeeper();

echo "Information for admins only";
```

Pluggable Authentication Modules

Elgg has support for pluggable authentication modules (PAM), which enables you to write your own authentication handlers. Whenever a request needs to get authenticated the system will call `elgg_authenticate()` which probes the registered PAM handlers until one returns success.

The preferred approach is to create a separate Elgg plugin which will have one simple task : to process an authentication request. This involves setting up an authentication handler in the plugin's *start.php* file, and to register it with the PAM module so it will get processed whenever the system needs to authenticate a request.

The authentication handler is a function and takes a single parameter. Registering the handler is being done by `register_pam_handler()` which takes the name of the authentication handler, the importance and the policy as parameters. It is advised to register the handler in the plugin's init function, for example :

```
function your_plugin_init() {
   // Register the authentication handler
   register_pam_handler('your_plugin_auth_handler');
}

function your_plugin_auth_handler($credentials) {
   // do things ...
}

// Add the plugin's init function to the system's init event
elgg_register_elgg_event_handler('init', 'system', 'your_plugin_init');
```

Importance

By default an authentication module is registered with an importance of **sufficient**.

In a list of authentication modules; if any one marked *sufficient* returns `true`, `pam_authenticate()` will also return `true`. The exception to this is when an authentication module is registered with an importance of **required**. All required modules must return `true` for `pam_authenticate()` to return `true`, regardless of whether all sufficient modules return `true`.

Passed credentials

The format of the credentials passed to the handler can vary, depending on the originating request. For example, a regular login via the login form will create a named array, with the keys `username` and `password`. If a request was made for example via XML-RPC then the credentials will be set in the HTTP header, so in this case nothing will get passed to the authentication handler and the handler will need to perform steps on its own to authenticate the request.

Return value

The authentication handle should return a `boolean`, indicating if the request could be authenticated or not. One caveat is that in case of a regular user login where credentials are available as username and password the user will get logged in. In case of the XML-RPC example the authentication handler will need to perform this step itself since the rest of the system will not have any idea of either possible formats of credentials passed nor its contents. Logging in a user is quite simple and is being done by `login()`, which expects an `ElggUser` object.

3.3.7 Context

Avertissement : The contents of this page are outdated. While the functionality is still in place, using global context to determine your business logic is bad practice, and will make your code less testable and succeptive to bugs.

Within the Elgg framework, context can be used by your plugin's functions to determine if they should run or not. You will be registering callbacks to be executed when particular *events are triggered*. Sometimes the events are generic and you only want to run your callback when your plugin caused the event to be triggered. In that case, you can use the page's context.

You can explicitly set the context with `set_context()`. The context is a string and typically you set it to the name of your plugin. You can retrieve the context with the function `get_context()`. It's however better to use `elgg_push_context($string)` to add a context to the stack. You can check if the context you want in in the current stack by calling `elgg_in_context($context)`. Don't forget to pop (with `elgg_pop_context()`) the context after you push one and don't need it anymore.

If you don't set it, Elgg tries to guess the context. If the page was called through the router, the context is set to the first segment of the current route, e.g. `profile` in `profile/username`.

Sometimes a view will return different HTML depending on the context. A plugin can take advantage of that by setting the context before calling `elgg_view()` on the view and then setting the context back. This is frequently done with the search context.

3.3.8 Cron

If you setup cron correctly as described in *Cron* special hooks will be triggered so you can register for these hooks from your own code.

The example below registers a function for the daily cron.

```
function my_plugin_init() {
    elgg_register_plugin_hook_handler('cron', 'daily', 'my_plugin_cron_handler');
}
```

If timing is important in your cron hook be advised that the functions are executed in order of registration. This could mean that your function may start (a lot) later then you may have expected. However the parameters provided in the hook contain the original starting time of the cron, so you can always use that information.

```
function my_plugin_cron_handler($hook, $period, $return, $params) {
    $start_time = elgg_extract('time', $params);
}
```

Voir aussi :

Événements et Hooks des plugins has more information about hooks

3.3.9 Database

Persist user-generated content and settings with Elgg's generic storage API.

Contents

Entities

Creating an object

To create an object in your code, you need to instantiate an `ElggObject`. Setting data is simply a matter of adding instance variables or properties. The built-in properties are :

- ``**guid**`` The entity's GUID ; set automatically
- ``**owner_guid**`` The owning user's GUID
- ``**subtype**`` A single-word arbitrary string that defines what kind of object it is, for example `blog`
- ``**access_id**`` An integer representing the access level of the object
- ``**title**`` The title of the object
- ``**description**`` The description of the object

The object subtype is a special property. This is an arbitrary string that describes what the object is. For example, if you were writing a blog plugin, your subtype string might be *blog*. It's a good idea to make this unique, so that other plugins don't accidentally try and use the same subtype. For the purposes of this document, let's assume we're building a simple forum. Therefore, the subtype will be *forum* :

```
$object = new ElggObject();
$object->subtype = "forum";
$object->access_id = 2;
$object->save();
```

`access_id` is another important property. If you don't set this, your object will be private, and only the creator user will be able to see it. Elgg defines constants for the special values of `access_id` :

- **ACCESS_PRIVATE** Only the owner can see it
- **ACCESS_LOGGED_IN** Any logged in user can see it
- **ACCESS_PUBLIC** Even visitors not logged in can see it

Saving the object will automatically populate the `$object->guid` property if successful. If you change any more base properties, you can call `$object->save()` again, and it will update the database for you.

You can set metadata on an object just like a standard property. Let's say we want to set the SKU of a product :

```
$object->SKU = 62784;
```

If you assign an array, all the values will be set for that metadata. This is how, for example, you set tags.

Metadata cannot be persisted to the database until the entity has been saved, but for convenience, ElggEntity can cache it internally and save it when saving the entity.

Loading an object

By GUID

```
$entity = get_entity($guid);
if (!$entity) {
    // The entity does not exist or you're not allowed to access it.
}
```

But what if you don't know the GUID ? There are several options.

By user, subtype or site

If you know the user ID you want to get objects for, or the subtype, you have several options. The easiest is probably to call the procedural function `elgg_get_entities` :

```
$entities = elgg_get_entities(array(
    'type' => $entity_type,
    'subtype' => $subtype,
    'owner_guid' => $owner_guid,
));
```

This will return an array of `ElggEntity` objects that you can iterate through. `elgg_get_entities` paginates by default, with a limit of 10 ; and offset 0.

You can leave out `owner_guid` to get all objects and leave out subtype or type to get objects of all types/subtypes.

If you already have an `ElggUser` – e.g. `elgg_get_logged_in_user_entity`, which always has the current user's object when you're logged in – you can simply use :

```
$objects = $user->getObjects($subtype, $limit, $offset)
```

But what about getting objects with a particular piece of metadata ?

By properties

You can fetch entities by their properties using `elgg_get_entities`. Using specific parameters passed to `$options` array, you can retrieve entities by their attributes, metadata, annotations, private settings and relationships.

Displaying entities

In order for entities to be displayed in listing functions you need to provide a view for the entity in the views system.

To display an entity, create a view EntityType/subtype where EntityType is one of the following :

object : for entities derived from ElggObject user : for entities derived from ElggUser site : for entities derived from ElggSite group : for entities derived from ElggGroup

A default view for all entities has already been created, this is called EntityType/default.

Entity Icons

Entity icons can be saved from uploaded files, existing local files, or existing ElggFile objects. These methods save the *master* size of the icon defined in the system. The other defined sizes will be generated when requested.

```
$object = new ElggObject();
$object->title = 'Example entity';
$object->description = 'An example object with an icon.';

// from an uploaded file
$object->saveIconFromUploadedFile('file_upload_input');

// from a local file
$object->saveIconFromLocalFile('/var/data/generic_icon.png');

// from a saved ElggFile object
$file = get_entity(123);
if ($file instanceof ElggFile) {
        $object->saveIconFromElggFile($file);
}

$object->save();
```

The following sizes exist by default :

- `master` - 10240px at longer edge (not upscaled)
- `large` - 200px at longer edge (not upscaled)
- `medium` - 100px square
- `small` - 40px square
- `tiny` - 25px square
- `topbar` - 16px square

Use `elgg_get_icon_sizes()` to get all possible icon sizes for a specific entity type and subtype. The function triggers the `entity:icon:sizes` *hook*.

To check if an icon is set, use `$object->hasIcon($size)`.

You can retrieve the URL of the generated icon with `ElggEntity::getIconURL($params)` method. This method accepts a `$params` argument as an array that specifies the size, type, and provide additional context for the hook to determine the icon to serve. The method triggers the `entity:icon:url` *hook*.

Use `elgg_view_entity_icon($entity, $size, $vars)` to render an icon. This will scan the following locations for a view and include the first match to .

1. views/$viewtype/icon/$type/$subtype.php
2. views/$viewtype/icon/$type/default.php
3. views/$viewtype/icon/default.php

Where

$viewtype Type of view, e.g. `'default'` or `'json'`.

$type Type of entity, e.g. `'group'` or `'user'`.

$subtype Entity subtype, e.g. `'blog'` or `'page'`.

You do not have to return a fallback icon from the hook handler. If no uploaded icon is found, the view system will scan the views (in this specific order) :

1. views/$viewtype/$icon_type/$entity_type/$entity_subtype.svg
2. views/$viewtype/$icon_type/$entity_type/$entity_subtype/$size.gif
3. views/$viewtype/$icon_type/$entity_type/$entity_subtype/$size.png
4. views/$viewtype/$icon_type/$entity_type/$entity_subtype/$size.jpg

Where

$viewtype Type of view, e.g. `'default'` or `'json'`.

$icon_type Icon type, e.g. `'icon'` or `'cover_image'`.

$entity_type Type of entity, e.g. `'group'` or `'user'`.

$entity_subtype Entity subtype, e.g. `'blog'` or `'page'` (or `'default'` if entity has not subtype).

$size Icon size (note that we do not use the size with svg icons)

Icon methods support passing an icon type if an entity has more than one icon. For example, a user might have an avatar and a cover photo icon. You would pass `'cover_photo'` as the icon type :

```
$object->saveIconFromUploadedFile('uploaded_photo', 'cover_photo');

$object->getIconUrl([
        'size' => 'medium',
        'type' => 'cover_photo'
]);
```

Note : Custom icon types (e.g. cover photos) only have a preset for *master* size, to add custom sizes use `entity:<icon_type>:url` *hook* to configure them.

By default icons will be stored in `/icons/<icon_type>/<size>.jpg` relative to entity's directory on filestore. To provide an alternative location, use the `entity:<icon_type>:file` *hook*.

Adding, reading and deleting annotations

Annotations could be used, for example, to track ratings. To annotate an entity you can use the object's `annotate()` method. For example, to give a blog post a rating of 5, you could use :

```
$blog_post->annotate('rating', 5);
```

To retrieve the ratings on the blog post, use `$blogpost->getAnnotations('rating')` and if you want to delete an annotation, you can operate on the `ElggAnnotation` class, eg `$annotation->delete()`.

Retrieving a single annotation can be done with `get_annotation()` if you have the annotation's ID. If you delete an ElggEntity of any kind, all its metadata, annotations, and relationships will be automatically deleted as well.

Extending ElggEntity

If you derive from one of the Elgg core classes, you'll need to tell Elgg how to properly instantiate the new type of object so that get_entity() et al. will return the appropriate PHP class. For example, if I customize ElggGroup in a class called « Committee », I need to make Elgg aware of the new mapping. Following is an example class extension :

```
// Class source
class Committee extends ElggGroup {

    protected function initializeAttributes() {
        parent::initializeAttributes();
        $this->attributes['subtype'] = 'committee';
    }

    // more customizations here
}
```

In your plugins `elgg-plugin.php` file add the `entities` section.

```
<?php // mod/example/elgg-plugin.php
return [
    // entities registration
    'entities' => [
                    [
                            'type' => 'group',
                            'subtype' => 'committee',
                            'class' => 'Committee',
                            'searchable' => true,
                    ],
            ],
];
```

The entities will be registered upon activation of the plugin.

Now if you invoke `get_entity()` with the GUID of a committee object, you'll get back an object of type Committee.

Advanced features

Entity URLs

Entity urls are provided by the `getURL()` interface and provide the Elgg framework with a common way of directing users to the appropriate display handler for any given object.

For example, a profile page in the case of users.

The url is set using the `elgg\_register\_entity\_url\_handler()` function. The function you register must return the appropriate url for the given type - this itself can be an address set up by a page handler.

The default handler is to use the default export interface.

Entity loading performance

`elgg_get_entities` has a couple options that can sometimes be useful to improve performance.

- **preload_owners** : If the entities fetched will be displayed in a list with the owner information, you can set this option to `true` to efficiently load the owner users of the fetched entities.
- **preload_containers** : If the entities fetched will be displayed in a list using info from their containers, you can set this option to `true` to efficiently load them.
- **distinct** : When Elgg fetches entities using an SQL query, Elgg must be sure that each entity row appears only once in the result set. By default it includes a `DISTINCT` modifier on the GUID column to enforce this, but some queries naturally return unique entities. Setting the `distinct` option to false will remove this modifier, and rely on the query to enforce its own uniqueness.

The internals of Elgg entity queries is a complex subject and it's recommended to seek help on the Elgg Community site before using the `distinct` option.

Custom database functionality

It is strongly recommended to use entities wherever possible. However, Elgg supports custom SQL queries using the database API.

Example : Run SQL script on plugin activation

This example shows how you can populate your database on plugin activation.

my_plugin/activate.php :

```
if (!elgg_get_plugin_setting('database_version', 'my_plugin') {
    run_sql_script(__DIR__ . '/sql/activate.sql');
    elgg_set_plugin_setting('database_version', 1, 'my_plugin');
}
```

my_plugin/sql/activate.sql :

```
-- Create some table
CREATE TABLE prefix_custom_table(
    id INTEGER AUTO_INCREMENT,
    name VARCHAR(32),
    description VARCHAR(32),
    PRIMARY KEY (id)
);

-- Insert initial values for table
INSERT INTO prefix_custom_table (name, description)
VALUES ('Peter', 'Some guy'), ('Lisa', 'Some girl');
```

Note that Elgg execute statements through PHPs built-in functions and have limited support for comments. I.e. only single line comments are supported and must be prefixed by « – » or « # « . A comment must start at the very beginning of a line.

Systemlog

Note : This section need some attention and will contain outdated information

The default Elgg system log is a simple way of recording what happens within an Elgg system. It's viewable and searchable directly from the administration panel.

System log storage

A system log row is stored whenever an event concerning an object whose class implements the *Loggable* interface is triggered. `ElggEntity` and `ElggExtender` implement *Loggable*, so a system log row is created whenever an event is performed on all objects, users, groups, sites, metadata and annotations.

Common events include :

- create
- update
- delete
- login

Creating your own system log

There are some reasons why you might want to create your own system log. For example, you might need to store a full copy of entities when they are updated or deleted, for auditing purposes. You might also need to notify an administrator when certain types of events occur.

To do this, you can create a function that listens to all events for all types of object :

```
register_elgg_event_handler('all','all','your_function_name');
```

Your function can then be defined as :

```
function your_function_name($object, $event) {
   if ($object instanceof Loggable) {
      ...
   }
}
```

You can then use the extra methods defined by *Loggable* to extract the information you need.

3.3.10 Error Handling

Under the hood, Elgg uses Monolog for logging errors to the server's error log (and stdout for CLI commands).

Monolog comes with a number of tools that can help administrators keep track of errors and debugging information.

You can add custom handlers (see Monolog documentation for a full list of handlers) :

```
// Add a new handler to notify a given email about a critical error
elgg()->logger->pushHandler(
        new \Monolog\Handler\NativeMailerHandler(
                'admin@example.com',
                'Critical error',
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'no-reply@mysite.com',
                \Monolog\Logger::CRITICAL
        )
);
```

3.3.11 List of events in core

For more information on how events work visit *Événements et Hooks des plugins*.

Contents

System events

plugins_boot, system Triggered just after the plugins are loaded. Rarely used. init, system is used instead.

init, system Plugins tend to use this event for initialization (extending views, registering callbacks, etc.)

ready, system Triggered after the `init, system` event. All plugins are fully loaded and the engine is ready to serve pages.

shutdown, system Triggered after the page has been sent to the user. Expensive operations could be done here and not make the user wait.

Note : Depending upon your server configuration the PHP output might not be shown until after the process is completed. This means that any long-running processes will still delay the page load.

Note : This event is prefered above using `register_shutdown_function` as you may not have access to all the Elgg services (eg. database) in the shutdown function but you will in the event.

Note : The Elgg session is already closed before this event. Manipulating session is not possible.

---

regenerate_site_secret :before, system Return false to cancel regenerating the site secret. You should also provide a message to the user.

regenerate_site_secret :after, system Triggered after the site secret has been regenerated.

log, systemlog Called for all triggered events by `system_log` plugin. Used internally by `system_log_default_logger()` to populate the `system_log` table.

upgrade, system Triggered after a system upgrade has finished. All upgrade scripts have run, but the caches are not cleared.

upgrade :execute, system Triggered as a sequence (so including `:before` and `:after`) when executing an `ElggUpgrade`. The `$object` of the event is the `ElggUpgrade`.

activate, plugin Return false to prevent activation of the plugin.

deactivate, plugin Return false to prevent deactivation of the plugin.

init :cookie, <name> Return false to override setting a cookie.

cache :flush, system Reset internal and external caches, by default including system_cache, simplecache, and memcache. One might use it to reset others such as APC, OPCache, or WinCache.

send :before, http_response Triggered before an HTTP response is sent. Handlers will receive an instance of *SymfonyComponentHttpFoundationResponse* that is to be sent to the requester. Handlers can terminate the event and prevent the response from being sent by returning *false*.

send :after, http_response Triggered after an HTTP response is sent. Handlers will receive an instance of *SymfonyComponentHttpFoundationResponse* that was sent to the requester.

reload :after, translations Triggered after the translations are (re)loaded.

User events

login :before, user Triggered during login. Returning false prevents the user from logging

login :after, user Triggered after the user logs in.

logout :before, user Triggered during logout. Returning false should prevent the user from logging out.

logout :after, user Triggered after the user logouts.

validate, user When a user registers, the user's account is disabled. This event is triggered to allow a plugin to determine how the user should be validated (for example, through an email with a validation link).

validate :after, user Triggered when user's account has been validated.

invalidate :after, user Triggered when user's account validation has been revoked.

profileupdate, user User has changed profile

profileiconupdate, user User has changed profile icon

ban, user Triggered before a user is banned. Return false to prevent.

unban, user Triggered before a user is unbanned. Return false to prevent.

make_admin, user Triggered before a user is promoted to an admin. Return false to prevent.

remove_admin, user Triggered before a user is demoted from an admin. Return false to prevent.

Relationship events

create, relationship Triggered after a relationship has been created. Returning false deletes the relationship that was just created.

delete, relationship Triggered before a relationship is deleted. Return false to prevent it from being deleted.

join, group Triggered after the user `$params['user']` has joined the group `$params['group']`.

leave, group Triggered before the user `$params['user']` has left the group `$params['group']`.

Entity events

create, <entity type> Triggered for user, group, object, and site entities after creation. Return false to delete entity.

update, <entity type> Triggered before an update for the user, group, object, and site entities. Return false to prevent update. The entity method `getOriginalAttributes()` can be used to identify which attributes have changed since the entity was last saved.

update :after, <entity type> Triggered after an update for the user, group, object, and site entities. The entity method `getOriginalAttributes()` can be used to identify which attributes have changed since the entity was last saved.

delete, <entity type> Triggered before entity deletion. Return false to prevent deletion.

disable, <entity type> Triggered before the entity is disabled. Return false to prevent disabling.

disable :after, <entity type> Triggered after the entity is disabled.

enable, <entity type> Return false to prevent enabling.

enable :after, <entity type> Triggered after the entity is enabled.

Metadata events

create, metadata Called after the metadata has been created. Return false to delete the metadata that was just created.

update, metadata Called after the metadata has been updated. Return false to *delete the metadata.*

delete, metadata Called before metadata is deleted. Return false to prevent deletion.

enable, metadata Called when enabling metadata. Return false to prevent enabling.

disable, metadata Called when disabling metadata. Return false to prevent disabling.

Annotation events

annotate, <entity type> Called before the annotation has been created. Return false to prevent annotation of this entity.

create, annotation Called after the annotation has been created. Return false to delete the annotation.

update, annotation Called after the annotation has been updated. Return false to *delete the annotation.*

delete, annotation Called before annotation is deleted. Return false to prevent deletion.

enable, annotation Called when enabling annotations. Return false to prevent enabling.

disable, annotations Called when disabling annotations. Return false to prevent disabling.

River events

created, river Called after a river item is created.

Note : Use the plugin hook `creating, river` to cancel creation (or alter options).

---

delete :before, river Triggered before a river item is deleted. Returning false cancels the deletion.

delete :after, river Triggered after a river item was deleted.

File events

upload :after, file Called after an uploaded file has been written to filestore. Receives an instance of `ElggFile` the uploaded file was written to. The `ElggFile` may or may not be an entity with a GUID.

Notes

Because of bugs in the Elgg core, some events may be thrown more than once on the same action. For example, `update, object` is thrown twice.

3.3.12 File System

Contents

Filestore

Location

Elgg's filestore is located in the site's `dataroot` that is configured during installation, and can be modified via site settings in Admin interface.

Directory Structure

The structure of the filestore is tied to file ownership by Elgg entities. Whenever the first file owned by an entity is written to the filestore, a directory corresponding to the entity GUID will be created within a parent bucket directory (buckets are bound to 5000 guids). E.g. files owned by user with guid 7777 will be located in `5000/7777/`.

When files are created, filenames can contain subdirectory names (often referred to as *$prefix* throughout the code). For instance, avatars of the above user, can be found under `5000/7777/profile/`.

File Objects

Writing Files

To write a file to the filestore, you would use an instance of `ElggFile`. Even though `ElggFile` extends *ElggObject* and can be stored as an actual Elgg entity, that is not always necessary (e.g. when writing thumbs of an image).

```
$file = new ElggFile();
$file->owner_guid = 7777;
$file->setFilename('portfolio/files/sample.txt');
$file->open('write');
$file->write('Contents of the file');
$file->close();
```

(suite sur la page suivante)

(suite de la page précédente)

```
// to uprade this file to an entity
$file->subtype = 'file';
$file->save();
```

Reading Files

You can read file contents using instanceof of `ElggFile`.

```
// from an Elgg entity
$file = get_entity($file_guid);
readfile($file->getFilenameOnFilestore());
```

```
// arbitrary file on the filestore
$file = new ElggFile();
$file->owner_guid = 7777;
$file->setFilename('portfolio/files/sample.txt');

// option 1
$file->open('read');
$contents = $file->grabFile();
$file->close();

// option 2
$contents = file_get_contents($file->getFilenameOnFilestore());
```

Serving Files

You can serve files from filestore using `elgg_get_inline_url()` and `elgg_get_download_url()`. Both functions accept 3 arguments :

— ``**file**`` An instance of `ElggFile` to be served
— ``**use_cookie**`` If set to true, validity of the URL will be limited to current session
— ``**expires**`` Expiration time of the URL

You can use `use_cookie` and `expires` arguments as means of access control. For example, users avatars in most cases have a long expiration time and do not need to be restricted by current session - this will allows browsers to cache the images and file service will send appropriate `Not Modified` headers on consecutive requests.

The default behaviour of `use_cookie` can be controlled on the admin security settings page.

For entities that are under Elgg's access control, you may want to use cookies to ensure that access settings are respected and users do not share download URLs with somebody else.

You can also invalidated all previously generated URLs by updating file's modified time, e.g. by using `touch()`.

Embedding Files

Please note that due to their nature inline and download URLs are not suitable for embedding. Embed URLs must be permanent, whereas inline and download URLs are volatile (bound to user session and file modification time).

To embed an entity icon, use `elgg_get_embed_url()`.

Handling File Uploads

In order to implement an action that saves a single file uploaded by a user, you can use the following approach :

```
// in your form
echo elgg_view('input/file', [
        'name' => 'upload',
        'label' => 'Select an image to upload',
        'help' => 'Only jpeg, gif and png images are supported',
]);
```

```
// in your action
$uploaded_file = elgg_get_uploaded_file('upload');
if (!$uploaded_file) {
        register_error("No file was uploaded");
        forward(REFERER);
}

$supported_mimes = [
        'image/jpeg',
        'image/png',
        'image/gif',
];

$mime_type = ElggFile::detectMimeType($uploaded_file->getPathname(), $uploaded_file->
↪getClientMimeType());
if (!in_array($mime_type, $supported_mimes)) {
        register_error("$mime_type is not supported");
        forward(REFERER);
}

$file = new ElggFile();
$file->owner_guid = elgg_get_logged_in_user_guid();
if ($file->acceptUploadedFile($uploaded_file)) {
        $file->save();
}
```

If your file input supports multiple files, you can iterate through them in your action :

```
// in your form
echo elgg_view('input/file', [
        'name' => 'upload[]',
        'multiple' => true,
        'label' => 'Select images to upload',
]);
```

```
// in your action
foreach (elgg_get_uploaded_files('upload') as $upload) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $file = new ElggFile();
        $file->owner_guid = elgg_get_logged_in_user_guid();
        if ($file->acceptUploadedFile($upload)) {
                $file->save();
        }
}
```

Note : If images are uploaded their is an automatic attempt to fix the orientation of the image.

Temporary files

If you ever need a temporary file you can use `elgg_get_temp_file()`. You'll get an instance of an `ElggTempFile` which has all the file functions of an `ElggFile`, but writes it's data to the systems temp folder.

Avertissement : It's not possible to save the `ElggTempFile` to the database. You'll get an `IOException` if you try.

3.3.13 Group Tools

Elgg groups allow group administrators to enable/disable various tools available within a group. These tools are provided by other plugins like blog or file.

Plugins can access group tool register via `elgg()->group_tools`.

```
elgg()->group_tools->register('my-tool', [
        'default_on' => false, // default is true
        'label' => elgg_echo('my-tool:checkbox:label'),
        'priority' => 300, // display this earlier than other modules/tools
]);
```

A registered tool will have an option to be toggled on the group edit form, and can have a profile view module associated with it. To add a profile module, simply add a corresponding view as `groups/profile/module/<tool_name>`.

```
// file: groups/profile/module/my-tool.php

echo elgg_view('groups/profile/module', [
        'title' => elgg_echo('my-tool'),
        'content' => 'Hello, world!',
]);
```

You can programmically enable and disable tools for a given group :

```
$group = get_entity($group_guid);

// enables the file tool for the group
$group->enableTool('file');

// disables the file tool for the group
$group->disableTool('file');
```

If you want to allow a certain feature in a group only if the group tool option is enabled, you can check this using `\ElggGroup::isToolEnabled($tool_option)`.

It is also a possibility to use a gatekeeper function to prevent access to a group page based on an enabled tool.

```
elgg_group_tool_gatekeeper('file', $group);
```

Voir aussi :

Read more about gatekeepers here : *Gatekeepers*

If you need the configured group tool options for a specific group you can use the `elgg()->group_tools->group($group)` function.

3.3.14 Plugin coding guidelines

In addition to the Elgg Coding Standards, these are guidelines for creating plugins. Core plugins are being updated to this format and all plugin authors should follow these guidelines in their own plugins.

Voir aussi :

Be sure to follow the *Plugin skeleton* for your plugin's layout.

Avertissement : *Don't Modify Core*

Contents

Use standardized routing with page handlers

- Example : Bookmarks plugin
- **Page handlers should accept the following standard URLs :**

Purpose	URL
All	page_handler/all
User	page_handler/owner/<username>
User friends'	page_handler/friends/<username>
Single entity	page_handler/view/<guid>/<title>
Add	page_handler/add/<container_guid>
Edit	page_handler/edit/<guid>
Group list	page_handler/group/<guid>/owner

- Include page handler scripts from the page handler. Almost every page handler should have a page handler script. (Example : `bookmarks/all` => `mod/bookmarks/views/default/resources/bookmarks/all.php`)
- Pass arguments like entity guids to the resource view via `$vars` in `elgg_view_resource()`.
- Call `elgg_gatekeeper()` and `elgg_admin_gatekeeper()` in the page handler function if required.

— The group URL should use views like `resources/groups/*.php` to render pages.
— Page handlers should not contain HTML.

Use standardized page handlers and scripts

— Example : Bookmarks plugin
— Store page functionality in `mod/<plugin>/views/default/resources/<page_handler>/<page_name>.php`
— Use `elgg_view_resource('<page_handler>/<page_name>')` to render that.
— Use the content page layout in page handler scripts : `$content = elgg_view_layout('content', $options);`
— Page handler scripts should not contain HTML
— Call `elgg_push_breadcrumb()` in the page handler scripts.
— No need to worry about setting the page owner if the URLs are in the standardized format
— For group content, check the `container_guid` by using `elgg_get_page_owner_entity()`

The object/<subtype> view

— Example : Bookmarks plugin
— Make sure there are views for `$vars['full_view'] == true` and `$vars['full_view'] == false`
— Check for the object in `$vars['entity']` . Use `elgg_instance_of()` to make sure it's the type entity you want. Return `true` to short circuit the view if the entity is missing or wrong.
— Use the new list body and list metadata views to help format. You should use almost no markup in these views.
— Update action structure - Example : Bookmarks plugin.
— Namespace action files and action names (example : `mod/blog/actions/blog/save.php` => `action/blog/save`)
— **Use the following action URLs :**

Purpose	URL
Add	action/plugin/save
Edit	action/plugin/save
Delete	action/plugin/delete

— Make the delete action accept `action/<handler>/delete?guid=<guid>` so the metadata entity menu has the correct URL by default

Actions

Actions are transient states to perform an action such as updating the database or sending a notification to a user. Used correctly, actions provide a level of access control and prevent against CSRF attacks.

Actions require action (CSRF) tokens to be submitted via GET/POST, but these are added automatically by elgg_view_form() and by using the `is_action` argument of the `output/url` view.

Action best practices

Action files are included within Elgg's action system ; like views, they are *not* regular scripts executable by users. Do not boot the Elgg core in your file and direct users to load it directly.

Because actions are time-sensitive they are not suitable for links in emails or other delayed notifications. An example of this would be invitations to join a group. The clean way to create an invitation link is to create a page handler for invitations and email that link to the user. It is then the page handler's responsibility to create the action links for a user to join or ignore the invitation request.

Consider that actions may be submitted via XHR requests, not just links or form submissions.

Directly calling a file

This is an easy one : **Don't do it**. With the exception of 3rd party application integration, there is not a reason to directly call a file in mods directory.

Recommended

These points are good ideas, but are not yet in the official guidelines. Following these suggestions will help to keep your plugin consistent with Elgg core.

- Update the widget views (see the blog or file widgets)
- Update the group profile "widget" using blog or file plugins as example
- **Update the forms**
  - Move form bodies to `/forms/<handler>/<action>` to use Evan's new `elgg_view_form()`
  - Use input views in form bodies rather than html
  - Add a function that prepares the form (see `mod/file/lib/file.php` for example)
  - Integrate sticky forms (see the file plugin's upload action and form prepare function)
- **Clean up CSS/HTML**
  - Should be able to remove almost all CSS (look for patterns that can be moved into core if you need CSS)
- Use hyphens rather than underscores in classes/ids
- Do not use the `bundled` category with your plugins. That is for plugins distributed with Elgg
- Don't use `register_shutdown_function` as you may not have access to certain Elgg parts anymore (eg database). Instead use the `shutdown system` event

3.3.15 Helper functions

Contents

Input and output

— `get_input($name)` Grabs information from a form field (or any variable passed using GET or POST). Also sanitises input, stripping Javascript etc.
— `set_input($name, $value)` Forces a value to a particular variable for subsequent retrieval by `get_input()`

Entity methods

— `$entity->getURL()` Returns the URL of any entity in the system
— `$entity->getGUID()` Returns the GUID of any entity in the system
— `$entity->canEdit()` Returns whether or not the current user can edit the entity
— `$entity->getOwnerEntity()` Returns the ElggUser owner of a particular entity

Entity and context retrieval

— `elgg_get_logged_in_user_entity()` Returns the ElggUser for the current user
— `elgg_get_logged_in_user_guid()` Returns the GUID of the current user
— `elgg_is_logged_in()` Is the viewer logged in
— `elgg_is_admin_logged_in()` Is the view an admin and logged in
— `elgg_gatekeeper()` Shorthand for checking if a user is logged in. Forwards user to front page if not
— `elgg_admin_gatekeeper()` Shorthand for checking the user is logged in and is an admin. Forwards user to front page if not
— `get_user($user_guid)` Given a GUID, returns a full ElggUser entity
— `elgg_get_page_owner_guid()` Returns the GUID of the current page owner, if there is one
— `elgg_get_page_owner_entity()` Like elgg_get_page_owner_guid() but returns the full entity
— `elgg_get_context()` Returns the current page's context - eg « blog » for the blog plugin, « thewire » for the wire, etc. Returns « main » as default
— `elgg_set_context($context)` Forces the context to be a particular value
— `elgg_push_context($context)` Adds a context to the stack
— `elgg_pop_context()` Removes the top context from the stack
— `elgg_in_context($context)` Checks if you're in a context (this checks the complete stack, eg. "widget" in "groups")

Plugins

— `elgg_is_active_plugin($plugin_id)` Check if a plugin is installed and enabled

Interface and annotations

— `elgg_view_image_block($icon, $info)` Return the result in a formatted list
— `elgg_view_comments($entity)` Returns any comments associated with the given entity
— `elgg_get_friendly_time($unix_timestamp)` Returns a date formatted in a friendlier way - « 18 minutes ago », « 2 days ago », etc.

Messages

— `system_message($message)` Registers a success message
— `register_error($message)` Registers an error message
— `elgg_view_message($type, $message)` Outputs a message

E-mail address formatting

Elgg has a helper class to aid in getting formatted e-mail addresses : `\Elgg\Email\Address`.

```
// the constructor takes two variables
// first is the email address, this is REQUIRED
// second is the name, this is optional
$address = new \Elgg\Email\Address('example@elgg.org', 'Example');

// this will result in 'Example <example@elgg.org>'
echo $address->toString();

// to change the name use:
$address->setName('New Example');

// to change the e-mail address use:
$address->setEmail('example2@elgg.org');
```

There are some helper functions available
— `\Elgg\Email\Address::fromString($string)` Will return an `\Elgg\Email\Address` class with e-mail and name set, provided a formatted string (eg. `Example <example@elgg.org>`)
— `\Elgg\Email\Address::getFormattedEmailAddress($email, $name)` Will return a formatted string provided an e-mail address and optionaly a name

3.3.16 List of plugin hooks in core

For more information on how hooks work visit *Événements et Hooks des plugins*.

Contents :local :

System hooks

page_owner, system Filter the page_owner for the current page. No options are passed.

siteid, system

gc, system Allows plugins to run garbage collection for `$params['period']`.

unit_test, system Add a Simple Test test. (Deprecated.)

diagnostics :report, system Filter the output for the diagnostics report download.

cron, <period> Triggered by cron for each period.

validate, input Filter GET and POST input. This is used by `get_input()` to sanitize user input.

prepare, html Triggered by `elgg_format_html()` and used to prepare untrusted HTML.

The `$return` value is an array :

— `html` - HTML string being prepared
— `options` - Preparation options

diagnostics :report, system Filters the output for a diagnostic report.

debug, log Triggered by the Logger. Return false to stop the default logging method. `$params` includes :
— **level - The debug level. One of :**
  — `Elgg_Logger::OFF`
  — `Elgg_Logger::ERROR`
  — `Elgg_Logger::WARNING`
  — `Elgg_Logger::NOTICE`
  — `Elgg_Logger::INFO`
— msg - The message
— display - Should this message be displayed ?

format, friendly :title Formats the « friendly » title for strings. This is used for generating URLs.

format, friendly :time Formats the « friendly » time for the timestamp `$params['time']`.

format, strip_tags Filters a string to remove tags. The original string is passed as `$params['original_string']` and an optional set of allowed tags is passed as `$params['allowed_tags']`.

output :before, page In `elgg_view_page()`, this filters `$vars` before it's passed to the page shell view (`page/<page_shell>`). To stop sending the X-Frame-Options header, unregister the handler `_elgg_views_send_header_x_frame_options()` from this hook.

output, page In `elgg_view_page()`, this filters the output return value.

parameters, menu :<menu_name> Triggered by `elgg_view_menu()`. Used to change menu variables (like sort order) before rendering.

The `$params` array will contain :
— `name` - name of the menu
— `sort_by` - preferring sorting parameter
— other parameters passed to `elgg_view_menu()`

register, menu :<menu_name> Filters the initial list of menu items pulled from configuration, before the menu has been split into sections. Triggered by `elgg_view_menu()` and `elgg()->menus->getMenu()`.

The `$params` array will contain parameters returned by `parameters, menu:<menu_name>` hook.

The return value is an instance of `\Elgg\Collections\Collection` containing `\ElggMenuItem` objects.

Hook handlers can add/remove items to the collection using the collection API, as well as array access operations.

prepare, menu :<menu_name> Filters the array of menu sections before they're displayed. Each section is a string key mapping to an area of menu items. This is a good hook to sort, add, remove, and modify menu items. Triggered by `elgg_view_menu()` and `elgg()->menus->prepareMenu()`.

The `$params` array will contain :
— `selected_item` - `ElggMenuItem` selected in the menu, if any
The return value is an instance of `\Elgg\Menu\PreparedMenu`. The prepared menu is a collection of `\Elgg\Menu\MenuSection`, which in turn are collections of `\ElggMenuItem` objects.

register, menu :filter :<filter_id> Allows plugins to modify layout filter tabs on layouts that specify `<filter_id>` parameter. Parameters and return values are same as in `register, menu:<menu_name>` hook.

filter_tabs, <context> Filters the array of `ElggMenuItem` used to display the All/Mine/Friends tabs. The `$params` array includes :
— `selected` : the selected menu item name
— `user` : the logged in `ElggUser` or `null`
— `vars` : The `$vars` argument passed to `elgg_get_filter_tabs`

creating, river The options for `elgg_create_river_item` are filtered through this hook. You may alter values or return `false` to cancel the item creation.

simplecache :generate, <view> Filters the view output for a `/cache` URL when simplecache is enabled.

cache :generate, <view> Filters the view output for a `/cache` URL when simplecache is disabled. Note this will be fired for every `/cache` request–no Expires headers are used when simplecache is disabled.

prepare, breadcrumbs In `elgg_get_breadcrumbs()`, this filters the registered breadcrumbs before returning them, allowing a plugin to alter breadcrumb strategy site-wide. `$params` array includes :
— `breadcrumbs` - an array of bredcrumbs, each with `title` and `link` keys
— `identifier` - route identifier of the current page
— `segments` - route segments of the current page

add, river

elgg.data, site Filters cached configuration data to pass to the client. *More info*

elgg.data, page Filters uncached, page-specific configuration data to pass to the client. *More info*

registration_url, site Filters site's registration URL. Can be used by plugins to attach invitation codes, referrer codes etc. to the registration URL. `$params` array contains an array of query elements added to the registration URL by the invoking script. The hook must return an absolute URL to the registration page.

login_url, site Filters site's login URL. `$params` array contains an array of query elements added to the login URL by the invoking script. The hook must return an absolute URL of the login page.

commands, cli Allows plugins to register their own commands executable via `elgg-cli` binary. Handlers must return an array of command class names. Commands must extend `\Elgg\Cli\Command` to be executable.

seeds, database Allows plugins to register their own database seeds. Seeds populate the database with fake entities for testing purposes. Seeds must extend `\Elgg\Database\Seeds\Seed` class to be executable via `elgg-cli database:seed`.

languages, translations Allows plugins to add/remove languages from the configurable languages in the system.

User hooks

usersettings :save, user Triggered in the aggregate action to save user settings. The hook handler must return `false` to prevent sticky forms from being cleared (i.e. to indicate that some of the values were not saved). Do not return `true` from your hook handler, as you will override other hooks" output, instead return `null` to indicate successful operation.

The `$params` array will contain :
— `user` - `\ElggUser`, whose settings are being saved
— `request` - `\Elgg\Request` to the action controller

change :email, user Triggered before the user email is changed. Allows plugins to implement additional logic required to change email, e.g. additional email validation. The hook handler must return false to prevent the email from being changed right away.

The `$params` array will contain :
— `user` - `\ElggUser`, whose settings are being saved
— `email` - Email address that passes sanity checks
— `request` - `\Elgg\Request` to the action controller

access :collections :write, user Filters an array of access permissions that the user `$params['user_id']` is allowed to save content with. Permissions returned are of the form (id => "Human Readable Name").

registeruser :validate :username, all Return boolean for if the string in `$params['username']` is valid for a username. Hook handler can throw `\RegistrationException` with an error message to be shown to the user.

registeruser :validate :password, all Return boolean for if the string in `$params['password']` is valid for a password. Hook handler can throw `\RegistrationException` with an error message to be shown to the user.

registeruser :validate :email, all Return boolean for if the string in `$params['email']` is valid for an email address. Hook handler can throw `\RegistrationException` with an error message to be shown to the user.

register, user Triggered by the `register` action after the user registers. Return `false` to delete the user. Note the function `register_user` does *not* trigger this hook. Hook handlers can throw `\RegistrationException` with an error message to be displayed to the user.

The `$params` array will contain :
— `user` - Newly registered user entity
— All parameters sent with the request to the action (incl. `password`, `friend_guid`, `invitecode` etc)

login :forward, user Filters the URL to which the user will be forwarded after login.

find_active_users, system Return the number of active users.

status, user Triggered by The Wire when adding a post.

username :character_blacklist, user Filters the string of blacklisted characters used to validate username during registration. The return value should be a string consisting of the disallowed characters. The default string can be found from `$params['blacklist']`.

Object hooks

comments, <entity_type> Triggered in `elgg_view_comments()`. If returning content, this overrides the `page/elements/comments` view.

comments :count, <entity_type> Return the number of comments on `$params['entity']`.

likes :count, <entity_type> Return the number of likes for `$params['entity']`.

Access hooks

access_collection :url, access_collection Can be used to filter the URL of the access collection.

The `$params` array will contain :
— `access_collection` - *ElggAccessCollection*

access_collection :name, access_collection Can be used to filter the display name (readable access level) of the access collection.

The `$params` array will contain :
— `access_collection` - *ElggAccessCollection*

access :collections :read, user Filters an array of access IDs that the user `$params['user_id']` can see.

> **Avertissement :** The handler needs to either not use parts of the API that use the access system (triggering the hook again) or to ignore the second call. Otherwise, an infinite loop will be created.

access :collections :write, user Filters an array of access IDs that the user `$params['user_id']` can write to. In get_write_access_array(), this hook filters the return value, so it can be used to alter the available options in the input/access view. For core plugins, the value « input_params » has the keys « entity » (ElggEntity|false), « entity_type » (string), « entity_subtype » (string), « container_guid » (int) are provided. An empty entity value generally means the form is to create a new object.

> **Avertissement :** The handler needs to either not use parts of the API that use the access system (triggering the hook again) or to ignore the second call. Otherwise, an infinite loop will be created.

access :collections :addcollection, collection Triggered after an access collection `$params['collection_id']` is created.

access :collections :deletecollection, collection Triggered before an access collection `$params['collection_id']` is deleted. Return false to prevent deletion.

access :collections :add_user, collection Triggered before adding user `$params['user_id']` to collection `$params['collection_id']`. Return false to prevent adding.

access :collections :remove_user, collection Triggered before removing user `$params['user_id']` to collection `$params['collection_id']`. Return false to prevent removal.

get_sql, access Filters SQL clauses restricting/allowing access to entities and annotations.

The hook is triggered regardless if the access is ignored. The handlers may need to check if access is ignored and return early, if appended clauses should only apply to access controlled contexts.

`$return` value is a nested array of `ands` and `ors`.

`$params` includes :
— `table_alias` - alias of the main table used in select clause
— `ignore_access` - whether ignored access is enabled
— `use_enabled_clause` - whether disabled entities are shown/hidden
— `access_column` - column in the main table containing the access collection ID value
— `owner_guid_column` - column in the main table referencing the GUID of the owner
— `guid_column` - column in the main table referencing the GUID of the entity
— `enabled_column` - column in the main table referencing the enabled status of the entity
— `query_builder` - an instance of the `QueryBuilder`

Action hooks

action, <action> Deprecated. Use `'action:validate', <action>` hook instead. Triggered before executing action scripts. Return false to abort action.

action :validate, <action>

Trigger before action script/controller is executed. This hook should be used to validate/alter user input, before proceeding with the action. The hook handler can throw an instance of `\Elgg\ValidationException` or return `false` to terminate further execution.

`$params` array includes :
— `request` - instance of `\Elgg\Request`

action_gatekeeper :permissions :check, all Triggered after a CSRF token is validated. Return false to prevent validation.

action_gatekeeper :upload_exceeded_msg, all Triggered when a POST exceeds the max size allowed by the server. Return an error message to display.

forward, <reason> Filter the URL to forward a user to when `forward($url, $reason)` is called. In certain cases, the `params` array will contain an instance of `HttpException` that triggered the error.

response, action :<action> Filter an instance of `\Elgg\Http\ResponseBuilder` before it is sent to the client. This hook can be used to modify response content, status code, forward URL, or set additional response headers. Note that the `<action>` value is parsed from the request URL, therefore you may not be able to filter the responses of *action()* calls if they are nested within the another action script file.

Ajax

ajax_response, * When the `elgg/Ajax` AMD module is used, this hook gives access to the response object (`\Elgg\Services\AjaxResponse`) so it can be altered/extended. The hook type depends on the method call :

elgg/Ajax method	plugin hook type
action()	action :<action_name>
path()	path :<url_path>
view()	view :<view_name>
form()	form :<action_name>

output, ajax This filters the JSON output wrapper returned to the legacy ajax API (`elgg.ajax`, `elgg.action`, etc.). Plugins can alter the output, forward URL, system messages, and errors. For the `elgg/Ajax` AMD module, use the `ajax_response` hook documented above.

Permission hooks

container_logic_check, <entity_type> Triggered by `ElggEntity:canWriteToContainer()` before triggering `permissions_check` and `container_permissions_check` hooks. Unlike permissions hooks, logic check can be used to prevent certain entity types from being contained by other entity types, e.g. discussion replies should only be contained by discussions. This hook can also be used to apply status logic, e.g. do disallow new replies for closed discussions.

The handler should return `false` to prevent an entity from containing another entity. The default value passed to the hook is `null`, so the handler can check if another hook has modified the value by checking if return value is set. Should this hook return `false`, `container_permissions_check` and `permissions_check` hooks will not be triggered.

The `$params` array will contain :
— `container` - An entity that will be used as a container
— `user` - User who will own the entity to be written to container
— `subtype` - Subtype of the entity to be written to container (entity type is assumed from hook type)

container_permissions_check, <entity_type> Return boolean for if the user `$params['user']` can use the entity `$params['container']` as a container for an entity of `<entity_type>` and subtype `$params['subtype']`.

In the rare case where an entity is created with neither the `container_guid` nor the `owner_guid` matching the logged in user, this hook is called *twice*, and in the first call `$params['container']` will be the *owner*, not the entity's real container.

The `$params` array will contain :
— `container` - An entity that will be used as a container
— `user` - User who will own the entity to be written to container
— `subtype` - Subtype of the entity to be written to container (entity type is assumed from hook type)

permissions_check, <entity_type> Return boolean for if the user `$params['user']` can edit the entity `$params['entity']`.

permissions_check :delete, <entity_type> Return boolean for if the user `$params['user']` can delete the entity `$params['entity']`. Defaults to `$entity->canEdit()`.

permissions_check :delete, river Return boolean for if the user `$params['user']` can delete the river item `$params['item']`. Defaults to `true` for admins and `false` for other users.

permissions_check :download, file Return boolean for if the user `$params['user']` can download the file in `$params['entity']`.

The `$params` array will contain :
— `entity` - Instance of `ElggFile`
— `user` - User who will download the file

permissions_check, widget_layout Return boolean for if `$params['user']` can edit the widgets in the context passed as `$params['context']` and with a page owner of `$params['page_owner']`.

permissions_check :metadata, <entity_type> (Deprecated) Return boolean for if the user `$params['user']` can edit the metadata `$params['metadata']` on the entity `$params['entity']`.

permissions_check :comment, <entity_type> Return boolean for if the user `$params['user']` can comment on the entity `$params['entity']`.

permissions_check :annotate :<annotation_name>, <entity_type> Return boolean for if the user `$params['user']` can create an annotation `<annotation_name>` on the entity `$params['entity']`. If logged in, the default is true.

Note : This is called before the more general `permissions_check:annotate` hook, and its return value is that hook's initial value.

---

permissions_check :annotate, <entity_type> Return boolean for if the user `$params['user']` can create an annotation `$params['annotation_name']` on the entity `$params['entity']`. if logged in, the default is true.

permissions_check :annotation Return boolean for if the user in `$params['user']` can edit the annotation `$params['annotation']` on the entity `$params['entity']`. The user can be null.

fail, auth Return the failure message if authentication failed. An array of previous PAM failure methods is passed as `$params`.

api_key, use Triggered by `api_auth_key()`. Returning false prevents the key from being authenticated.

gatekeeper, <entity_type> :<entity_subtype> Filters the result of `elgg_entity_gatekeeper()` to prevent or allow access to an entity that user would otherwise have or not have access to. A handler can return `false` or an instance of `HttpException` to prevent access to an entity. A handler can return `true` to override the result of the gatekeeper. **Important** that the entity received by this hook is fetched with ignored access and including disabled entities, so you have to be careful to not bypass the access system.

`$params` array includes :
— `entity` - Entity that is being accessed
— `user` - User accessing the entity (`null` implies logged in user)

Notifications

These hooks are listed chronologically in the lifetime of the notification event. Note that not all hooks apply to instant notifications.

enqueue, notification Can be used to prevent a notification event from sending **subscription** notifications. Hook handler must return `false` to prevent a subscription notification event from being enqueued.

`$params` array includes :
— `object` - object of the notification event
— `action` - action that triggered the notification event. E.g. corresponds to `publish` when `elgg_trigger_event('publish', 'object', $object)` is called

get, subscriptions

Filters subscribers of the notification event. Applies to **subscriptions** and **instant** notifications. In case of a subscription event, by default, the subscribers list consists of the users subscribed to the container entity of the event object. In case of an instant notification event, the subscribers list consists of the users passed as recipients to `notify_user()`

IMPORTANT Always validate the notification event, object and/or action types before adding any new recipients to ensure that you do not accidentally dispatch notifications to unintended recipients. Consider a situation, where a mentions plugin sends out an instant notification to a mentioned user - any hook acting on a subject or an object without validating an event or action type (e.g. including an owner of the original wire thread) might end up sending notifications to wrong users.

`$params` array includes :
— `event` - `\Elgg\Notifications\NotificationEvent` instance that describes the notification event
— `origin` - `subscriptions_service` or `instant_notifications`

— `methods_override` - delivery method preference for instant notifications
Handlers must return an array in the form :

```
array(
        <user guid> => array('sms'),
        <user_guid2> => array('email', 'sms', 'ajax')
);
```

send :before, notifications Triggered before the notification event queue is processed. Can be used to terminate the notification event. Applies to **subscriptions** and **instant** notifications.

`$params` array includes :
— `event` - `\Elgg\Notifications\NotificationEvent` instance that describes the notification event
— `subscriptions` - a list of subscriptions. See `'get', 'subscriptions'` hook for details

prepare, notification A high level hook that can be used to alter an instance of `\Elgg\Notifications\Notification` before it is sent to the user. Applies to **subscriptions** and **instant** notifications. This hook is triggered before a more granular `'prepare', 'notification:<action>:<entity_type>:<entity_subtype>'` and after `'send:before', 'notifications'`. Hook handler should return an altered notification object.

`$params` may vary based on the notification type and may include :
— `event` - `\Elgg\Notifications\NotificationEvent` instance that describes the notification event
— `object` - object of the notification `event`. Can be `null` for instant notifications
— `action` - action that triggered the notification `event`. May default to `notify_user` for instant notifications
— `method` - delivery method (e.g. `email`, `site`)
— `sender` - sender
— `recipient` - recipient
— `language` - language of the notification (recipient's language)
— `origin` - `subscriptions_service` or `instant_notifications`

prepare, notification :<action> :<entity_type> :<entity_type> A granular hook that can be used to filter a notification `\Elgg\Notifications\Notification` before it is sent to the user. Applies to **subscriptions** and **instant** notifications. In case of instant notifications that have not received an object, the hook will be called as `'prepare', 'notification:<action>'`. In case of instant notifications that have not received an action name, it will default to `notify_user`.

`$params` include :
— `event` - `\Elgg\Notifications\NotificationEvent` instance that describes the notification event
— `object` - object of the notification `event`. Can be `null` for instant notifications
— `action` - action that triggered the notification `event`. May default to `notify_user` for instant notifications
— `method` - delivery method (e.g. `email`, `site`)
— `sender` - sender
— `recipient` - recipient
— `language` - language of the notification (recipient's language)
— `origin` - `subscriptions_service` or `instant_notifications`

format, notification :<method> This hook can be used to format a notification before it is passed to the `'send', 'notification:<method>'` hook. Applies to **subscriptions** and **instant** notifications. The hook handler should return an instance of `\Elgg\Notifications\Notification`. The hook does not receive any `$params`. Some of the use cases include :
— Strip tags from notification title and body for plaintext email notifications
— Inline HTML styles for HTML email notifications
— Wrap notification in a template, add signature etc.

send, notification :<method> Delivers a notification. Applies to **subscriptions** and **instant** notifications. The handler must return `true` or `false` indicating the success of the delivery.

`$params` array includes :
— `notification` - a notification object `\Elgg\Notifications\Notification`

send :after, notifications Triggered after all notifications in the queue for the notifications event have been processed. Applies to **subscriptions** and **instant** notifications.

`$params` array includes :
— `event` - `\Elgg\Notifications\NotificationEvent` instance that describes the notification event
— `subscriptions` - a list of subscriptions. See `'get', 'subscriptions'` hook for details
— `deliveries` - a matrix of delivery statuses by user for each delivery method

Emails

prepare, system :email Triggered by `elgg_send_email()`. Applies to all outgoing system and notification emails. This hook allows you to alter an instance of `\Elgg\Email` before it is passed to the email transport. This hook can be used to alter the sender, recipient, subject, body, and/or headers of the email.

`$params` are empty. The `$return` value is an instance of `\Elgg\Email`.

validate, system :email Triggered by `elgg_send_email()`. Applies to all outgoing system and notification emails. This hook allows you to suppress or whitelist outgoing emails, e.g. when the site is in a development mode. The handler must return `false` to supress the email delivery.

`$params` contains :
— `email` - An instance of `\Elgg\Email`

transport, system :email Triggered by `elgg_send_email()`. Applies to all outgoing system and notification emails. This hook allows you to implement a custom email transport, e.g. delivering emails via a third-party proxy service such as SendGrid or Mailgun. The handler must return `true` to indicate that the email was transported.

`$params` contains :
— `email` - An instance of `\Elgg\Email`

zend :message, system :email Triggered by the default email transport handler (Elgg uses `zendframework/zend-mail`). Applies to all outgoing system and notification emails that were not transported using the **transport, system :email** hook. This hook allows you to alter an instance of `\Zend\Mail\Message` before it is passed to the Zend email transport.

`$params` contains :
— `email` - An instance of `\Elgg\Email`

Routing

route :config, <route_name> Allows altering the route configuration before it is registered. This hook can be used to alter the path, default values, requirements, as well as to set/remove middleware. Please note that the handler for this hook should be registered outside of the `init` event handler, as core routes are registered during `plugins_boot` event.

route :rewrite, <identifier> Allows altering the site-relative URL path for an incoming request. See *Routing* for details. Please note that the handler for this hook should be registered outside of the `init` event handler, as route rewrites take place after `plugins_boot` event has completed.

response, path :<path> Filter an instance of `\Elgg\Http\ResponseBuilder` before it is sent to the client. This hook type will only be used if the path did not start with « action/ » or « ajax/ ». This hook can be used to modify response content, status code, forward URL, or set additional response headers. Note that the `<path>` value is parsed from the request URL, therefore plugins using the `route` hook should use the original `<path>` to filter the response, or switch to using the `route:rewrite` hook.

ajax_response, path :<path> Filters ajax responses before they're sent back to the `elgg/Ajax` module. This hook type will only be used if the path did not start with « action/ » or « ajax/ ».

Views

view_vars, <view_name> Filters the `$vars` array passed to the view

view, <view_name> Filters the returned content of the view

layout, page In `elgg_view_layout()`, filters the layout name. `$params` array includes :
- `identifier` - ID of the page being rendered
- `segments` - URL segments of the page being rendered
- other `$vars` received by `elgg_view_layout()`

shell, page In `elgg_view_page()`, filters the page shell name

head, page In `elgg_view_page()`, filters `$vars['head']` Return value contains an array with `title`, `metas` and `links` keys, where `metas` is an array of elements to be formatted as `<meta>` head tags, and `links` is an array of elements to be formatted as `<link>` head tags. Each meta and link element contains a set of key/value pairs that are formatted into html tag attributes, e.g.

```
return [
   'title' => 'Current page title',
   'metas' => [
      'viewport' => [
         'name' => 'viewport',
         'content' => 'width=device-width',
      ]
   ],
   'links' => [
      'rss' => [
         'rel' => 'alternative',
         'type' => 'application/rss+xml',
         'title' => 'RSS',
         'href' => elgg_format_url($url),
      ],
      'icon-16' => [
         'rel' => 'icon',
         'sizes' => '16x16',
         'type' => 'image/png',
         'href' => elgg_get_simplecache_url('graphics/favicon-16.png'),
      ],
   ],
];
```

ajax_response, view :<view> Filters `ajax/view/` responses before they're sent back to the `elgg/Ajax` module.

ajax_response, form :<action> Filters `ajax/form/` responses before they're sent back to the `elgg/Ajax` module.

response, view :<view_name> Filter an instance of `\Elgg\Http\ResponseBuilder` before it is sent to the client. Applies to request to `/ajax/view/<view_name>`. This hook can be used to modify response content, status code, forward URL, or set additional response headers.

response, form :<form_name> Filter an instance of `\Elgg\Http\ResponseBuilder` before it is sent to the client. Applies to request to `/ajax/form/<form_name>`. This hook can be used to modify response content, status code, forward URL, or set additional response headers.

table_columns :call, <name> When the method `elgg()->table_columns->$name()` is called, this hook is called to allow plugins to override or provide an implementation. Handlers receive the method arguments via `$params['arguments']` and should return an instance of `Elgg\Views\TableColumn` if they wish to specify the column directly.

vars :compiler, css Allows plugins to alter CSS variables passed to CssCrush during compilation. See *CSS variables <_guides/theming#css-vars>*.

Files

download :url, file

Allows plugins to filter the download URL of the file. By default, the download URL is generated by the file service.

`$params` array includes :
— `entity` - instance of `ElggFile`

inline :url, file

Allows plugins to filter the inline URL of the image file. By default, the inline URL is generated by the file service.

`$params` array includes :
— `entity` - instance of `ElggFile`

mime_type, file Return the mimetype for the filename `$params['filename']` with original filename `$params['original_filename']` and with the default detected mimetype of `$params['default']`.

simple_type, file In `elgg_get_file_simple_type()`, filters the return value. The hook uses `$params['mime_type']` (e.g. `application/pdf` or `image/jpeg`) and determines an overall category like `document` or `image`. The bundled file plugin and other-third party plugins usually store `simpletype` metadata on file entities and make use of it when serving icons and constructing `ege*` filters and menus.

upload, file Allows plugins to implement custom logic for moving an uploaded file into an instance of `ElggFile`. The handler must return `true` to indicate that the uploaded file was moved. The handler must return `false` to indicate that the uploaded file could not be moved. Other returns will indicate that `ElggFile::acceptUploadedFile` should proceed with the default upload logic.

`$params` array includes :
— `file` - instance of `ElggFile` to write to
— `upload` - instance of Symfony's `UploadedFile`

Search

search :results, <search_type> Triggered by `elgg_search()`. Receives normalized options suitable for `elgg_get_entities()` call and must return an array of entities matching search options. This hook is designed for use by plugins integrating third-party indexing services, such as Solr and Elasticsearch.

search :params, <search_type> Triggered by `elgg_search()`. Filters search parameters (query, sorting, search fields etc) before search clauses are prepared for a given search type. Elgg core only provides support for `entities` search type.

search :fields, <entity_type> Triggered by `elgg_search()`. Filters search fields before search clauses are prepared. `$return` value contains an array of names for each entity property type, which should be matched against the search query. `$params` array contains an array of search params passed to and filtered by `elgg_search()`.

```
return [
    'attributes' => [],
    'metadata' => ['title', 'description'],
    'annotations' => ['revision'],
    'private_settings' => ['internal_notes'],
];
```

search :fields, <entity_type> :<entity_subtype> See **search :fields, <entity_type>**

search :fields, <search_type> See **search :fields, <entity_type>**

search :options, <entity_type> Triggered by `elgg_search()`. Prepares search clauses (options) to be passed to `elgg_get_entities()`.

search :options, <entity_type> :<entity_subtype> See **search :options, <entity_type>**

search :options, <search_type> See **search :options, <entity_type>**

search :config, search_types Implemented in the **search** plugin. Filters an array of custom search types. This allows plugins to add custom search types (e.g. tag or location search). Adding a custom search type will extend the search plugin user interface with appropriate links and lists.

search :config, type_subtype_pairs Implemented in the **search** plugin. Filters entity type/subtype pairs before entity search is performed. Allows plugins to remove certain entity types/subtypes from search results, group multiple subtypes together, or to reorder search sections.

search :format, entity Implemented in the **search** plugin. Allows plugins to populate entity's volatile data before it's passed to search view. This is used for highlighting search hit, extracting relevant substrings in long text fields etc.

Other

config, comments_per_page Filters the number of comments displayed per page. Default is 25. `$params['entity']` will hold the containing entity or null if not provided.

config, comments_latest_first Filters the order of comments. Default is `true` for latest first. `$params['entity']` will hold the containing entity or null if not provided.

default, access In get_default_access(), this hook filters the return value, so it can be used to alter the default value in the input/access view. For core plugins, the value « input_params » has the keys « entity » (ElggEntity|false), « entity_type » (string), « entity_subtype » (string), « container_guid » (int) are provided. An empty entity value generally means the form is to create a new object.

classes, icon Can be used to filter CSS classes applied to icon glyphs. By default, Elgg uses FontAwesome. Plugins can use this hook to switch to a different font family and remap icon classes.

entity :icon :sizes, <entity_type> Triggered by `elgg_get_icon_sizes()` and sets entity type/subtype specific icon sizes. `entity_subtype` will be passed with the `$params` array to the callback.

entity :<icon_type> :sizes, <entity_type>

Allows filtering sizes for custom icon types, see `entity:icon:sizes, <entity_type>`.

The hook must return an associative array where keys are the names of the icon sizes (e.g. « large »), and the values are arrays with the following keys :

- `w` - Width of the image in pixels
- `h` - Height of the image in pixels
- `square` - Should the aspect ratio be a square (true/false)
- `upscale` - Should the image be upscaled in case it is smaller than the given width and height (true/false)
- `crop` - Is cropping allowed on this image size (true/false, default : true)

If the configuration array for an image size is empty, the image will be saved as an exact copy of the source without resizing or cropping.

Example :

```
return [
        'small' => [
                'w' => 60,
                'h' => 60,
                'square' => true,
                'upscale' => true,
        ],
        'large' => [
                'w' => 600,
                'h' => 600,
                'upscale' => false,
        ],
        'original' => [],
];
```

entity :icon :url, <entity_type> Triggered when entity icon URL is requested, see *entity icons*. Callback should return URL for the icon of size `$params['size']` for the entity `$params['entity']`. Following parameters are available through the `$params` array :

entity Entity for which icon url is requested.

viewtype The type of *view* e.g. `'default'` or `'json'`.

size Size requested, see *entity icons* for possible values.

Example on how one could default to a Gravatar icon for users that have not yet uploaded an avatar :

```
// Priority 600 so that handler is triggered after avatar handler
elgg_register_plugin_hook_handler('entity:icon:url', 'user', 'gravatar_icon_handler',␣
↪600);

/**
 * Default to icon from gravatar for users without avatar.
 */
function gravatar_icon_handler($hook, $type, $url, $params) {
        // Allow users to upload avatars
        if ($params['entity']->icontime) {
                return $url;
        }

        // Generate gravatar hash for user email
        $hash = md5(strtolower(trim($params['entity']->email)));

        // Default icon size
        $size = '150x150';

        // Use configured size if possible
        $config = elgg_get_icon_sizes('user');
        $key = $params['size'];
        if (isset($config[$key])) {
                $size = $config[$key]['w'] . 'x' . $config[$key]['h'];
        }

        // Produce URL used to retrieve icon
        return "http://www.gravatar.com/avatar/$hash?s=$size";
}
```

entity :<icon_type> :url, <entity_type> Allows filtering URLs for custom icon types, see `entity:icon:url, <entity_type>`

entity :icon :file, <entity_type> Triggered by `ElggEntity::getIcon()` and allows plugins to provide an alternative `ElggIcon` object that points to a custom location of the icon on filestore. The handler must return an instance of `ElggIcon` or an exception will be thrown.

entity :<icon_type> :file, <entity_type> Allows filtering icon file object for custom icon types, see `entity:icon:file, <entity_type>`

entity :<icon_type> :prepare, <entity_type> Triggered by `ElggEntity::saveIcon*()` methods and can be used to prepare an image from uploaded/linked file. This hook can be used to e.g. rotate the image before it is resized/cropped, or it can be used to extract an image frame if the uploaded file is a video. The handler must return an instance of `ElggFile` with a *simpletype* that resolves to *image*. The `$return` value passed to the hook is an instance of `ElggFile` that points to a temporary copy of the uploaded/linked file.

The `$params` array contains :
— `entity` - entity that owns the icons
— `file` - original input file before it has been modified by other hooks

entity :<icon_type> :save, <entity_type> Triggered by `ElggEntity::saveIcon*()` methods and can be used to apply custom image manipulation logic to resizing/cropping icons. The handler must return `true` to prevent the core APIs from resizing/cropping icons. The `$params` array contains :
— `entity` - entity that owns the icons
— `file` - `ElggFile` object that points to the image file to be used as source for icons
— `x1`, `y1`, `x2`, `y2` - cropping coordinates

entity :<icon_type> :saved, <entity_type> Triggered by `ElggEntity::saveIcon*()` methods once icons have been created. This hook can be used by plugins to create river items, update cropping coordinates for custom icon types etc. The handler can access the created icons using `ElggEntity::getIcon()`. The `$params` array contains :
— `entity` - entity that owns the icons
— `x1`, `y1`, `x2`, `y2` - cropping coordinates

entity :<icon_type> :delete, <entity_type> Triggered by `ElggEntity::deleteIcon()` method and can be used for clean up operations. This hook is triggered before the icons are deleted. The handler can return `false` to prevent icons from being deleted. The `$params` array contains :
— `entity` - entity that owns the icons

entity :url, <entity_type> Return the URL for the entity `$params['entity']`. Note : Generally it is better to override the `getUrl()` method of ElggEntity. This hook should be used when it's not possible to subclass (like if you want to extend a bundled plugin without overriding many views).

to :object, <entity_type|metadata|annotation|relationship|river_item> Converts the entity `$params['entity']` to a StdClass object. This is used mostly for exporting entity properties for portable data formats like JSON and XML.

extender :url, <annotation|metadata> Return the URL for the annotation or metadatum `$params['extender']`.

file :icon :url, override Override a file icon URL.

is_member, group Return boolean for if the user `$params['user']` is a member of the group `$params['group']`.

entity :annotate, <entity_type> Triggered in `elgg_view_entity_annotations()`, which is called by `elgg_view_entity()`. Can be used to add annotations to all full entity views.

usersetting, plugin Filter user settings for plugins. `$params` contains :
— `user` - An ElggUser instance
— `plugin` - An ElggPlugin instance
— `plugin_id` - The plugin ID
— `name` - The name of the setting
— `value` - The value to set

setting, plugin Filter plugin settings. `$params` contains :
— `plugin` - An ElggPlugin instance

— `plugin_id` - The plugin ID
— `name` - The name of the setting
— `value` - The value to set

relationship :url, <relationship_name> Filter the URL for the relationship object `$params['relationship']`.

profile :fields, group Filter an array of profile fields. The result should be returned as an array in the format `name => input view name`. For example :

```
array(
        'about' => 'longtext'
);
```

profile :fields, profile Filter an array of profile fields. The result should be returned as an array in the format `name => input view name`. For example :

```
array(
        'about' => 'longtext'
);
```

widget_settings, <widget_handler> Triggered when saving a widget settings `$params['params']` for widget `$params['widget']`. If handling saving the settings, the handler should return true to prevent the default code from running.

handlers, widgets Triggered when a list of available widgets is needed. Plugins can conditionally add or remove widgets from this list or modify attributes of existing widgets like `context` or `multiple`.

get_list, default_widgets Filters a list of default widgets to add for newly registered users. The list is an array of arrays in the format :

```
array(
        'event' => $event,
        'entity_type' => $entity_type,
        'entity_subtype' => $entity_subtype,
        'widget_context' => $widget_context
)
```

public_pages, walled_garden Filters a list of URLs (paths) that can be seen by logged out users in a walled garden mode. Handlers must return an array of regex strings that will allow access if matched. Please note that system public routes are passed as the default value to the hook, and plugins must take care to not accidentally override these values.

The `$params` array contains :
— `url` - URL of the page being tested for public accessibility

volatile, metadata Triggered when exporting an entity through the export handler. This is rare. This allows handler to handle any volatile (non-persisted) metadata on the entity. It's preferred to use the `to:object, <type>` hook.

maintenance :allow, url

Return boolean if the URL `$params['current_url']` and the path `$params['current_path']` is allowed during maintenance mode.

robots.txt, site Filter the robots.txt values for `$params['site']`.

config, amd Filter the AMD config for the requirejs library.

Plugins

Embed

embed_get_items, <active_section>

embed_get_sections, all

embed_get_upload_sections, all

Groups

profile_buttons, group Filters buttons (`ElggMenuItem` instances) to be registered in the title menu of the group profile page

tool_options, group Filters a collection of tools available within a specific group :

The `$return` is `\Elgg\Collections\Collection<\Elgg\Groups\Tool>`, a collection of group tools.

The `$params` array contains :
— `entity` - `\ElggGroup`

HTMLawed

allowed_styles, htmlawed Filter the HTMLawed allowed style array.

config, htmlawed Filter the HTMLawed `$config` array.

spec, htmlawed Filter the HTMLawed `$spec` string (default empty).

Likes

likes :is_likable, <type> :<subtype> This is called to set the default permissions for whether to display/allow likes on an entity of type `<type>` and subtype `<subtype>`.

Note : The callback `'Elgg\Values::getTrue'` is a useful handler for this hook.

Members

members :list, <page_segment> To handle the page `/members/$page_segment`, register for this hook and return the HTML of the list.

members :config, tabs This hook is used to assemble an array of tabs to be passed to the navigation/tabs view for the members pages.

Reported Content

reportedcontent :add, system Triggered after adding the reported content object `$params['report']`. Return false to delete report.

reportedcontent :archive, system Triggered before archiving the reported content object `$params['report']`. Return false to prevent archiving.

reportedcontent :delete, system Triggered before deleting the reported content object `$params['report']`. Return false to prevent deleting.

Web Services

rest, init Triggered by the web services rest handler. Plugins can set up their own authentication handlers, then return `true` to prevent the default handlers from being registered.

rest :output, <method_name> Filter the result (and subsequently the output) of the API method

3.3.17 Internationalization

Make your UI translatable into many different languages.

If you'd like to contribute translations to Elgg, see the contributors" guide.

The default language is `en` for English. Elgg uses a fallback system for languages :

1. The language of the user
2. The site language
3. English

Overview

Translations are stored in PHP files in the `/languages` directory of your plugin. Each file corresponds to a language. The format is `/languages/{language-code}.php` where `{language-code}` is the ISO 639-1 short code for the language. For example :

```
<?php // mod/example/languages/en.php

return [
        'example:text' => 'Some example text',
];
```

To override an existing translation, include it in your plugin's language file, and make sure your plugin is ordered later on the Admin > Plugins page :

```
<?php // mod/better_example/languages/en.php

return [
        'example:text' => 'Some better text!',
];
```

Note : Unless you are overriding core's or another plugin's language strings, it is good practice for the language keys to start with your plugin name. For example : `yourplugin:success`, `yourplugin:title`, etc. This helps avoid conflicts with other language keys.

Server-side API

`elgg_echo($key, $args, $language)`

Output the translation of the key in the current language.

Example :

```
echo elgg_echo('example:text');
```

It also supports variable replacement using `vsprintf` syntax :

```
// 'welcome' => 'Welcome to %s, %s!'
echo elgg_echo('welcome', [
        elgg_get_config('sitename'),
        elgg_get_logged_in_user_entity()->getDisplayName(),
]);
```

To force which language should be used for translation, set the third parameter :

```
echo elgg_echo('welcome', [], $user->language);
```

To first test whether `elgg_echo()` can find a translation :

```
$key = 'key:that:might:not:exist';
if (!elgg_language_key_exists($key)) {
        $key = 'fallback:key';
}

echo elgg_echo($key);
```

Note : Some APIs allow creating translations for new keys. Translators should always include an English translation as a fallback. This makes `elgg_language_key_exists($key)` a reliable way to predict whether `elgg_echo($key)` will succeed.

Javascript API

`elgg.echo(key, args)`

This function is like `elgg_echo` in PHP.

Client-side translations are loaded asynchronously. Ensure translations are available by requiring the « elgg » AMD module :

```
define(function(require) {
        var elgg = require("elgg");

        alert(elgg.echo('my_key'));
});
```

Translations are also available after the `init, system` JavaScript event.

3.3.18 JavaScript

Contents

AMD

Developers should use the AMD (Asynchronous Module Definition) standard for writing JavaScript code in Elgg.

Here we'll describe making and executing AMD modules. The RequireJS documentation for defining modules may also be of use.

Executing a module in the current page

Telling Elgg to load an existing module in the current page is easy :

```
<?php
elgg_require_js("myplugin/say_hello");
```

On the client-side, this will asynchronously load the module, load any dependencies, and execute the module's definition function, if it has one.

Defining the Module

Here we define a basic module that alters the page, by passing a « definition function » to `define()` :

```
// in views/default/myplugin/say_hello.js

define(function(require) {
    var elgg = require("elgg");
    var $ = require("jquery");

    $('body').append(elgg.echo('hello_world'));
});
```

The module's name is determined by the view name, which here is `myplugin/say_hello.js`. We strip the `.js` extension, leaving `myplugin/say_hello`.

Avertissement : The definition function **must** have one argument named `require`.

Making modules dependent on other modules

Below we refactor a bit so that the module depends on a new `myplugin/hello` module to provide the greeting :

```
// in views/default/myplugin/hello.js

define(function(require) {
    var elgg = require("elgg");

    return elgg.echo('hello_world');
});
```

```
// in views/default/myplugin/say_hello.js

define(function(require) {
    var $ = require("jquery");
    var hello = require("myplugin/hello");

    $('body').append(hello);
});
```

Passing settings to modules

The `elgg.data` plugin hooks

The `elgg` module provides an object `elgg.data` which is populated from two server side hooks :

— **elgg.data, site** : This filters an associative array of site-specific data passed to the client and cached.

— **elgg.data, page** : This filters an associative array of uncached, page-specific data passed to the client.

Let's pass some data to a module :

```
<?php

function myplugin_config_site($hook, $type, $value, $params) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // this will be cached client-side
    $value['myplugin']['api'] = elgg_get_site_url() . 'myplugin-api';
    $value['myplugin']['key'] = 'none';
    return $value;
}

function myplugin_config_page($hook, $type, $value, $params) {
    $user = elgg_get_logged_in_user_entity();
    if ($user) {
        $value['myplugin']['key'] = $user->myplugin_api_key;
        return $value;
    }
}

elgg_register_plugin_hook_handler('elgg.data', 'site', 'myplugin_config_site');
elgg_register_plugin_hook_handler('elgg.data', 'page', 'myplugin_config_page');
```

```
define(function(require) {
    var elgg = require("elgg");

    var api = elgg.data.myplugin.api;
    var key = elgg.data.myplugin.key; // "none" or a user's key

    // ...
});
```

Note : In `elgg.data`, page data overrides site data. Also note `json_encode()` is used to copy data client-side, so the data must be JSON-encodable.

Making a config module

You can use a PHP-based module to pass values from the server. To make the module `myplugin/settings`, create the view file `views/default/myplugin/settings.js.php` (note the double extension `.js.php`).

```
<?php

// this will be cached client-side
$settings = [
    'api' => elgg_get_site_url() . 'myplugin-api',
    'key' => null,
];
?>
define(<?php echo json_encode($settings); ?>);
```

You must also manually register the view as an external resource :

```
<?php
// note the view name does not include ".php"
elgg_register_simplecache_view('myplugin/settings.js');
```

Note : The PHP view is cached, so you should treat the output as static (the same for all users) and avoid session-

specific logic.

Setting the URL of a module

You may have an AMD script outside your views you wish to make available as a module.

The best way to accomplish this is by configuring the path to the file using the `views.php` file in the root of your plugin :

```
<?php // views.php
return [
    'default' => [
        'underscore.js' => 'vendor/bower-asset/underscore/underscore.min.js',
    ],
];
```

If you've copied the script directly into your plugin instead of managing it with Composer, you can use something like this instead :

```
<?php // views.php
return [
    'default' => [
        'underscore.js' => __DIR__ . '/bower_components/underscore/underscore.min.js',
    ],
];
```

That's it ! Elgg will now load this file whenever the « underscore » module is requested.

Using traditional JS libraries as modules

It's possible to support JavaScript libraries that do not declare themselves as AMD modules (i.e. they declare global variables instead) if you shim them by setting `exports` and `deps` in `elgg_define_js` :

```
// set the path, define its dependencies, and what value it returns
elgg_define_js('jquery.form', [
    'deps' => ['jquery'],
    'exports' => 'jQuery.fn.ajaxForm',
]);
```

When this is requested client-side :

1. The jQuery module is loaded, as it's marked as a dependency.
2. `https://elgg.example.org/cache/125235034/views/default/jquery.form.js` is loaded and executed.
3. The value of `window.jQuery.fn.ajaxForm` is returned by the module.

Avertissement : Calls to `elgg_define_js()` must be in an `init, system` event handler.

Some things to note

1. Do not use `elgg.provide()` anymore nor other means to attach code to `elgg` or other global objects. Use modules.
2. Return the value of the module instead of adding to a global variable.
3. Static (.js,.css,etc.) files are automatically minified and cached by Elgg's simplecache system.
4. The configuration is also cached in simplecache, and should not rely on user-specific values like `get_language()`.

Booting your plugin

To add functionality to each page, or make sure your hook handlers are registered early enough, you may create a boot module for your plugin, with the name `boot/<plugin_id>`.

```
// in views/default/boot/example.js

define(function(require) {
    var elgg = require("elgg");
    var Plugin = require("elgg/Plugin");

    // plugin logic
    function my_init() { ... }

    return new Plugin({
        // executed in order of plugin priority
        init: function () {
            elgg.register_hook_handler("init", "system", my_init, 400);
        }
    });
});
```

When your plugin is active, this module will automatically be loaded on each page. Other modules can depend on `elgg/init` to make sure all boot modules are loaded.

Each boot module **must** return an instance of `elgg/Plugin`. The constructor must receive an object with a function in the `init` key. The `init` function will be called in the order of the plugin in Elgg's admin area.

Note : Though not strictly necessary, you may want to use the `init, system` event to control when your initialization code runs with respect to other modules.

Avertissement : A boot module **cannot** depend on the modules `elgg/init` or `elgg/ready`.

Modules provided with Elgg

Modules `jquery` and `jquery-ui`

You must depend on these modules to use `$` or `$.ui` methods. In the future Elgg may stop loading these by default.

Module `elgg`

`elgg.echo()`

Translate interface text

```
elgg.echo('example:text', ['arg1']);
```

`elgg.system_message()`

Display a status message to the user.

```
elgg.system_message(elgg.echo('success'));
```

`elgg.register_error()`

Display an error message to the user.

```
elgg.register_error(elgg.echo('error'));
```

`elgg.normalize_url()`

Normalize a URL relative to the elgg root :

```
// "http://localhost/elgg/blog"
elgg.normalize_url('/blog');
```

`elgg.forward()`

Redirect to a new page.

```
elgg.forward('/blog');
```

This function automatically normalizes the URL.

`elgg.parse_url()`

Parse a URL into its component parts :

```
// returns {
//   fragment: "fragment",
//   host: "community.elgg.org",
//   path: "/file.php",
//   query: "arg=val"
// }
elgg.parse_url('http://community.elgg.org/file.php?arg=val#fragment');
```

`elgg.get_page_owner_guid()`

Get the GUID of the current page's owner.

`elgg.register_hook_handler()`

Register a hook handler with the event system. For best results, do this in a plugin boot module.

```
// boot module: /views/default/boot/example.js
define(function (require) {
    var elgg = require('elgg');
    var Plugin = require('elgg/Plugin');

    elgg.register_hook_handler('foo', 'bar', function () { ... });

    return new Plugin();
});
```

`elgg.trigger_hook()`

Emit a hook event in the event system. For best results depend on the elgg/init module.

```
// old
value = elgg.trigger_hook('my_plugin:filter', 'value', {}, value);

define(function (require) {
    require('elgg/init');
    var elgg = require('elgg');

    value = elgg.trigger_hook('my_plugin:filter', 'value', {}, value);
});
```

`elgg.security.refreshToken()`

Force a refresh of all XSRF tokens on the page.

This is automatically called every 5 minutes by default.

The user will be warned if their session has expired.

`elgg.security.addToken()`

Add a security token to an object, URL, or query string :

```
// returns {
//   __elgg_token: "1468dc44c5b437f34423e2d55acfdd87",
//   __elgg_ts: 1328143779,
//   other: "data"
// }
elgg.security.addToken({'other': 'data'});

// returns: "action/add?__elgg_ts=1328144079&__elgg_
↪token=55fd9c2d7f5075d11e722358afd5fde2"
elgg.security.addToken("action/add");

// returns "?arg=val&__elgg_ts=1328144079&__elgg_
↪token=55fd9c2d7f5075d11e722358afd5fde2"
elgg.security.addToken("?arg=val");
```

`elgg.get_logged_in_user_entity()`

Returns the logged in user as an JS ElggUser object.

`elgg.get_logged_in_user_guid()`

Returns the logged in user's guid.

`elgg.is_logged_in()`

True if the user is logged in.

`elgg.is_admin_logged_in()`

True if the user is logged in and is an admin.

`elgg.config.get_language()`

Get the current page's language.

There are a number of configuration values set in the elgg object :

```
// The root of the website.
elgg.config.wwwroot;
// The default site language.
elgg.config.language;
// The current page's viewtype
elgg.config.viewtype;
// The Elgg version (YYYYMMDDXX).
elgg.config.version;
// The Elgg release (X.Y.Z).
elgg.config.release;
```

Module `elgg/Ajax`

See the *Ajax* page for details.

Module `elgg/init`

`elgg/init` loads and initializes all boot modules in priority order and triggers the [init, system] hook.

Require this module to make sure all plugins are ready.

Module `elgg/Plugin`

Used to create a *boot module*.

Module `elgg/ready`

`elgg/ready` loads and initializes all plugin boot modules in priority order.

Require this module to make sure all plugins are ready.

Module `elgg/spinner`

The `elgg/spinner` module can be used to create an Ajax loading indicator fixed to the top of the window.

```
define(function (require) {
   var spinner = require('elgg/spinner');

   elgg.action('friend/add', {
       beforeSend: spinner.start,
       complete: spinner.stop,
       success: function (json) {
           // ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }
    });
});
```

Note : The `elgg/Ajax` module uses the spinner by default.

Module `elgg/popup`

The `elgg/popup` module can be used to display an overlay positioned relatively to its anchor (trigger).

The `elgg/popup` module is loaded by default, and binding a popup module to an anchor is as simple as adding `rel="popup"` attribute and defining target module with a `href` (or `data-href`) attribute. Popup module positioning can be defined with `data-position` attribute of the trigger element.

```
echo elgg_format_element('div', [
    'class' => 'elgg-module-popup hidden',
    'id' => 'popup-module',
], 'Popup module content');

// Simple anchor
echo elgg_view('output/url', [
    'href' => '#popup-module',
    'text' => 'Show popup',
    'rel' => 'popup',
]);

// Button with custom positioning of the popup
echo elgg_format_element('button', [
    'rel' => 'popup',
    'class' => 'elgg-button elgg-button-submit',
    'text' => 'Show popup',
    'data-href' => '#popup-module',
    'data-position' => json_encode([
        'my' => 'center bottom',
        'at' => 'center top',
    ]),
]);
```

The `elgg/popup` module allows you to build out more complex UI/UX elements. You can open and close popup modules programmatically :

```
define(function(require) {
    var $ = require('jquery');
    $(document).on('click', '.elgg-button-popup', function(e) {

        e.preventDefault();

        var $trigger = $(this);
        var $target = $('#my-target');
        var $close = $target.find('.close');

        require(['elgg/popup'], function(popup) {
            popup.open($trigger, $target, {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'collision': 'fit none'
        });

        $close.on('click', popup.close);
    });
});
```

You can use `getOptions, ui.popup` plugin hook to manipulate the position of the popup before it has been opened. You can use jQuery `open` and `close` events to manipulate popup module after it has been opened or closed.

```
define(function(require) {

   var elgg = require('elgg');
   var $ = require('jquery');

   $('#my-target').on('open', function() {
      var $module = $(this);
      var $trigger = $module.data('trigger');

      elgg.ajax('ajax/view/my_module', {
         beforeSend: function() {
            $trigger.hide();
            $module.html('').addClass('elgg-ajax-loader');
         },
         success: function(output) {
            $module.removeClass('elgg-ajax-loader').html(output);
         }
      });
   }).on('close', function() {
      var $trigger = $(this).data('trigger');
      $trigger.show();
   });
});
```

Open popup modules will always contain the following data that can be accessed via `$.data()` :

— `trigger` - jQuery element used to trigger the popup module to open

— `position` - An object defining popup module position that was passed to `$.position()`

By default, `target` element will be appended to `$('body')` thus altering DOM hierarchy. If you need to preserve the DOM position of the popup module, you can add `.elgg-popup-inline` class to your trigger.

Module `elgg/widgets`

Plugins that load a widget layout via Ajax should initialize via this module :

```
require(['elgg/widgets'], function (widgets) {
    widgets.init();
});
```

Module `elgg/lightbox`

Elgg is distributed with the Colorbox jQuery library. Please go to http://www.jacklmoore.com/colorbox for more information on the options of this lightbox.

Use the following classes to bind your anchor elements to a lightbox :

- `elgg-lightbox` - loads an HTML resource
- `elgg-lightbox-photo` - loads an image resource (should be used to avoid displaying raw image bytes instead of an `img` tag)
- `elgg-lightbox-inline` - displays an inline HTML element in a lightbox
- `elgg-lightbox-iframe` - loads a resource in an `iframe`

You may apply colorbox options to an individual `elgg-lightbox` element by setting the attribute `data-colorbox-opts` to a JSON settings object.

```
echo elgg_view('output/url', [
   'text' => 'Open lightbox',
   'href' => 'ajax/view/my_view',
   'class' => 'elgg-lightbox',
   'data-colorbox-opts' => json_encode([
      'width' => '300px',
   ])
]);
```

Use `"getOptions", "ui.lightbox"` plugin hook to filter options passed to `$.colorbox()` whenever a lightbox is opened. Note that the hook handler should depend on `elgg/init` AMD module.

`elgg/lightbox` AMD module should be used to open and close the lightbox programmatically :

```
define(function(require) {
   var lightbox = require('elgg/lightbox');
   var spinner = require('elgg/spinner');

   lightbox.open({
      html: '<p>Hello world!</p>',
      onClosed: function() {
         lightbox.open({
            onLoad: spinner.start,
            onComplete: spinner.stop,
            photo: true,
            href: 'https://elgg.org/cache/1457904417/default/community_theme/graphics/
↪logo.png',
         });
      }
   });
});
```

To support gallery sets (via `rel` attribute), you need to bind colorbox directly to a specific selector (note that this will ignore `data-colorbox-opts` on all elements in a set) :

```
require(['elgg/lightbox'], function(lightbox) {
   var options = {
      photo: true,
      width: 500
   };
   lightbox.bind('a[rel="my-gallery"]', options, false); // 3rd attribute ensures␣
↪binding is done without proxies
});
```

You can also resize the lightbox programmatically if needed :

```
define(function(require) {
   var lightbox = require('elgg/lightbox');

   lightbox.resize({
      width: '300px'
   });
});
```

Module `elgg/ckeditor`

This module can be used to add WYSIWYG editor to a textarea (requires `ckeditor` plugin to be enabled). Note that WYSIWYG will be automatically attached to all instances of `.elgg-input-longtext`.

```
require(['elgg/ckeditor'], function (elggCKEditor) {
   elggCKEditor.bind('#my-text-area');

   // Toggle CKEditor
   elggCKEditor.toggle('#my-text-area');

   // Focus on CKEditor input
   elggCKEditor.focus('#my-text-area');
   // or
   $('#my-text-area').trigger('focus');

   // Reset CKEditor input
   elggCKEditor.reset('#my-text-area');
   // or
   $('#my-text-area').trigger('reset');

});
```

Inline tabs component

Inline tabs component fires an `open` event whenever a tabs is open and, in case of ajax tabs, finished loading :

```
// Add custom animation to tab content
require(['jquery', 'elgg/ready'], function($) {
        $(document).on('open', '.theme-sandbox-tab-callback', function() {
                $(this).find('a').text('Clicked!');
                $(this).data('target').hide().show('slide', {
                        duration: 2000,
                        direction: 'right',
                        complete: function() {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                                    alert('Thank you for clicking. We hope you enjoyed␣
↪the show!');
                                    $(this).css('display', ''); // .show() adds display␣
↪property
                            }
                    });
            });
});
```

Traditional scripts

Although we highly recommend using AMD modules, you can register scripts with `elgg_register_js` :

```
elgg_register_js('jquery', $cdnjs_url);
```

This will override any URLs previously registered under this name.

Load a library on the current page with `elgg_load_js` :

```
elgg_load_js('jquery');
```

This will load the library in the page footer. You must use the `require()` function to depend on modules like `elgg` and `jquery`.

Avertissement :

Using inline scripts is NOT SUPPORTED because :

- They are not testable (maintainability)
- They are not cacheable (performance)
- They prevent use of Content-Security-Policy (security)
- They prevent scripts from being loaded with `defer` or `async` (performance)

Inline scripts in core or bundled plugins are considered legacy bugs.

Hooks

The JS engine has a hooks system similar to the PHP engine's plugin hooks : hooks are triggered and plugins can register functions to react or alter information. There is no concept of Elgg events in the JS engine ; everything in the JS engine is implemented as a hook.

Registering hook handlers

Handler functions are registered using `elgg.register_hook_handler()`. Multiple handlers can be registered for the same hook.

The following example registers the `handleFoo` function for the `foo, bar` hook.

```
define(function (require) {
    var elgg = require('elgg');
    var Plugin = require('elgg/Plugin');

    function handleFoo(hook, type, params, value) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // do something
    }

    elgg.register_hook_handler('foo', 'bar', handleFoo);

    return new Plugin();
});
```

The handler function

The handler will receive 4 arguments :

- **hook** - The hook name
- **type** - The hook type
- **params** - An object or set of parameters specific to the hook
- **value** - The current value

The `value` will be passed through each hook. Depending on the hook, callbacks can simply react or alter data.

Triggering custom hooks

Plugins can trigger their own hooks :

```
define(function(require) {
    require('elgg/init');
    var elgg = require('elgg');

    elgg.trigger_hook('name', 'type', {params}, "value");
});
```

Note : Be aware of timing. If you don't depend on elgg/init, other plugins may not have had a chance to register their handlers.

Available hooks

init, system Plugins should register their init functions for this hook. It is fired after Elgg's JS is loaded and all plugin boot modules have been initialized. Depend on the `elgg/init` module to be sure this has completed.

ready, system This hook is fired when the system has fully booted (after init). Depend on the `elgg/ready` module to be sure this has completed.

getOptions, ui.popup This hook is fired for pop up displays (`"rel"="popup"`) and allows for customized placement options.

getOptions, ui.lightbox This hook can be used to filter options passed to `$.colorbox()`

config, ckeditor This filters the CKEditor config object. Register for this hook in a plugin boot module. The defaults can be seen in the module `elgg/ckeditor/config`.

prepare, ckeditor This hook can be used to decorate `CKEDITOR` global. You can use this hook to register new CKEditor plugins and add event bindings.

ajax_request_data, * This filters request data sent by the `elgg/Ajax` module. See *Ajax* for details. The hook must check if the data is a plain object or an instanceof `FormData` to piggyback the values using correct API.

ajax_response_data, * This filters the response data returned to users of the `elgg/Ajax` module. See *Ajax* for details.

insert, editor This hook is triggered by the embed plugin and can be used to filter content before it is inserted into the textarea. This hook can also be used by WYSIWYG editors to insert content using their own API (in this case the handler should return `false`). See ckeditor plugin for an example.

Third-party assets

We recommend managing third-party scripts and styles with Composer. Elgg's composer.json is configured to install dependencies from the **Bower** or **Yarn** package repositories using Composer command-line tool. Core configuration installs the assets from Asset Packagist (a repository managed by the Yii community).

Alternatively, you can install `fxp/composer-asset-plugin` globally to achieve the same results, but the installation and update takes much longer.

For example, to include jQuery, you could run the following Composer commands :

```
composer require bower-asset/jquery:~2.0
```

If you are using a starter-project, or pulling in Elgg as a composer dependency via a custom composer project, update your `composer.json` with the following configuration :

```
{
    "repositories": [
        {
            "type": "composer",
            "url": "https://asset-packagist.org"
        }
    ],
        "config": {
        "fxp-asset": {
            "enabled": false
        }
    },
}
```

You can find additional information at Asset Packagist website.

3.3.19 Menus

Elgg contains helper code to build menus throughout the site.

Every single menu requires a name, as does every single menu item. These are required in order to allow easy overriding and manipulation, as well as to provide hooks for theming.

Contents

— *JavaScript*

Basic usage

Basic functionalities can be achieved through these two functions :
- elgg_register_menu_item() to add an item to a menu
- elgg_unregister_menu_item() to remove an item from a menu

You normally want to call them from your plugin's init function.

Examples

```
// Add a new menu item to the site main menu
elgg_register_menu_item('site', array(
        'name' => 'itemname',
        'text' => 'This is text of the item',
        'href' => '/item/url',
));
```

```
// Remove the "Elgg" logo from the topbar menu
elgg_unregister_menu_item('topbar', 'elgg_logo');
```

Admin menu

You can also register *page* menu items to the admin backend menu. When registering for the admin menu you can set the context of the menu items to *admin* so the menu items only show in the *admin* context. There are 3 default sections to add your menu items to.
- *administer* for daily tasks, usermanagement and other actionable tasks
- *configure* for settings, configuration and utilities that configure stuff
- *information* for statistics, overview of information or status

Advanced usage

You can get more control over menus by using *plugin hooks* and the public methods provided by the ElggMenuItem class.

There are two hooks that can be used to modify a menu :
- `'register', 'menu:<menu name>'` to add or modify items (especially in dynamic menus)
- `'prepare', 'menu:<menu name>'` to modify the structure of the menu before it is displayed

When you register a plugin hook handler, replace the `<menu name>` part with the internal name of the menu.

The third parameter passed into a menu handler contains all the menu items that have been registered so far by Elgg core and other enabled plugins. In the handler we can loop through the menu items and use the class methods to interact with the properties of the menu item.

Examples

Example 1 : Change the URL for menu item called « albums » in the `owner_block` menu :

```
/**
 * Initialize the plugin
 */
function my_plugin_init() {
        // Register a plugin hook handler for the owner_block menu
        elgg_register_plugin_hook_handler('register', 'menu:owner_block', 'my_owner_
↪block_menu_handler');
}

/**
 * Change the URL of the "Albums" menu item in the owner_block menu
 */
function my_owner_block_menu_handler($hook, $type, $items, $params) {
        $owner = $params['entity'];

        // Owner can be either user or a group, so we
        // need to take both URLs into consideration:
        switch ($owner->getType()) {
                case 'user':
                        $url = "album/owner/{$owner->guid}";
                        break;
                case 'group':
                        $url = "album/group/{$owner->guid}";
                        break;
        }

        if ($items->has('albums')) {
                $items->get('albums')->setURL($url);
        }

        return $items;
}
```

Example 2 : Modify the `entity` menu for the `ElggBlog` objects

— Remove the thumb icon

— Change the « Edit » text into a custom icon

```
/**
 * Initialize the plugin
 */
function my_plugin_init() {
        // Register a plugin hook handler for the entity menu
        elgg_register_plugin_hook_handler('register', 'menu:entity', 'my_entity_menu_
↪handler');
}

/**
 * Customize the entity menu for ElggBlog objects
 */
function my_entity_menu_handler($hook, $type, $items, $params) {
        // The entity can be found from the $params parameter
        $entity = $params['entity'];
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // We want to modify only the ElggBlog objects, so we
        // return immediately if the entity is something else
        if (!$entity instanceof ElggBlog) {
                return $menu;
        }

        $items->remove('likes');

        if ($items->has('edit')) {
                $items->get('edit')->setText('Modify');
                $items->get('edit')->icon = 'pencil';
        }

        return $items;
}
```

Creating a new menu

Elgg provides multiple different menus by default. Sometimes you may however need some menu items that don't fit in any of the existing menus. If this is the case, you can create your very own menu with the elgg_view_menu() function. You must call the function from the view, where you want to menu to be displayed.

Example : Display a menu called « my_menu » that displays it's menu items in alphapetical order :

```
// in a resource view
echo elgg_view_menu('my_menu', array('sort_by' => 'text'));
```

You can now add new items to the menu like this :

```
// in plugin init
elgg_register_menu_item('my_menu', array(
        'name' => 'my_page',
        'href' => 'path/to/my_page',
        'text' => elgg_echo('my_plugin:my_page'),
));
```

Furthermore it is now possible to modify the menu using the hooks `'register', 'menu:my_menu'` and `'prepare', 'menu:my_menu'`.

Child Dropdown Menus

Child menus can be configured using `child_menu` factory option on the parent item.

`child_menu` options array accepts `display` parameter, which can be used to set the child menu to open as `dropdown` or be displayed via `toggle`. All other key value pairs will be passed as attributes to the `ul` element.

```
// Register a parent menu item that has a dropdown submenu
elgg_register_menu_item('my_menu', array(
        'name' => 'parent_item',
        'href' => '#',
        'text' => 'Show dropdown menu',
        'child_menu' => [
                'display' => 'dropdown',
                'class' => 'elgg-additional-child-menu-class',
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'data-position' => json_encode([
                        'at' => 'right bottom',
                        'my' => 'right top',
                        'collision' => 'fit fit',
                ]),
                'data-foo' => 'bar',
                'id' => 'dropdown-menu-id',
        ],
));

// Register a parent menu item that has a hidden submenu toggled when item is clicked
elgg_register_menu_item('my_menu', array(
        'name' => 'parent_item',
        'href' => '#',
        'text' => 'Show submenu',
        'child_menu' => [
                'display' => 'dropdown',
                'class' => 'elgg-additional-submenu-class',
                'data-toggle-duration' => 'medium',
                'data-foo' => 'bar2',
                'id' => 'submenu-id',
        ],
));
```

Theming

The menu name, section names, and item names are all embedded into the HTML as CSS classes (normalized to contain only hyphens, rather that underscores or colons). This increases the size of the markup slightly but provides themers with a high degree of control and flexibility when styling the site.

Example : The following would be the output of the `foo` menu with sections `alt` and `default` containing items `baz` and `bar` respectively.

```
<ul class="elgg-menu elgg-menu-foo elgg-menu-foo-alt">
        <li class="elgg-menu-item elgg-menu-item-baz"></li>
</ul>
<ul class="elgg-menu elgg-menu-foo elgg-menu-foo-default">
        <li class="elgg-menu-item elgg-menu-item-bar"></li>
</ul>
```

Toggling Menu Items

There are situations where you wish to toggle menu items that are actions that are the opposite of each other and ajaxify them. E.g. like/unlike, friend/unfriend, ban/unban, etc. Elgg has built-in support for this kind of actions. When you register a menu item you can provide a name of the menu item (in the same menu) that should be toggled. An ajax call will be made using the href of the menu item.

```
elgg_register_menu_item('my_menu', [
        'name' => 'like',
        'data-toggle' => 'unlike',
        'href' => 'action/like',
        'text' => elgg_echo('like'),
]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
elgg_register_menu_item('my_menu', [
        'name' => 'unlike',
        'data-toggle' => 'like',
        'href' => 'action/unlike',
        'text' => elgg_echo('unlike'),
]);
```

Note : The menu items are optimistically toggled. This means the menu items are toggled before the actions finish. If the actions fail, the menu items will be toggled back.

JavaScript

It is common that menu items rely on JavaScript. You can bind client-side events to menu items by placing your JavaScript into AMD module and defining the requirement during the registration.

```
elgg_register_menu_item('my_menu', array(
        'name' => 'hide_on_click',
        'href' => '#',
        'text' => elgg_echo('hide:on:click'),
        'item_class' => '.hide-on-click',
        'deps' => ['navigation/menu/item/hide_on_click'],
));
```

```
// in navigation/menu/item/hide_on_click.js
define(function(require) {
    var $ = require('jquery');

    $(document).on('click', '.hide-on-click', function(e) {
        e.preventDefault();
        $(this).hide();
    });
});
```

3.3.20 Notifications

There are two ways to send notifications in Elgg :

— Instant notifications
— Event-based notifications send using a notifications queue

Contents

Instant notifications

The generic method to send a notification to a user is via the function notify_user(). It is normally used when we want to notify only a single user. Notification like this might for example inform that someone has liked or commented the user's post.

The function usually gets called in an *action* file.

Example :

In this example a user (`$user`) is triggering an action to rate a post created by another user (`$owner`). After saving the rating (`ElggAnnotation $rating`) to database, we could use the following code to send a notification about the new rating to the owner.

```
// Subject of the notification
$subject = elgg_echo('ratings:notification:subject', array(), $owner->language);

// Summary of the notification
$summary = elgg_echo('ratings:notification:summary', array($user->getDisplayName()),
↪$owner->language);

// Body of the notification message
$body = elgg_echo('ratings:notification:body', array(
        $user->getDisplayName(),
        $owner->getDisplayName(),
        $rating->getValue() // A value between 1-5
), $owner->language);

$params = array(
        'object' => $rating,
        'action' => 'create',
        'summary' => $summary
);

// Send the notification
notify_user($owner->guid, $user->guid, $subject, $body, $params);
```

Note : The language used by the recipient isn't necessarily the same as the language of the person who triggers the notification. Therefore you must always remember to pass the recipient's language as the third parameter to `elgg_echo()`.

Note : The `'summary'` parameter is meant for notification plugins that only want to display a short message instead of both the subject and the body. Therefore the summary should be terse but still contain all necessary information.

Enqueued notifications

On large sites there may be many users who have subscribed to receive notifications about a particular event. Sending notifications immediately when a user triggers such an event might remarkably slow down page loading speed. This is why sending of such notifications shoud be left for Elgg's notification queue.

New notification events can be registered with the elgg_register_notification_event() function. Notifications about registered events will be sent automatically to all subscribed users.

This is the workflow of the notifications system :

1. **Someone does an action that triggers an event within Elgg**
   - The action can be `create`, `update` or `delete`
   - The target of the action can be any instance of the `ElggEntity` class (e.g. a Blog post)
2. The notifications system saves this event into a notifications queue in the database
3. When the pluging hook handler for the one-minute interval gets triggered, the event is taken from the queue and it gets processed
4. **Subscriptions are fetched for the user who triggered the event**
   - By default this includes all the users who have enabled any notification method for the user at `www.site.com/notifications/personal/<username>`
5. Plugins are allowed to alter the subscriptions using the `[get, subscriptions]` hook
6. Plugins are allowed to terminate notifications queue processing with the `[send:before, notifications]` hook
7. Plugins are allowed to alter the notification parameters with the `[prepare, notification]` hook
8. Plugins are allowed to alter the notification subject/message/summary with the `[prepare, notification:<action>:<type>:<subtype>]` hook
9. Plugins are allowed to format notification subject/message/summary for individual delivery methods with `[format, notification:<method>]` hook
10. **Notifications are sent to each subscriber using the methods they have chosen**
    - Plugins can take over or prevent sending of each individual notification with the `[send, notification:<method>]` hook
11. The `[send:after, notifications]` hook is triggered for the event after all notifications have been sent

Example

Tell Elgg to send notifications when a new object of subtype « photo » is created :

```
/**
 * Initialize the photos plugin
 */
function photos_init() {
        elgg_register_notification_event('object', 'photo', array('create'));
}
```

Note : In order to send the event-based notifications you must have the one-minute *CRON* interval configured.

Contents of the notification message can be defined with the `'prepare', 'notification:[action]:[type]:[subtype]'` hook.

Example

Tell Elgg to use the function `photos_prepare_notification()` to format the contents of the notification when a new objects of subtype "photo" is created :

```
/**
 * Initialize the photos plugin
 */
function photos_init() {
    elgg_register_notification_event('object', 'photo', array('create'));
    elgg_register_plugin_hook_handler('prepare', 'notification:create:object:photo',
↪'photos_prepare_notification');
}

/**
 * Prepare a notification message about a new photo
 *
 * @param string                           $hook         Hook name
 * @param string                           $type         Hook type
 * @param Elgg_Notifications_Notification $notification The notification to prepare
 * @param array                            $params       Hook parameters
 * @return Elgg_Notifications_Notification
 */
function photos_prepare_notification($hook, $type, $notification, $params) {
    $entity = $params['event']->getObject();
    $owner = $params['event']->getActor();
    $recipient = $params['recipient'];
    $language = $params['language'];
    $method = $params['method'];

    // Title for the notification
    $notification->subject = elgg_echo('photos:notify:subject', [$entity->
↪getDisplayName()], $language);

    // Message body for the notification
    $notification->body = elgg_echo('photos:notify:body', array(
        $owner->getDisplayName(),
        $entity->getDisplayName(),
        $entity->getExcerpt(),
        $entity->getURL()
    ), $language);

    // Short summary about the notification
    $notification->summary = elgg_echo('photos:notify:summary', [$entity->
↪getDisplayName()], $language);

    return $notification;
}
```

Note : Make sure the notification will be in the correct language by passing the reciepient's language into the `elgg_echo()` function.

Registering a new notification method

By default Elgg has two notification methods : email and the bundled site_notifications plugin. You can register a new notification method with the elgg_register_notification_method() function.

Example :

Register a handler that will send the notifications via SMS.

```
/**
 * Initialize the plugin
 */
function sms_notifications_init () {
        elgg_register_notification_method('sms');
}
```

After registering the new method, it will appear to the notification settings page at `www.example.com/notifications/personal/[username]`.

Sending the notifications using your own method

Besides registering the notification method, you also need to register a handler that takes care of actually sending the SMS notifications. This happens with the `'send', 'notification:[method]'` hook.

Example :

```
/**
 * Initialize the plugin
 */
function sms_notifications_init () {
        elgg_register_notification_method('sms');
        elgg_register_plugin_hook_handler('send', 'notification:sms', 'sms_
↪notifications_send');
}

/**
 * Send an SMS notification
 *
 * @param string $hook   Hook name
 * @param string $type   Hook type
 * @param bool   $result Has anyone sent a message yet?
 * @param array  $params Hook parameters
 * @return bool
 * @internal
 */
function sms_notifications_send($hook, $type, $result, $params) {
        /* @var Elgg_Notifications_Notification $message */
        $message = $params['notification'];

        $recipient = $message->getRecipient();

        if (!$recipient || !$recipient->mobile) {
                return false;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }

        // (A pseudo SMS API class)
        $sms = new SmsApi();

        return $sms->send($recipient->mobile, $message->body);
}
```

Subscriptions

In most cases Elgg core takes care of handling the subscriptions, so notification plugins don't usually have to alter them.

Subscriptions can however be :

- Added using the elgg_add_subscription() function
- Removed using the elgg_remove_subscription() function

It's possible to modify the recipients of a notification dynamically with the `'get', 'subscriptions'` hook.

Example :

```
/**
 * Initialize the plugin
 */
function discussion_init() {
        elgg_register_plugin_hook_handler('get', 'subscriptions', 'discussion_get_
↪subscriptions');
}

/**
 * Get subscriptions for group notifications
 *
 * @param string $hook          'get'
 * @param string $type          'subscriptions'
 * @param array  $subscriptions Array containing subscriptions in the form
 *                      <user guid> => array('email', 'site', etc.)
 * @param array  $params        Hook parameters
 * @return array
 */
function discussion_get_subscriptions($hook, $type, $subscriptions, $params) {
        $reply = $params['event']->getObject();

        if (!elgg_instanceof($reply, 'object', 'discussion_reply')) {
                return $subscriptions;
        }

        $group_guid = $reply->getContainerEntity()->container_guid;
        $group_subscribers = elgg_get_subscriptions_for_container($group_guid);

        return ($subscriptions + $group_subscribers);
}
```

E-mail attachments

`notify_user()` or enqueued notifications support attachments for e-mail notifications if provided in `$params`. To add one or more attachments add a key `attachments` in `$params` which is an array of the attachments. An attachment should be in one of the following formats :

- An `ElggFile` which points to an existing file
- An array with the file contents
- An array with a filepath

```
// this example is for notify_user()
$params['attachments'] = [];

// Example of an ElggFile attachment
$file = new \ElggFile();
$file->owner_guid = <some owner_guid>;
$file->setFilename('<some filename>');

$params['attachments'][] = $file;

// Example of array with content
$params['attachments'][] = [
        'content' => 'The file content',
        'filename' => 'test_file.txt',
        'type' => 'text/plain',
];

// Example of array with filepath
// 'filename' can be provided, if not basename() of filepath will be used
// 'type' can be provided, if not will try a best guess
$params['attachments'][] = [
        'filepath' => '<path to a valid file>',
];

notify_user($to_guid, $from_guid, $subject, $body, $params);
```

3.3.21 Page ownership

One recurring task of any plugin will be to determine the page ownership in order to decide which actions are allowed or not. Elgg has a number of functions related to page ownership and also offers plugin developers flexibility by letting the plugin handle page ownership requests as well. Determining the owner of a page can be determined with `elgg_get_page_owner_guid()`, which will return the GUID of the owner. Alternatively, `elgg_get_page_owner_entity()` will retrieve the whole page owner entity. If the page already knows who the page owner is, but the system doesn't, the page can set the page owner by passing the GUID to `elgg_set_page_owner_guid($guid)`.

Note : The page owner entity can be any `ElggEntity`. If you wish to only apply some setting in case of a user or a group make sure you check that you have the correct entity.

Custom page owner handlers

Plugin developers can create page owner handlers, which could be necessary in certain cases, for example when integrating third party functionality. The handler will be a function which will need to get registered with `elgg_register_plugin_hook_handler('page_owner', 'system', 'your_page_owner_function_name');` . The handler will only need to return a value (an integer GUID) when it knows for certain who the page owner is.

By default, the system uses `default_page_owner_handler()` to determine the page_owner from the following elements :

- The `username` URL parameter
- The `owner_guid` URL parameter
- The URL path

It then passes off to any page owner handlers defined using the *plugin hook*. If no page owner can be determined, the page owner is set to 0, which is the same as the logged out user.

3.3.22 Permissions Check

Avertissement : As stated in the page, this method works **only** for granting **write** access to entities. You **cannot** use this method to retrieve or view entities for which the user does not have read access.

Elgg provides a mechanism of overriding write permissions check through the *permissions_check plugin hook* . This is useful for allowing plugin write to all accessible entities regardless of access settings. Entities that are hidden, however, will still be unavailable to the plugin.

Hooking permissions_check

In your plugin, you must register the plugin hook for `permissions_check`.

```
elgg_register_plugin_hook_handler('permissions_check', 'all', 'myplugin_permissions_
↪check');
```

The override function

Now create the function that will be called by the permissions check hook. In this function we determine if the entity (in parameters) has write access. Since it is important to keep Elgg secure, write access should be given only after checking a variety of situations including page context, logged in user, etc. Note that this function can return 3 values : true if the entity has write access, false if the entity does not, and null if this plugin doesn't care and the security system should consult other plugins.

```
function myplugin_permissions_check($hook_name, $entity_type, $return_value,
↪$parameters) {
   $has_access = determine_access_somehow();

   if ($has_access === true) {
      return true;
   } else if ($has_access === false) {
      return false;
   }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return null;
}
```

Full Example

This is a full example using the context to determine if the entity has write access.

```
<?php

function myaccess_init() {
    // Register cron hook
    if (!elgg_get_plugin_setting('period', 'myaccess')) {
        elgg_set_plugin_setting('period', 'fiveminute', 'myaccess');
    }

    // override permissions for the myaccess context
    elgg_register_plugin_hook_handler('permissions_check', 'all', 'myaccess_
↪permissions_check');

    elgg_register_plugin_hook_handler('cron', elgg_get_plugin_setting('period',
↪'myaccess'), 'myaccess_cron');
}

/**
 * Hook for cron event.
 */
function myaccess_cron($event, $object_type, $object) {

    elgg_push_context('myaccess_cron');

    // returns all entities regardless of access permissions.
    // will NOT return hidden entities.
    $entities = get_entities();

    elgg_pop_context();
}

/**
 * Overrides default permissions for the myaccess context
 */
function myaccess_permissions_check($hook_name, $entity_type, $return_value,
↪$parameters) {
    if (elgg_in_context('myaccess_cron')) {
        return true;
    }

    return null;
}

// Initialise plugin
register_elgg_event_handler('init', 'system', 'myaccess_init');
?>
```

3.3.23 Plugins

Plugins must provide a `manifest.xml` file in the plugin root in order to be recognized by Elgg.

Contents

elgg-plugin.php

`elgg-plugin.php` is a static plugin configuration file. It is read by Elgg to configure various services, and must return an array if present. It should not be included by plugins and is not guaranteed to run at any particular time. Besides magic constants like `__DIR__`, its return value should not change. The currently supported sections are :

- `bootstrap` - defines a class used to bootstrap the plugin
- `entities` - defines entity types and classes, and optionally registers them for search
- `actions` - eliminates the need for calling `elgg_register_action()`
- `routes` - eliminates the need for calling `elgg_register_route()`
- `settings` - eliminates the need for setting default values on each call to `elgg_get_plugin_setting()`
- `user_settings` - eliminates the need for setting default values on each call to `elgg_get_plugin_user_setting()`
- `views` - allows plugins to alias vendor assets to a path within the Elgg's view system
- `widgets` - eliminates the need for calling `elgg_register_widget_type()`

```
return [
        // Bootstrap must implement \Elgg\PluginBootstrapInterface
        'bootstrap' => MyPluginBootstrap::class,

        'entities' => [
                [
                        // Register a new object subtype and tell Elgg to use a␣
↪specific class to instantiate it
                        'type' => 'object',
                        'subtype' => 'my_object_subtype',
                        'class' => MyObjectClass::class,

                        // Register this subtype for search
                        'searchable' => true,
                ],
        ],

        'actions' => [
                // Registers an action
                // By default, action is registered with 'logged_in' access
                // By default, Elgg will look for file in plugin's actions/␣
↪directory: actions/my_plugin/action.php
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'my_plugin/action/default' => [],

                'my_plugin/action/custom_access' => [
                        'access' => 'public', // supports 'public', 'logged_in',
↪'admin'
                ],

                // you can use action controllers instead of action files by setting␣
↪the controller parameters
                // controller must be a callable that receives \Elgg\Request as the␣
↪first and only argument
                // in example below, MyActionController::__invoke(\Elgg\Request
↪$request) will be called
                'my_plugin/action/controller' => [
                        'controller' => MyActionController::class,
                ],
        ],

        'routes' => [
                // routes can be associated with resource views or controllers
                'collection:object:my_object_subtype:all' => [
                        'path' => '/my_stuff/all',
                        'resource' => 'my_stuff/all', // view file is in resources/my_
↪stuff/all
                ],

                // similar to actions, routes can be associated with a callable␣
↪controller that receives an instance of \Elgg\Request
                'collection:object:my_object_subtype:json' => [
                        'path' => '/my_stuff/json',
                        'controller' => JsonDumpController::class,
                ],

                // route definitions support other parameters, such as 'middleware',
↪'requirements', 'defaults'
                // see elgg_register_route() for all options
        ],

        'widgets' => [
                // register a new widget
                // corresponds to a view in widgets/my_stuff/content
                'my_stuff' => [
                        'description' => elgg_echo('widgets:my_stuff'),
                        'context' => ['profile', 'dashboard'],
                ],
        ],

        'settings' => [
                'plugin_setting_name' => 'plugin_setting_value',
        ],

        'user_settings' => [
                'user_setting_name' => 'user_setting_value',
        ],

        // this is identical to using views.php in Elgg 2.x series
        'views' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'default' => [
                        'cool_lib/' => __DIR__ . '/vendors/cool_lib/dist/',
                ],
        ],
];
```

Bootstrap class

As of Elgg 3.0 the recommended way to bootstrap you plugin is to use a bootstrap class. This class must implement the `\Elgg\PluginBootstrapInterface` interface. You can register you bootstrap class in the `elgg-plugin.php`.

The bootstrap interface defines several function to be implemented which are called during different events in the system booting process.

Voir aussi :

For more information about the different functions defined in the `\Elgg\PluginBootstrapInterface` please read *Plugin bootstrap*

start.php

The `start.php` file bootstraps plugin by registering event listeners and plugin hooks.

It is advised that plugins return an instance of Closure from the `start.php` instead of placing registrations in the root of the file. This allows for consistency in Application bootstrapping, especially for testing purposes.

```
function my_plugin_does_something_else() {
    // Some procedural code that you want to run before any events are fired
}

function my_plugin_init() {
    // Your plugin's initialization logic
}

function my_plugin_rewrite_hook() {
    // Path rewrite hook
}

return function() {
    my_plugin_do_something_else();
    elgg_register_event_handler('init', 'system', 'my_plugin_init');
    elgg_register_plugin_hook_handler('route:rewrite', 'proifle', 'my_plugin_rewrite_
↪hook');
}
```

elgg-services.php

Plugins can attach their services to Elgg's public DI container by providing PHP-DI definitions in `elgg-services.php` in the root of the plugin directory.

This file must return an array of PHP-DI definitions. Services will by available via `elgg()`.

```
return [
   PluginService::class => \DI\object()->constructor(\DI\
↪get(DependencyService::class)),
];
```

Plugins can then use PHP-DI API to autowire and call the service :

```
$service = elgg()->get(PluginService::class);
```

See PHP-DI documentation for a comprehensive list of definition and invokation possibilities.

Syntax

Here's a trivial example configuring view locations via the `views` key :

```
return [
        'views' => [
                'default' => [
                        'file/icon/' => __DIR__ . '/graphics/icons',
                ],
        ],
];
```

activate.php, deactivate.php

The `activate.php` and `deactivate.php` files contain procedural code that will run upon plugin activation and deactivation. Use these files to perform one-time events such as registering a persistent admin notice, registering subtypes, or performing garbage collection when deactivated.

manifest.xml

Elgg plugins are required to have a `manifest.xml` file in the root of a plugin.

The `manifest.xml` file includes information about the plugin itself, requirements to run the plugin, and optional information including where to display the plugin in the admin area and what APIs the plugin provides.

Syntax

The manifest file is a standard XML file in UTF-8. Everything is a child of the `<plugin_manifest>` element.

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
```

The manifest syntax is as follows :

```
<name>value</name>
```

Many elements can contain children attributes :

```
<parent_name>
        <child_name>value</child_name>
        <child_name_2>value_2</child_name_2>
</parent_name>
```

Required Elements

All plugins are required to define the following elements in their manifest files :

- id - This has the name as the directory that the plugin uses.
- name - The display name of the plugin.
- author - The name of the author who wrote the plugin.
- version - The version of the plugin.
- description - A description of the what the plugin provides, its features, and other relevant information
- requires - Each plugin must specify the release of Elgg it was developed for. See the plugin Dependencies page for more information.

Available Elements

In addition to the require elements above, the follow elements are available to use :

- blurb - A short description of the plugin.
- category - The category of the plugin. It is recommended to follow the *Plugin coding guidelines* and use one of the defined categories. There can be multiple entries.
- conflicts - Specifies that the plugin conflicts with a certain system configuration.
- copyright - The plugin's copyright information.
- license - The plugin's license information.
- provides - Specifies that this plugin provides the same functionality as another Elgg plugin or a PHP extension.
- screenshot - Screenshots of the plugin. There can be multiple entries. See the advanced example for syntax.
- suggests - Parallels the requires system, but doesn't affect if the plugin can be enabled. Used to suggest other plugins that interact or build on the plugin.
- website - A link to the website for the plugin.

Voir aussi :

Plugin Dependencies

Simple Example

This manifest file is the bare minimum a plugin must have.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
        <name>Example Manifest</name>
        <author>Elgg</author>
        <version>1.0</version>
        <description>This is a simple example of a manifest file. In this example,␣
↪there are not screenshots, dependencies, or additional information about the plugin.
↪</description>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        <requires>
                <type>elgg_release</type>
                <version>1.9</version>
        </requires>
</plugin_manifest>
```

Advanced example

This example uses all of the available elements :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
        <name>Example Manifest</name>
        <author>Brett Profitt</author>
        <version>1.0</version>
        <blurb>This is an example manifest file.</blurb>
        <description>This is a simple example of a manifest file. In this example,␣
↪there are many options used, including screenshots, dependencies, and additional␣
↪information about the plugin.</description>
        <website>http://www.elgg.org/</website>
        <copyright>(C) Brett Profitt 2014</copyright>
        <license>GNU Public License version 2</license>

        <category>3rd_party_integration</category>

        <requires>
                <type>elgg_release</type>
                <version>1.9.1</version>
        </requires>

        <!-- The path is relative to the plugin's root. -->
        <screenshot>
                <description>Elgg profile.</description>
                <path>screenshots/profile.png</path>
        </screenshot>

        <provides>
                <type>plugin</type>
                <name>example_plugin</name>
                <version>1.5</version>
        </provides>

        <suggests>
                <type>plugin</type>
                <name>twitter</name>
                <version>1.0</version>
        </suggests>
</plugin_manifest>
```

composer.json

Since Elgg supports being installed as a Composer dependency, having your plugins also support Composer makes for easier installation by site administrators. In order to make your plugin compatible with Composer you need to at least have a `composer.json` file in the root of your plugin.

Here is an example of a `composer.json` file :

```
{
        "name": "company/example_plugin",
        "description": "Some description of the plugin",
        "type": "elgg-plugin",
        "keywords": ["elgg", "plugin"],
        "license": "GPL-2.0-only",
        "support": {
                "source": "URL to your code repository",
                "issues": "URL to your issue tracker"
        },
        "require": {
                "composer/installers": "^1.0.8"
        },
        "conflict": {
                "elgg/elgg": "<3.0"
        }
}
```

Read more about the `composer.json` format on the Composer website.

Important parts in the `composer.json` file are :

- `name` : the name of your plugin, keep this inline with the name of your plugin folder to ensure correct installation
- `type` : this will tell Composer where to install your plugin, ALWAYS keep this as `elgg-plugin`
- `require` : the `composer/installers` requirement is to make sure Composer knows where to install your plugin

As a suggestion, include a `conflict` rule with any Elgg version below your mininal required version, this will help prevent the accidental installation of your plugin on an incompatible Elgg version.

After adding a `composer.json` file to your plugin project, you need to register your project on Packagist in order for other people to be able to install your plugin.

Tests

It's encouraged to create PHPUnit test for your plugin. All tests should be located in `tests/phpunit/unit` for unit tests and `tests/phpunit/integration` for integration tests.

An easy example of adding test is the `ViewStackTest`, this will test that the views in your plugin are registered correctly and have no syntax errors. To add this test create a file `ViewStackTest.php` in the folder `tests/phpunit/unit/<YourNameSpace>/<YourPluginName>/` with the content :

```
namespace <YourNameSpace>\<YourPluginName>;

/**
 * @group ViewsService
 */
class ViewStackTest extends \Elgg\Plugins\ViewStackTest {

}
```

Note : If you wish to see a better example, look in any of the Elgg core plugins.

Voir aussi :

Écrire des tests

Related

Plugin skeleton

The following is the standard for plugin structure in Elgg as of Elgg 2.0.

Example Structure

The following is an example of a plugin with standard structure. For further explanation of this structure, see the details in the following sections. Your plugin may not need all the files listed

The following files for plugin `example` would go in `/mod/example/`

```
actions/
    example/
        action.php
        other_action.php
classes/
    VendorNamespace/
        ExampleClass.php
languages/
    en.php
vendors/
    example_3rd_party_lib/
views/
    default/
        example/
          component.css
          component.js
          component.png
        forms/
            example/
                action.php
                other_action.php
        object/
            example.php
            example/
                context1.php
                context2.php
        plugins/
            example/
                settings.php
                usersettings.php
        resources/
            example/
                all.css
                all.js
                all.php
```

(suite sur la page suivante)

(suite de la page précédente)

```
                owner.css
                owner.js
                owner.php
        widgets/
            example_widget/
                content.php
                edit.php
activate.php
deactivate.php
elgg-plugin.php
CHANGES.txt
COPYRIGHT.txt
INSTALL.txt
LICENSE.txt
manifest.xml
README.txt
start.php
composer.json
```

Required Files

Plugins **must** provide a `manifest.xml` file in the plugin root in order to be recognized by Elgg.

Therefore the following is the minimally compliant structure :

```
mod/example/
    manifest.xml
```

Actions

Plugins *should* place scripts for actions an `actions/` directory, and furthermore *should* use the name of the action to determine the location within that directory.

For example, the action `my/example/action` would go in `my_plugin/actions/my/example/action.php`. This makes it very obvious which script is associated with which action.

Similarly, the body of the form that submits to this action should be located in `forms/my/example/action.php`. Not only does this make the connection b/w action handler, form code, and action name obvious, but it allows you to use the `elgg_view_form()` function easily.

Text Files

Plugins *may* provide various *.txt as additional documentation for the plugin. These files **must** be in Markdown syntax and will generate links on the plugin management sections.

README.txt *should* provide additional information about the plugin of an unspecified nature

COPYRIGHT.txt If included, **must** provide an explanation of the plugin's copyright, besides what is included in `manifest.xml`

LICENSE.txt If included, **must** provide the text of the license that the plugin is released under.

INSTALL.txt If included, **must** provide additional instructions for installing the plugin if the process is sufficiently complicated (e.g. if it requires installing third party libraries on the host machine, or requires acquiring an API key from a third party).

CHANGES.txt If included, **must** provide a list of changes for their plugin, grouped by version number, with the most recent version at the top.

Plugins *may* include additional *.txt files besides these, but no interface is given for reading them.

Pages

To render full pages, plugins should use **resource views** (which have names beginning with `resources/`). This allows other plugins to easily replace functionality via the view system.

Note : The reason we encourage this structure is
— To form a logical relationship between urls and scripts, so that people examining the code can have an idea of what it does just by examining the structure.
— To clean up the root plugin directory, which historically has quickly gotten cluttered with the page handling scripts.

Classes

Elgg provides PSR-0 autoloading out of every active plugin's `classes/` directory.

You're encouraged to follow the PHP-FIG standards when writing your classes.

Note : Files with a « .class.php » extension will **not** be recognized by Elgg.

Vendors

Included third-party libraries of any kind *should* be included in the `vendors/` folder in the plugin root. Though this folder has no special significance to the Elgg engine, this has historically been the location where Elgg core stores its third-party libraries, so we encourage the same format for the sake of consistency and familiarity.

Views

In order to override core views, a plugin's views can be placed in `views/`, or an `elgg-plugin.php` config file can be used for more detailed file/path mapping. See *Views*.

Javascript and CSS will live in the views system. See *JavaScript*.

activate.php and deactivate.php

The `activate.php` and `deactivate.php` files contain procedural code that will run respectively upon plugin activation or deactivation. Use these files to perform one-time events such as registering a persistent admin notice, registering subtypes, or performing garbage collection when deactivated.

Plugin Dependencies

In Elgg the plugin dependencies system is there to prevent plugins from being used on incompatible systems.

Contents

Overview

The dependencies system is controlled through a plugin's `manifest.xml` file. Plugin authors can specify that a plugin :

- Requires certain Elgg versions, Elgg plugins, PHP extensions, and PHP settings.
- Suggests certain Elgg versions, Elgg plugins, PHP extensions, and PHP settings.
- Conflicts with certain Elgg versions or Elgg plugins.
- Provides the equivalent of another Elgg plugin or PHP extension.

The dependency system uses the four verbs above (`requires`, `suggests`, `conflicts`, and `provides`) as parent elements to indicate what type of dependency is described by its children. All dependencies have a similar format with similar options :

```
<verb>
   <type>type</type>
   <noun>value</noun>
   <noun2>value2</noun2>
</verb>
```

Note : `type` is always required

Verbs

With the exception of `provides`, all verbs use the same six types with differing effects, and the type options are the same among the verbs. `provides` only supports `plugin` and `php_extension`.

Requires

Using a `requires` dependency means that the plugin cannot be enabled unless the dependency is exactly met.

Mandatory requires : elgg_release

Every plugin must have at least one requires : the version of Elgg the plugin is developed for. This is specified by the Elgg API `release` (1.8). The default comparison >=, but you can specify your own by passing the `<comparison>` element.

Using elgg_release :

```
<requires>
   <type>elgg_release</type>
   <version>1.8</version>
</requires>
```

Suggests

`suggests` dependencies signify that the plugin author suggests a specific system configuration, but it is not required to use the plugin. The suggestions can also be another plugin itself which could interact, extend, or be extended by this plugin, but is not required for it to function.

Suggest another plugin :

```
<suggests>
   <type>plugin</type>
   <name>profile</name>
   <version>1.0</version>
</suggests>
```

Suggest a certain PHP setting :

```
<suggests>
   <type>php_ini</type>
   <name>memory_limit</name>
   <value>64M</value>
   <comparison>ge</comparison>
</suggests>
```

Conflicts

`conflicts` dependencies mean the plugin cannot be used under a specific system configuration.

Conflict with any version of the profile plugin :

```
<conflicts>
   <type>plugin</type>
   <name>profile</name>
</conflicts>
```

Conflict with a specific release of Elgg :

```
<conflicts>
   <type>elgg_release</type>
   <version>1.8</version>
   <comparison>==</comparison>
</conflicts>
```

Provides

`provides` dependencies tell Elgg that this plugin is providing the functionality of another plugin or PHP extension. Unlike the other verbs, it only supports two types : `plugin` and `php_extension`.

The purpose of this is to provide interchangeable APIs implemented by different plugins. For example, the twitter_services plugin provides an API for other plugins to Tweet on behalf of the user via curl and Oauth. A plugin author could write a compatible plugin for servers without curl support that uses sockets streams and specify that it provides twitter_services. Any plugins that suggest or require twitter_services would then know they can work.

```
<provides>
   <type>plugin</type>
   <name>twitter_services</name>
   <version>1.8</version>
</provides>
```

Note : All plugins provide themselves as their plugin id (directory name) at the version defined in the their manifest.

Types

Every dependency verb has a mandatory `<type>` element that must be one of the following six values :

1. **elgg_release** - The release version of Elgg (1.8)
2. **plugin** - An Elgg plugin
3. **priority** - A plugin load priority
4. **php_extension** - A PHP extension
5. **php_ini** - A PHP setting
6. **php_version** - A PHP version

Note : `provides` only supports `plugin` and `php_extension` types.

Every type is defined with a dependency verb as the parent element. Additional option elements are at the same level as the type element :

```
<verb>
   <type>type</type>
   <option_1>value_1</option_1>
   <option_2>value_2</option_2>
</verb>
```

elgg_release

These concern the API and release versions of Elgg and requires the following option element :
- **version** - The API or release version

The following option element is supported, but not required :
- **comparison** - The comparison operator to use. Defaults to >= if not passed

plugin

Specifies an Elgg plugin by its ID (directory name). This requires the following option element :
- **name** - The ID of the plugin

The following option elements are supported, but not required :
- **version** - The version of the plugin
- **comparison** - The comparison operator to use. Defaults to >= if not passed

priority

This requires the plugin to be loaded before or after another plugin, if that plugin exists. `requires` should be used to require that a plugin exists. The following option elements are required :
- **plugin** - The plugin ID to base the load order on
- **priority** - The load order : "before" or "after"

php_extension

This checks PHP extensions. The follow option element is required :
- **name** - The name of the PHP extension

The following option elements are supported, but not required :
- **version** - The version of the extension
- **comparison** - The comparison operator to use. Defaults to ==

Note : The format of extension versions varies greatly among PHP extensions and is sometimes not even set. This is generally worthless to check.

---

php_ini

This checks PHP settings. The following option elements are required :
- **name** - The name of the setting to check
- **value** - The value of the setting to compare against

The following options are supported, but not required :
- **comparison** - The comparison operator to use. Defaults to ==

php_version

This checks the PHP version. The following option elements are required :
- **version** - The PHP version

The following option element is supported, but not required :
- **comparison** - The comparison operator to use. Defaults to >= if not passed

Comparison Operators

Dependencies that check versions support passing a custom operator via the `<comparison>` element.

The follow are valid comparison operators :
- < or lt
- <= or le
- =, ==, or eq
- !=, <>, or ne
- > or gt
- >= or ge

If `<comparison>` is not passed, the follow are used as defaults, depending upon the dependency type :
- requires->elgg_release : >=
- requires->plugin : >=
- requires->php_extension : =
- requires->php_ini : =
- all conflicts : =

Note : You must escape < and > to `&gt;` and `&lt;`. For comparisons that use these values, it is recommended you use the string equivalents instead !

Quick Examples

Requires Elgg 1.8.2 or higher

```
<requires>
   <type>elgg_release</type>
   <version>1.8.2</version>
</requires>
```

Requires the Groups plugin is active

```
<requires>
   <type>plugin</type>
   <name>groups</name>
</requires>
```

Requires to be after the Profile plugin if Profile is active

```
<requires>
   <type>priority</type>
   <priority>after</priority>
   <plugin>profile</plugin>
</requires>
```

Conflicts with The Wire plugin

```
<conflicts>
   <type>plugin</type>
   <name>thewire</name>
</conflicts>
```

Requires at least 256 MB memory in PHP

```
<requires>
   <type>php_ini</type>
   <name>memory_limit</name>
   <value>256M</value>
   <comparison>ge</comparison>
</requires>
```

Requires at least PHP version 5.3

```
<requires>
   <type>php_version</type>
   <version>5.3</version>
</requires>
```

Suggest the TidyPics plugin is loaded

```
<suggests>
   <type>plugin</type>
   <name>tidypics</name>
</suggests>
```

Plugin bootstrap

In order to bootstrap your plugin as of Elgg 3.0 you can use a bootstrap class. This class must implement the `\Elgg\PluginBootstrapInterface` interface, but it's recommended you extend the `\Elgg\PluginBootstrap` abstract class as some preparations have already been done.

If you only need a limited subset of the bootstrap functions your class can also extend the `\Elgg\DefaultPluginBootstrap` class, this class already has all the functions of `\Elgg\PluginBootstrapInterface` implemented. So you can overload only the functions you need.

Contents

Registering the bootstrap class

You must register your bootstrap class in the `elgg-plugin.php` file.

```
return [
        // Bootstrap must implement \Elgg\PluginBootstrapInterface
        'bootstrap' => MyPluginBootstrap::class,
];
```

Available functions

->load()

Executed during `plugins_load`, `system` event

Allows the plugin to require additional files, as well as configure services prior to booting the plugin.

->boot()

Executed during `plugins_boot:before`, `system` event

Allows the plugin to register handlers for `plugins_boot`, `system` and `init`, `system` events, as well as implement boot time logic.

->init()

Executed during `init`, `system` event

Allows the plugin to implement business logic and register all other handlers.

->ready()

Executed during `ready`, `system` event

Allows the plugin to implement logic after all plugins are initialized.

->shutdown()

Executed during `shutdown`, `system` event

Allows the plugin to implement logic during shutdown.

->activate()

Executed when plugin is activated, after `activate`, `plugin` event and before `activate.php` is included.

->deactivate()

Executed when plugin is deactivated, after `deactivate`, `plugin` event and before `deactivate.php` is included.

->upgrade()

Registered as handler for `upgrade`, `system` event

Allows the plugin to implement logic during system upgrade.

Available helper functions

This assumes your bootstrap class extends the `\Elgg\PluginBootstrap` abstract class or the `\Elgg\DefaultPluginBootstrap` class.

->elgg()

Returns Elgg's public DI container. This can be helpfull if you wish to register plugin hooks or event listeners.

```
$hooks = $this->elgg()->hooks;
$hooks->registerHandler('register', 'menu:entity', 'my_custom_menu_callback');

$events = $this->elgg()->events;
$events->registerHandler('create', 'object', MyCustomObjectHandler::class);
```

->plugin()

Returns plugin entity this bootstrap is related to. This makes it easier to get plugin settings.

```
$plugin = $this->plugin();
$my_setting = $plugin->getSetting('my_setting');
```

3.3.24 River

Elgg natively supports the « river », an activity stream containing descriptions of activities performed by site members. This page gives an overview of adding events to the river in an Elgg plugin.

Pushing river items

Items are pushed to the activity river through a function call, which you must include in your plugins for the items to appear.

Here we add a river item telling that a user has created a new blog post :

```
<?php

elgg_create_river_item([
        'view' => 'river/object/blog/create',
        'action_type' => 'create',
        'subject_guid' => $blog->owner_guid,
        'object_guid' => $blog->getGUID(),
]);
```

All available parameters :

— `view` => STR The view that will handle the river item (must exist)

- `action_type` => STR An arbitrary string to define the action (e.g. “create”, “update”, “vote”, “review”, etc)
- `subject_guid` => INT The GUID of the entity doing the action (default : the logged in user guid)
- `object_guid` => INT The GUID of the entity being acted upon
- `target_guid` => INT The GUID of the the object entity’s container (optional)
- `access_id` => INT The access ID of the river item (default : same as the object)
- `posted` => INT The UNIX epoch timestamp of the river item (default : now)
- `annotation_id` => INT The annotation ID associated with this river entry (optional)

When an item is deleted or changed, the river item will be updated automatically.

River views

As of Elgg 3.0 the `view` parameter is no longer required. A fallback logic has been created to check a series of views for you :

1. `/river/{$type}/{$subtype}/{$action_type}` : eg. `river/object/blog/create` only the `create` action will come to this view
2. `river/{$type}/{$subtype}/default` : eg. `river/object/blog/default` all river activity for `object blog` will come here
3. `river/{$type}/{$action_type}` : eg. `river/object/create` all `create` actions for `object` will come here
4. `river/{$type}/default` : eg. `river/object/default` all actions for all `object` will come here
5. `river/elements/layout` : ultimate fall back view, this should always be called in any of the river views to make a consistent layout

Both `type` and `subtype` are based on the `type` and `subtype` of the `object_guid` for which the river item was created.

Summary

If no `summary` parameter is provided to the `river/elements/layout` the view will try to create it for you. The basic result will be a text with the text *Somebody did something on Object*, where *Somebody* is based on `subject_guid` and *Object* is based on `object_guid`. For both *Somebody* and *Object* links will be created. These links are passed to a series of language keys so you can create a meaningfull summary.

The language keys are :

1. `river:{$type}:{$subtype}:{$action_type}` : eg. `river:object:blog:create`
2. `river:{$type}:{$subtype}:default` : eg. `river:object:blog:default`
3. `river:{$type}:{$action_type}` : eg. `river:object:create`
4. `river:{$type}:default` : eg. `river:object:default`

Custom river view

If you wish to add some more information to the river view, like an attachment (image, YouTube embed, etc), you must specify the *view* when creating the river item. This view **MUST** exist.

We recommend `/river/{type}/{subtype}/{action}`, where :

- `{type}` is the entity type of the content we’re interested in (`object` for objects, `user` for users, etc)
- `{subtype}` is the entity subtype of the content we’re interested in (`blog` for blogs, `photo_album` for albums, etc)
- `{action}` is the action that took place (`create`, `update`, etc)

River item information will be passed in an object called `$vars['item']`, which contains the following important parameters :

— `$vars['item']->subject_guid` The GUID of the user performing the action
— `$vars['item']->object_guid` The GUID of the entity being acted upon

Timestamps etc will be generated for you.

For example, the blog plugin uses the following code for its river view :

```
$item = elgg_extract('item', $vars);
if (!$item instanceof ElggRiverItem) {
        return;
}

$blog = $item->getObjectEntity();
if (!$blog instanceof ElggBlog) {
        return;
}

$vars['message'] = $blog->getExcerpt();

echo elgg_view('river/elements/layout', $vars);
```

3.3.25 Routing

Elgg has two mechanisms to respond to HTTP requests that don't already go through the *Actions* and *Simplecache* systems.

URL Identifier and Segments

After removing the site URL, Elgg splits the URL path by `/` into an array. The first element, the **identifier**, is shifted off, and the remaining elements are called the **segments**. For example, if the site URL is `http://example.com/elgg/`, the URL `http://example.com/elgg/blog/owner/jane?foo=123` produces :

Identifier : `'blog'`. Segments : `['owner', 'jane']`. (the query string parameters are available via `get_input()`)

The site URL (home page) is a special case that produces an empty string identifier and an empty segments array.

> **Avertissement :** URL identifier/segments should be considered potentially dangerous user input. Elgg uses `htmlspecialchars` to escapes HTML entities in them.

Page Handling

Elgg offers a facility to manage your plugin pages via custom routes, enabling URLs like `http://yoursite/my_plugin/section`. You can register a new route using `elgg_register_route()`, or via `routes` config in `elgg-plugin.php`. Routes map to resource views, where you can render page contents.

```
// in your 'init', 'system' handler
elgg_register_route('my_plugin:section' [
        'path' => '/my_plugin/section/{guid}/{subsection?}',
        'resource' => 'my_plugin/section',
        'requirements' => [
                'guid' => '\d+',
                'subsection' => '\w+',
        ],
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);

// in my_plugin/views/default/resources/my_plugin/section.php
$guid = elgg_extract('guid', $vars);
$subsection = elgg_extract('subsection', $vars);

// render content
```

In the example above, we have registered a new route that is accessible via `http://yoursite/my_plugin/section/<guid>/<subsection>`. Whenever that route is accessed with a required `guid` segment and an optional `subsection` segment, the router will render the specified `my_plugin/section` resource view and pass the parameters extracted from the URL to your resource view with `$vars`.

Routes names

Route names can then be used to generate a URL :

```
$url = elgg_generate_url('my_plugin:section', [
        'guid' => $entity->guid,
        'subsection' => 'assets',
]);
```

The route names are unique across all plugins and core, so another plugin can override the route by registering different parameters to the same route name.

Route names follow a certain convention and in certain cases will be used to automatically resolve URLs, e.g. to display an entity.

The following conventions are used in core and recommended for plugins :

view :<entity_type> :<entity_subtype> Maps to the entity profile page, e.g. `view:user:user` or `view:object:blog` The path must contain a `guid`, or `username` for users

edit :<entity_type> :<entity_subtype> Maps to the form to edit the entity, e.g. `edit:user:user` or `edit:object:blog` The path must contain a `guid`, or `username` for users If you need to add subresources, use suffixes, e.g. `edit:object:blog:images`, keeping at least one subresource as a default without suffix.

add :<entity_type> :<entity_subtype> Maps to the form to add a new entity of a given type, e.g. `add:object:blog` The path, as a rule, contains `container_guid` parameter

collection :<entity_type> :<entity_subtype> :<collection_type> Maps to listing pages. Common route names used in core are, as follows :

— `collection:object:blog:all` : list all blogs
— `collection:object:blog:owner` : list blogs owned by a user with a given username
— `collection:object:blog:friends` : list blogs owned by friends of the logged in user (or user with a given username)
— `collection:object:blog:group` : list blogs in a group

default :<entity_type> :<entity_subtype> Maps to the default page for a resource, e.g. the path `/blog`. Elgg happens to use the « all » collection for these routes.

— `default:object:blog` : handle the generic path `/blog`.

`<entity_subtype>` can be omitted from route names to register global routes applicable to all entities of a given type. URL generator will first try to generate a URL using the subtype, and will then fallback to a route name without a subtype. For example, user profiles are routed to the same resource view regardless of user subtype.

```
elgg_register_route('view:object:attachments', [
        'path' => '/attachments/{guid}',
        'resource' => 'attachments',
]);

elgg_register_route('view:object:blog:attachments', [
        'path' => '/blog/view/{guid}/attachments',
        'resource' => 'blog/attachments',
]);

$blog = get_entity($blog_guid);
$url = elgg_generate_entity_url($blog, 'view', 'attachments'); // /blog/view/$blog_
↪guid/attachments

$other = get_entity($other_guid);
$url = elgg_generate_entity_url($other, 'view', 'attachments'); // /attachments/
↪$other_guid
```

Route configuration

Segments can be defined using wildcards, e.g. `profile/{username}`, which will match all URLs that contain `profile/` followed by and arbitrary username.

To make a segment optional you can add a `?` (question mark) to the wildcard name, e.g. `profile/{username}/{section?}`. In this case the URL will be matched even if the `section` segment is not provided.

You can further constrain segments using regex requirements :

```
// elgg-plugin.php
return [
        'routes' => [
                'profile' => [
                        'path' => '/profile/{username}/{section?}',
                        'resource' => 'profile',
                        'requirements' => [
                                'username' => '[\p{L}\p{Nd}._-]+', // only allow␣
↪valid usernames
                                'section' => '\w+', // can only contain alphanumeric␣
↪characters
                        ],
                        'defaults' => [
                                'section' => 'index',
                        ],
                ],
        ]
];
```

By default, Elgg will set the following requirements for named URL segments :

```
$patterns = [
        'guid' => '\d+', // only digits
        'group_guid' => '\d+', // only digits
        'container_guid' => '\d+', // only digits
        'owner_guid' => '\d+', // only digits
        'username' => '[\p{L}\p{Nd}._-]+', // letters, digits, underscores, dashes
];
```

Route middleware

Route middleware can be used to prevent access to a certain route, or to perform some business logic before the route is rendered. Middleware can be used, e.g. to implement a paywall, or to log analytics, or to set open graph metatags.

Elgg core implements several middleware handlers, including :

- `\Elgg\Router\Middleware\Gatekeeper` - prevent access by non-authenticated users
- `\Elgg\Router\Middleware\AdminGatekeeper` - prevent access by non-admin users
- `\Elgg\Router\Middleware\AjaxGatekeeper` - prevent access with non-xhr requests
- `\Elgg\Router\Middleware\CsrfFirewall` - prevent access without CSRF tokens

Middleware handlers can be set to any callable that receives an instance of `\Elgg\Request` : The handler should throw an instance of `HttpException` to prevent route access. The handler can return an instance of `\Elgg\Http\ResponseBuilder` to prevent further implementation of the routing sequence (a redirect response can be returned to re-route the request).

```
class MyMiddleware {

        public function __invoke(\Elgg\Request $request) {
                $entity = $request->getEntityParam();
                if ($entity) {
                        // do stuff
                } else {
                        throw new EntityNotFoundException();
                }
        }
}

elgg_register_route('myroute', [
        'path' => '/myroute/{guid?}',
        'resource' => 'myroute',
        'middleware' => [
                \Elgg\Router\Middleware\Gatekeeper::class,
                MyMiddleware::class,
        ]
]);
```

Route controllers

In certain cases, using resource views is not appropriate. In these cases you can use a controller - any callable that receives an instance of `\Elgg\Request` :

```
class MyController {

        public function handleFoo(\Elgg\Request $request) {
                elgg_set_http_header('Content-Type: application/json');
                $data = [
                        'entity' => $request->getEntityParam(),
                ];
                return elgg_ok_response($data);
        }

}

elgg_register_route('myroute', [
        'path' => '/myroute/{guid?}',
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'controller' => [MyController::class, 'handleFoo'],
]);
```

The `route:rewrite` Plugin Hook

For URL rewriting, the `route:rewrite` hook (with similar arguments as `route`) is triggered very early, and allows modifying the request URL path (relative to the Elgg site).

Here we rewrite requests for `news/*` to `blog/*` :

```
function myplugin_rewrite_handler($hook, $type, $value, $params) {
    $value['identifier'] = 'blog';
    return $value;
}

elgg_register_plugin_hook_handler('route:rewrite', 'news', 'myplugin_rewrite_handler
↪');
```

Avertissement : The hook must be registered directly in your plugin `start.php` (the `[init, system]` event is too late).

Routing overview

For regular pages, Elgg's program flow is something like this :

1. A user requests `http://example.com/news/owner/jane`.
2. Plugins are initialized.
3. Elgg parses the URL to identifier `news` and segments `['owner', 'jane']`.
4. Elgg triggers the plugin hook `route:rewrite, news` (see above).
5. Elgg triggers the plugin hook `route, blog` (was rewritten in the rewrite hook).
6. Elgg finds a registered route that matches the final route path, and renders a resource view associated with it. It calls `elgg_view_resource('blog/owner', $vars)` where `$vars` contains the username.
7. The `resources/blog/owner` view gets the username via `$vars['username']`, and uses many other views and formatting functions like `elgg_view_layout()` and `elgg_view_page()` to create the entire HTML page.
8. PHP invokes Elgg's shutdown sequence.
9. The user receives a fully rendered page.

Elgg's coding standards suggest a particular URL layout, but there is no syntax enforced.

3.3.26 Search

Contents

Entity search

Elgg core provides flexible `elgg_search()`, which prepares custom search clauses and utilizes `elgg_get_entities()` to fetch the results.

In addition to all parameters accepted by `elgg_get_entities()`, `elgg_search()` accepts the following :

— `query` Search query
— `fields` An array of names by property type to search in (see example below)
— `sort` An array containing sorting options, including *property*, *property_type* and *direction*
— `type` Entity type to search
— `subtype` Optional entity subtype to search
— `search_type` Custom search type (required if no `type` is provided)
— **`partial_match` Allow partial matches** By default partial matches are allowed, meaning that `elgg` will be matched when searching for `el` Exact matches may be helpful when you want to match tag values, e.g. when you want to find all objects that are `red` and not `darkred`
— **`tokenize` Break down search query into tokens** By default search queries are tokenized, meaning that we will match `elgg has been released` when searching for `elgg released`

```
// List all users who list United States as their address or mention it in their␣
↪description
$options = [
    'type' => 'user',
    'query' => 'us',
    'fields' => [
        'metadata' => ['description'],
        'annotations' => ['location'],
    ],
    'sort' => [
        'property' => 'zipcode',
        'property_type' => 'annotation',
        'direction' => 'asc',
    ]
];

echo elgg_list_entities($options, 'elgg_search');
```

Search fields

You can customize search fields for each entity type/subtype, using `search:fields` hook :

```
// Let's remove search in location and add address field instead
elgg_register_plugin_hook_handler('search:fields', 'user', 'my_plugin_search_user_
↪fields');

function my_plugin_search_user_fields(\Elgg\Hook $hook) {
    $fields = $hook->getValue();
    $location_key = array_search('location', $fields['annotations']);
    if ($location_key) {
        unset($fields[$location_key]['annotations']);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $fields['metadata'][] = 'address';

    return $fields;
}
```

Searchable types

To register an entity type for search, use `elgg_register_entity_type()`, or do so when defining an entity type in `elgg-plugin.php`. To combine search results or filter how search results are presented in the search plugin, use `'search:config', 'type_subtype_pairs'` hook.

```
// Let's add places and place reviews as public facing entities
elgg_register_entity_type('object', 'place');
elgg_register_entity_type('object', 'place_review');

// Now let's include place reviews in the search results for places
elgg_register_plugin_hook_handler('search:options', 'object:place', 'my_plugin_place_
↪search_options');
elgg_register_plugin_hook_handler('search:config', 'type_subtype_pairs', 'my_plugin_
↪place_search_config');

// Add place review to search options as a subtype
function my_plugin_place_search_options($hook, $type, $value, $params) {

    if (empty($params) || !is_array($params)) {
        return;
    }

    if (isset($params['subtypes'])) {
        $subtypes = (array) $params['subtypes'];
    } else {
        $subtypes = (array) elgg_extract('subtype', $params);
    }

    if (!in_array('place', $subtypes)) {
        return;
    }

    unset($params["subtype"]);

    $subtypes[] = 'place_review';
    $params['subtypes'] = $subtypes;

    return $params;
}

// Remove place reviews as a separate entry in search sections
function my_plugin_place_search_config(\Elgg\Hook $hook) {

    $types = $hook->getValue();

    if (empty($types['object'])) {
        return;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    foreach ($types['object'] as $key => $subtype) {
        if ($subtype == 'place_review') {
            unset($types['object'][$key]);
        }
    }

    return $types;
}
```

Custom search types

Elgg core only supports entity search. You can implement custom searches, e.g. using search query as a location and listing entities by proximity to that location.

```
// Let's added proximity search type
elgg_register_plugin_hook_handler('search:config', 'search_types', function (\Elgg\
↪Hook $hook) {
    $search_types = $hook->getValue();
    $search_types[] = 'promimity';

    return $search_types;
});

// Let's add search options that will look for entities that have geo coordinates and␣
↪order them by proximity to the query location
elgg_register_plugin_hook_handler('search:options', 'proximity', function (\Elgg\Hook
↪$hook) {

    $query = $hook->getParam('query');
    $options = $hook->getValue();

    // Let's presume we have a geocoding API
    $coords = geocode($query);

    // We are not using standard 'selects' options here, because counting queries do␣
↪not use custom selects
    $options['wheres']['proximity'] = function (QueryBuilder $qb, $alias) use ($lat,
↪$long) {
        $dblat = $qb->joinMetadataTable($alias, 'guid', 'geo:lat');
        $dblong = $qb->joinMetadataTable($alias, 'guid', 'geo:long');

        $qb->addSelect("(((acos(sin(($lat*pi()/180))
                    *sin(($dblat.value*pi()/180)) + cos(($lat*pi()/180))
                    *cos(($dblat.value*pi()/180))
                    *cos((($long-$dblong.value)*pi()/180)))))*180/pi())
                    *60*1.1515*1.60934
                    AS proximity");

        $qb->orderBy('proximity', 'asc');

        return $qb->merge([
            $qb->compare("$dblat.value", 'is not null'),
            $qb->compare("$dblong.value", 'is not null'),
        ]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    };

    return $options;
});
```

Autocomplete and livesearch endpoint

Core provides a JSON endpoint for searching users and groups. These endpoints are used by `input/autocomplete` and `input/userpicker` views.

```
// Get JSON results of a group search for 'class'
$json = file_get_contents('http://example.com/livesearch/groups?view=json&q=class');
```

You can add custom search types, by adding a corresponding resource view :

```
// Let's add an endpoint that will search for users that are not members of a group
// and render a userpicker for our invite form
echo elgg_view('input/userpicker', [
    'handler' => 'livesearch/non_members',
    'options' => [
        // this will be sent as URL query elements
        'group_guid' => $group_guid,
    ],
]);

// To enable /livesearch/non_members endpoint, we need to add a view
// in /views/json/resources/livesearch/non_members.php

$limit = get_input('limit', elgg_get_config('default_limit'));
$query = get_input('term', get_input('q'));
$input_name = get_input('name');

// We have passed this value to our input view, and we want to make sure
// external scripts are not using it to mine data on group members
// so let's validate the HMAC that was generated by the userpicker input
$group_guid = (int) get_input('group_guid');

$data = [
    'group_guid' => $group_guid,
];

// let's sort by key, in case we have more elements
ksort($data);

$hmac = elgg_build_hmac($data);
if (!$hmac->matchesToken(get_input('mac'))) {
     // request does not originate from our input view
     forward('', '403');
}

elgg_set_http_header("Content-Type: application/json;charset=utf-8");

$options = [
    'query' => $query,
    'type' => 'user',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'limit' => $limit,
    'sort' => 'name',
    'order' => 'ASC',
    'fields' => [
        'metadata' => ['name', 'username'],
    ],
    'item_view' => 'search/entity',
    'input_name' => $input_name,
    'wheres' => function (QueryBuilder $qb) use ($group_guid) {
        $subquery = $qb->subquery('entity_relationships', 'er');
        $subquery->select('1')
            ->where($qb->compare('er.guid_one', '=', 'e.guid'))
            ->andWhere($qb->compare('er.relationship', '=', 'member', ELGG_VALUE_
↪STRING))
            ->andWhere($qb->compare('er.guid_two', '=', $group_guid, ELGG_VALUE_
↪INTEGER));

        return "NOT EXISTS ({$subquery->getSQL()})";
    }
];

echo elgg_list_entities($options, 'elgg_search');
```

3.3.27 Services

Elgg uses the `Elgg\Application` class to load and bootstrap Elgg. In future releases this class will offer a set of service objects for plugins to use.

Note : If you have a useful idea, you can *add a new service* !

Menus

`elgg()->menus` provides low-level methods for constructing menus. In general, menus should be passed to `elgg_view_menu` for rendering instead of manual rendering.

3.3.28 Plugin settings

You need to perform some extra steps if your plugin needs settings to be saved and controlled via the administration panel :

- Create a file in your plugin's default view folder called `plugins/your_plugin/settings.php`, where `your_plugin` is the name of your plugin's directory in the `mod` hierarchy
- Fill this file with the form elements you want to display together with *internationalised* text labels
- Set the name attribute in your form components to `params[`varname`]` where `varname` is the name of the variable. These will be saved as private settings attached to a plugin entity. So, if your variable is called `params[myparameter]` your plugin (which is also passed to this view as `$vars['entity']`) will be called `$vars['entity']->myparameter`

An example `settings.php` would look like :

```
<p>
   <?php echo elgg_echo('myplugin:settings:limit'); ?>

   <select name="params[limit]">
      <option value="5" <?php if ($vars['entity']->limit == 5) echo " selected=\"yes\
↪" "; ?>>5</option>
      <option value="8" <?php if ((!$vars['entity']->limit) || ($vars['entity']->
↪limit == 8)) echo " selected=\"yes\" "; ?>>8</option>
      <option value="12" <?php if ($vars['entity']->limit == 12) echo " selected=\
↪"yes\" "; ?>>12</option>
      <option value="15" <?php if ($vars['entity']->limit == 15) echo " selected=\
↪"yes\" "; ?>>15</option>
   </select>
</p>
```

Note : You don't need to add a save button or the form, this will be handled by the framework.

Note : You cannot use form components that send no value when « off. » These include radio inputs and check boxes.

If your plugin settings require a cache flush you can add a (hidden) input on the form with the name "flush_cache" and value "1"

```
elgg_view_field([
        '#type' => 'hidden',
        'name' => 'flush_cache',
        'value' => 1,
]);
```

User settings

Your plugin might need to store per user settings too, and you would like to have your plugin's options to appear in the user's settings page. This is also easy to do and follows the same pattern as setting up the global plugin configuration explained earlier. The only difference is that instead of using a `settings` file you will use `usersettings`. So, the path to the user edit view for your plugin would be `plugins/your_plugin/usersettings.php`.

Note : The title of the usersettings form will default to the plugin name. If you want to change this, add a translation for `plugin_id:usersettings:title`.

Retrieving settings in your code

To retrieve settings from your code use :

```
$setting = elgg_get_plugin_setting($name, $plugin_id);
```

or for user settings

```
$user_setting = elgg_get_plugin_user_setting($name, $user_guid, $plugin_id);
```

where :

— `$name` Is the value you want to retrieve
— `$user_guid` Is the user you want to retrieve these for (defaults to the currently logged in user)
— `$plugin_name` Is the name of the plugin (detected if run from within a plugin)

Setting values while in code

Values may also be set from within your plugin code, to do this use one of the following functions :

```
elgg_set_plugin_setting($name, $value, $plugin_id);
```

or

```
elgg_set_plugin_user_setting($name, $value, $user_guid, $plugin_id);
```

Avertissement : The `$plugin_id` needs to be provided when setting plugin (user)settings.

Default plugin (user) settings

If a plugin or a user not have a setting stored in the database, you sometimes have the need for a certain default value. You can pass this when using the getter functions.

```
$user_setting = elgg_get_plugin_user_setting($name, $user_guid, $plugin_id, $default);

$plugin_setting = elgg_get_plugin_setting($name, $plugin_id, $default);
```

Alternatively you can also provide default plugin and user settings in the `elgg-plugin.php` file.

```
<?php

return [
        'settings' => [
            'key' => 'value',
        ],
        'user_settings' => [
            'key' => 'value',
        ],
];
```

3.3.29 Themes

Customize the look and feel of Elgg.

A theme is a type of *plugin* that overrides display aspects of Elgg.

This guide assumes you are familiar with :
— *Plugins*
— *Views*

Contents

Theming Principles and Best Practices

No third-party CSS frameworks Elgg does not use a CSS framework, because such frameworks lock users into a specific HTML markup, which in the end makes it much harder for plugins to collaborate on the appearance. What's *is-primary* in one theme, might be something else in the other. Having no framework allows plugins to alter appearance using pure css, without having to overwrite views and append framework-specific selectors to HTML markup elements.

```
/* BAD */
<div class="box has-shadow is-inline">
        This is bad, because if the plugin wants to change the styling, it will have␣
↪to either write really specific css
        clearing all the attached styles, or replace the view entirely just to modify␣
↪the markup
</div>

/* GOOD */
<div class="box-role">
        This is good, because a plugin can just simply add .box-role rule
</div>
<style>
        .box-role {
         padding: 1rem;
         display: inline-block;
         box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);
        }
</style>
```

8-point grid system Elgg uses an *8-point grid system <https ://builttoadapt.io/intro-to-the-8-point-grid-system-d2573cde8632>*, so sizing of elements, their padding, margins etc is done in increments and fractions of *8px*. Because our default font-size is `16px`, we use fractions of *rem*, so `0.5rem = 8px`. 8-point grid system makes it a lot easier for developers to collaborate on styling elements : we no longer have to think if the padding should be `5px` or `6px`.

```
/* BAD */
.menu > li {
        margin: 2px 2px 2px 0;
}

.menu > li > a {
        padding: 3px 5px;
}

/* GOOD */
```

(suite sur la page suivante)

(suite de la page précédente)

```
.menu > li > a {
        padding: 0.25rem 0.5rem;
}
```

Mobile first We write mobile-first CSS. We use two breakpoints : `50rem` and `80rem` (800px and 1280px at 16px/rem).

```
/* BAD: mobile defined in media blocks, different display types */

.menu > li {
        display: inline-block;
}
@media screen and (max-width: 820px) {
        .menu > li {
                display: block;
                width: 100%;
        }
}

/* GOOD: mobile by default. Media blocks style larger viewports. */

.menu {
        display: flex;
        flex-direction: column;
}
@media screen and (min-width: 50rem) {
        .menu {
                flex-direction: row;
        }
}
```

Flexbox driven Flexbox provides simplicity in stacking elements into grids. Flexbox is used for everything from menus to layout elements. We avoid `float` and `clearfix` as they are hard to collaborate on and create lots of room for failure and distortion.

```
/* BAD */
.heading:after {
        visibility: hidden;
        height: 0;
        clear: both;
        content: " ";
}
.heading > h2 {
        float: left;
}
.heading > .controls {
        float: right;
}

/* GOOD */
.heading {
        display: flex;
        justify-content: flex-end;
}
.heading > h2 {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        order: 1;
        margin-right: auto;
}
.heading > .controls {
        order: 2;
}
```

Symmetrical We maintain symmetry.

```
/* BAD */
.row .column:first-child {
        margin-right: 10px;
}

/* GOOD */
.row {
        margin: 0 -0.5rem;
}
.row .column {
        margin: 0.5rem;
}
```

Simple color transitions We maintain 4 sets of colors for text, background and border : `soft`, `mild`, `strong` and `highlight`. When transitioning to hover or active state, we go one level up, e.g. from `soft` to `mild`, or use `highlight`. When transition to inactive or disabled state, we go one level down.

Increase the click area When working with nested anchors, we increase the click area of the anchor, rather than the parent

```
/* BAD */
.menu > li {
        margin: 5px;
        padding: 5px 10px;
}

/* GOOD */
.menu > li {
        margin: 0.5rem;
}
.menu > li > a {
        padding: 0.5rem 1rem;
}
```

No z-index 999999 z-indexes are incremented with a step of 1.

Wrap HTML siblings We make sure that there are no orphaned strings within a parent and that siblings are wrapped in a way that they can be targeted by CSS.

```
/* BAD */
<label>
        Orphan
        <span>Sibling</span>
</label>

/* GOOD */
```

(suite sur la page suivante)

(suite de la page précédente)

```
<label>
        <span>Sibling</span>
        <span>Sibling</span>
</label>
```

```
/* BAD */
<div>
        <h3>Title</h3>
        <p>Subtitle</p>
        <div class="right">This goes to the right</div>
</div>

/* GOOD */
<div>
        <div class="left">
         <h3>Title</h3>
         <p>Subtitle</p>
        </div>
        <div class="right">This goes to the right</div>
</div>
```

Create your plugin

Create your plugin as described in the *developer guide*.

- Create a new directory under mod/
- Create a new start.php
- Create a manifest.xml file describing your theme.

Customize the CSS

The css is split into several files based on what aspects of the site you're theming. This allows you to tackle them one at a time, giving you a chance to make real progress without getting overwhelmed.

Here is a list of the existing CSS views :

- elements/buttons.css : Provides a way to style all the different kinds of buttons your site will use. There are 5 kinds of buttons that plugins will expect to be available : action, cancel, delete, submit, and special.
- elements/chrome.css : This file has some miscellaneous look-and-feel classes.
- elements/components.css : This file contains many "css objects" that are used all over the site : media block, list, gallery, table, owner block, system messages, river, tags, photo, and comments.
- elements/forms.css : This file determines what your forms and input elements will look like.
- elements/icons.css : Contains styles for the icons and avatars used on your site.
- elements/layout.css : Determines what your page layout will look like : sidebars, page wrapper, main body, header, footer, etc.
- elements/modules.css : Lots of content in Elgg is displayed in boxes with a title and a content body. We called these modules. There are a few kinds : info, aside, featured, dropdown, popup, widget. Widget styles are included in this file too, since they are a subset of modules.
- elements/navigation.css : This file determines what all your menus will look like.
- elements/typography.css : This file determines what the content and headings of your site will look like.
- rtl.css : Custom rules for users viewing your site in a right-to-left language.
- admin.css : A completely separate theme for the admin area (usually not overridden).
- elgg.css : Compiles all the core elements/* files into one file (DO NOT OVERRIDE).
- elements/core.css : Contains base styles for the more complicated "css objects". If you find yourself wanting to override this, you probably need to report a bug to Elgg core instead (DO NOT OVERRIDE).

— elements/reset.css : Contains a reset stylesheet that forces elements to have the same default

CSS variables

Elgg uses CssCrush for preprocessing CSS files. This gives us the flexibility of using global CSS variables. Plugins should, wherever possible, use global CSS variables, and extend the core theme with their plugin variables, so they can be simply altered by other plugins.

To add or alter variables, use the `vars:compiler, css` hook. Note that you may need to flush the cache to see your changes in action.

For a list of default core variables, see `engine/theme.php`.

View extension

There are two ways you can modify views :

The first way is to add extra stuff to an existing view via the extend view function from within your start.php's initialization function.

For example, the following start.php will add mytheme/css to Elgg's core css file :

```
<?php

        function mytheme_init() {
                elgg_extend_view('elgg.css', 'mytheme/css');
        }

        elgg_register_event_handler('init', 'system', 'mytheme_init');
?>
```

View overloading

Plugins can have a view hierarchy, any file that exists here will replace any files in the existing core view hierarchy... so for example, if my plugin has a file :

`/mod/myplugin/views/default/elements/typography.css`

it will replace :

`/views/default/elements/typography.css`

But only when the plugin is active.

This gives you total control over the way Elgg looks and behaves. It gives you the option to either slightly modify or totally replace existing views.

Icons

As of Elgg 2.0 the default Elgg icons come from the FontAwesome library. You can use any of these icons by calling :

`elgg_view_icon('icon-name');`

`icon-name` can be any of the FontAwesome icons without the `fa-`-prefix.

By default you will get the solid styled variant of the icons. Postfixing the icon name with `-solid`, `-regular` or `-light` allows you to target a specific style. Be advised ; the light styled variant is only available as a FontAwesome Pro licensed icon.

Tools

We've provided you with some development tools to help you with theming : Turn on the "Developers" plugin and go to the "Theme Preview" page to start tracking your theme's progress.

Customizing the front page

The main Elgg index page runs a plugin hook called "index,system". If this returns true, it assumes that another front page has been drawn and doesn't display the default page.

Therefore, you can override it by registering a function to the "index,system" plugin hook and then returning true from that function.

Here's a quick overview :

— Create your new plugin

— In the start.php you will need something like the following :

```
<?php

function pluginname_init() {
        // Replace the default index page
        elgg_register_plugin_hook_handler('index', 'system', 'new_index');
}

function new_index() {
        if (!include_once(dirname(dirname(__FILE__)) . "/pluginname/pages/index.php"))
                return false;

        return true;
}

// register for the init, system event when our plugin start.php is loaded
elgg_register_event_handler('init', 'system', 'pluginname_init');
?>
```

— **Then, create an index page (/pluginname/pages/index.php) and use that** to put the content you would like on the front page of your Elgg site.

3.3.30 Writing a plugin upgrade

Every now and then there comes a time when a plugin needs to change the contents or the structure of the data it has stored either in the database or the dataroot.

The motivation for this may be that the data structure needs to be converted to more efficient or flexible structure. Or perhaps due to a bug the data items have been saved in an invalid way, and they needs to be converted to the correct format.

Migrations and convertions like this may take a long time if there is a lot of data to be processed. This is why Elgg provides the `Elgg\Upgrade\Batch` interface that can be used for implementing long-running upgrades.

Declaring a plugin upgrade

Plugin can communicate the need for an upgrade under the `upgrades` key in `elgg-plugin.php` file. Each value of the array must be the fully qualified name of an upgrade class that implements the `Elgg\Upgrade\Batch` interface.

Example from `mod/blog/elgg-plugin.php` file :

```
return [
        'upgrades' => [
                Blog\Upgrades\AccessLevelFix::class,
                Blog\Upgrades\DraftStatusUpgrade::class,
        ]
];
```

The class names in the example refer to the classes :

- `mod/blog/classes/Blog/Upgrades/AccessLevelFix`
- `mod/blog/classes/Blog/Upgrades/DraftStatusUpgrade`

Note : Elgg core upgrade classes can be declared in `engine/lib/upgrades/async-upgrades.php`.

The upgrade class

A class implementing the `Elgg\Upgrade\Batch` interface has a lot of freedom on how it wants to handle the actual processing of the data. It must however declare some constant variables and also take care of marking whether each processed item was upgraded successfully or not.

The basic structure of the class is the following :

```
<?php

namespace Blog\Upgrades;

use Elgg\Upgrade\Batch;
use Elgg\Upgrade\Result;

/**
 * Fixes invalid blog access values
 */
class AccessLevelFix implements Batch {

```

(suite sur la page suivante)

(suite de la page précédente)

```
        /**
         * Version of the upgrade
         *
         * @return int
         */
        public function getVersion() {
                return 2016120300;
        }

        /**
         * Should the run() method receive an offset representing all processed items?
         *
         * @return bool
         */
        public function needsIncrementOffset() {
                return true;
        }

        /**
         * Should this upgrade be skipped?
         *
         * @return bool
         */
        public function shouldBeSkipped() {
                return false;
        }

        /**
         * The total number of items to process in the upgrade
         *
         * @return int
         */
        public function countItems() {
                // return count of all blogs
        }

        /**
         * Runs upgrade on a single batch of items
         *
         * @param Result $result Result of the batch (this must be returned)
         * @param int    $offset Number to skip when processing
         *
         * @return Result Instance of \Elgg\Upgrade\Result
         */
        public function run(Result $result, $offset) {
                // fix 50 blogs skipping the first $offset
        }
}
```

Avertissement : Do not assume when your class will be instantiated or when/how often its public methods will be called.

Class methods

getVersion()

This must return an integer representing the date the upgrade was added. It consists of eight digits and is in format `yyyymmddnn` where :

- `yyyy` is the year
- `mm` is the month (with leading zero)
- `dd` is the day (with leading zero)
- `nn` is an incrementing number (starting from `00`) that is used in case two separate upgrades have been added during the same day

shouldBeSkipped()

This should return `false` unless the upgrade won't be needed.

Avertissement : If `true` is returned the upgrade cannot be run later.

needsIncrementOffset()

If `true`, your `run()` method will receive as `$offset` the number of items aready processed. This is useful if you are only modifying data, and need to use the `$offset` in a function like `elgg_get_entities()` to know how many you've already handled.

If `false`, your `run()` method will receive as `$offset` the total number of failures. `false` should be used if your process deletes or moves data out of the way of the process. E.g. if you delete 50 objects on each `run()`, you don't really need the `$offset`.

countItems()

Get the total number of items to process during the upgrade. If unknown, `Batch::UNKNOWN_COUNT` can be returned, but `run()` must manually mark the upgrade complete.

run()

This must perform a portion of the actual upgrade. And depending on how long it takes, it may be called multiple times during a single request.

It receives two arguments :

- `$result` : An instance of `Elgg\Upgrade\Result` object
- `$offset` : The offset where the next upgrade portion should start (or total number of failures)

For each item the method processes, it must call either :

- `$result->addSuccesses()` : If the item was upgraded successfully
- `$result->addFailures()` : If it failed to upgrade the item

Both methods default to one item, but you can optionally pass in the number of items.

Additionally it can set as many error messages as it sees necessary in case something goes wrong :

- `$result->addError("Error message goes here")`

If `countItems()` returned `Batch::UNKNOWN_COUNT`, then at some point `run()` must call `$result->markComplete()` to finish the upgrade.

In most cases your `run()` method will want to pass the `$offset` parameter to one of the `elgg_get_entities()` functions :

```
/**
 * Process blog posts
 *
 * @param Result $result The batch result (will be modified and returned)
 * @param int    $offset Starting point of the batch
 * @return Result Instance of \Elgg\Upgrade\Result;
 */
public function run(Result $result, $offset) {
        $blogs = elgg_get_entitites([
                'type' => 'object'
                'subtype' => 'blog'
                'offset' => $offset,
        ]);

        foreach ($blogs as $blog) {
                if ($this->fixBlogPost($blog)) {
                        $result->addSuccesses();
                } else {
                        $result->addFailures();
                        $result->addError("Failed to fix the blog {$blog->guid}.");
                }
        }

        return $result;
}
```

Administration interface

Each upgrade implementing the `Elgg\Upgrade\Batch` interface gets listed in the admin panel after triggering the site upgrade from the Administration dashboard.

While running the upgrades Elgg provides :

- Estimated duration of the upgrade
- Count of processed items
- Number of errors
- Possible error messages

3.3.31 Views

Contents

Introduction

Views are responsible for creating output. They handle everything from :
- the layout of pages
- chunks of presentation output (like a footer or a toolbar)
- individual links and form inputs.
- the images, js, and css needed by your web page

Using views

At their most basic level, the default views are just PHP files with snippets of html :

```
<h1>Hello, World!</h1>
```

Assuming this view is located at `/views/default/hello.php`, we could output it like so :

```
echo elgg_view('hello');
```

For your convenience, Elgg comes with quite a lot of views by default. In order to keep things manageable, they are organized into subdirectories. Elgg handles this situation quite nicely. For example, our simple view might live in `/views/default/hello/world.php`, in which case it would be called like so :

```
echo elgg_view('hello/world');
```

The name of the view simply reflects the location of the view in the views directory.

Views as templates

You can pass arbitrary data to a view via the `$vars` array. Our `hello/world` view might be modified to accept a variable like so :

```
<h1>Hello, <?= $vars['name']; ?>!</h1>
```

In this case, we can pass an arbitrary name parameter to the view like so :

```
echo elgg_view('hello/world', ['name' => 'World']);
```

which would produce the following output :

```
<h1>Hello, World!</h1>
```

Avertissement : Views don't do any kind of automatic output sanitization by default. You are responsible for doing the correct sanitization yourself to prevent XSS attacks and the like.

Views as cacheable assets

As mentioned before, views can contain JS, CSS, or even images.

Asset views must meet certain requirements :
- They *must not* take any `$vars` parameters
- They *must not* change their output based on global state like
  - who is logged in
  - the time of day
- They *must* contain a valid file extension
  - Bad : `my/cool/template`
  - Good : `my/cool/template.html`

For example, suppose you wanted to load some CSS on a page. You could define a view `mystyles.css`, which would look like so :

```
/* /views/default/mystyles.css */
.mystyles-foo {
  background: red;
}
```

Note : Leave off the trailing « .php » from the filename and Elgg will automatically recognize the view as cacheable.

To get a URL to this file, you would use `elgg_get_simplecache_url` :

```
// Returns "https://mysite.com/.../289124335/default/mystyles.css
elgg_get_simplecache_url('mystyles.css');
```

Elgg automatically adds the magic numbers you see there for cache-busting and sets long-term expires headers on the returned file.

Avertissement : Elgg may decide to change the location or structure of the returned URL in a future release for whatever reason, and the cache-busting numbers change every time you flush Elgg's caches, so the exact URL is not stable by design.

With that in mind, here's a couple anti-patterns to avoid :
— Don't rely on the exact structure/location of this URL
— Don't try to generate the URLs yourself
— Don't store the returned URLs in a database

In your plugin's init function, register the css file :

```
elgg_register_css('mystyles', elgg_get_simplecache_url('mystyles.css'));
```

Then on the page you want to load the css, call :

```
elgg_load_css('mystyles');
```

Views and third-party assets

The best way to serve third-party assets is through views. However, instead of manually copy/pasting the assets into the right location in `/views/*`, you can map the assets into the views system via the `"views"` key in your plugin's `elgg-plugin.php` config file.

The views value must be a 2 dimensional array. The first level maps a viewtype to a list of view mappings. The secondary lists map view names to file paths, either absolute or relative to the Elgg root directory.

If you check your assets into source control, point to them like this :

```
<?php // mod/example/elgg-plugin.php
return [
    // view mappings
    'views' => [
        // viewtype
        'default' => [
            // view => /path/from/filesystem/root
            'js/jquery-ui.js' => __DIR__ . '/bower_components/jquery-ui/jquery-ui.min.
↪js',
        ],
    ],
];
```

To point to assets installed with composer, use install-root-relative paths by leaving off the leading slash :

```
<?php // mod/example/elgg-plugin.php
return [
    'views' => [
        'default' => [
            // view => path/from/install/root
            'js/jquery-ui.js' => 'vendor/bower-asset/jquery-ui/jquery-ui.min.js',
        ],
    ],
];
```

Elgg core uses this feature extensively, though the value is returned directly from `/engine/views.php`.

Note : You don't have to use Bower, Composer Asset Plugin, or any other script for managing your plugin's assets, but we highly recommend using a package manager of some kind because it makes upgrading so much easier.

Specifying additional views directories

In `elgg-plugin.php` you can also specify directories to be scanned for views. Just provide a view name prefix ending with `/` and a directory path (like above).

```
<?php // mod/file/elgg-plugin.php
return [
    'views' => [
        'default' => [
            'file/icon/' => __DIR__ . '/graphics/icons',
        ],
    ],
];
```

With the above, files found within the `icons` folder will be interpreted as views. E.g. the view `file/icon/general.gif` will be created and mapped to `mod/file/graphics/icons/general.gif`.

Note : This is a fully recursive scan. All files found will be brought into the views system.

Multiple paths can share the same prefix, just give an array of paths :

```
<?php // mod/file/elgg-plugin.php
return [
    'views' => [
        'default' => [
            'file/icon/' => [
                __DIR__ . '/graphics/icons',
                __DIR__ . '/more_icons', // processed 2nd (may override)
            ],
        ],
    ],
];
```

Viewtypes

You might be wondering : « Why `/views/default/hello/world.php` instead of just `/views/hello/world.php` ? ».

The subdirectory under `/views` determines the *viewtype* of the views below it. A viewtype generally corresponds to the output format of the views.

The default viewtype is assumed to be HTML and other static assets necessary to render a responsive web page in a desktop or mobile browser, but it could also be :

- RSS
- ATOM
- JSON
- Mobile-optimized HTML
- TV-optimized HTML
- Any number of other data formats

You can force Elgg to use a particular viewtype to render the page by setting the `view` input variable like so : `https://mysite.com/?view=rss`.

You could also write a plugin to set this automatically using the `elgg_set_viewtype()` function. For example, your plugin might detect that the page was accessed with an iPhone's browser string, and set the viewtype to `iphone` by calling :

```
elgg_set_viewtype('iphone');
```

The plugin would presumably also supply a set of views optimized for those devices.

Altering views via plugins

Without modifying Elgg's core, Elgg provides several ways to customize almost all output :

- You can *override a view*, completely changing the file used to render it.
- You can *extend a view* by prepending or appending the output of another view to it.
- You can *alter a view's inputs* by plugin hook.
- You can *alter a view's output* by plugin hook.

Overriding views

Views in plugin directories always override views in the core directory ; however, when plugins override the views of other plugins, *later plugins take precedent*.

For example, if we wanted to customize the `hello/world` view to use an `h2` instead of an `h1`, we could create a file at `/mod/example/views/default/hello/world.php` like this :

```
<h2>Hello, <?= $vars['name']; ?></h2>
```

Note : When considering long-term maintenance, overriding views in the core and bundled plugins has a cost : Upgrades may bring changes in views, and if you have overridden them, you will not get those changes.

You may instead want to alter *the input* or *the output* of the view via plugin hooks.

Note : Elgg caches view locations. This means that you should disable the system cache while developing with views. When you install the changes to a production environment you must flush the caches.

Extending views

There may be other situations in which you don't want to override the whole view, you just want to prepend or append some more content to it. In Elgg this is called *extending a view*.

For example, instead of overriding the `hello/world` view, we could extend it like so :

```
elgg_extend_view('hello/world', 'hello/greeting');
```

If the contents of `/views/default/hello/greeting.php` is :

```
<h2>How are you today?</h2>
```

Then every time we call `elgg_view('hello/world');`, we'll get :

```
<h1>Hello, World!</h1>
<h2>How are you today?</h2>
```

You can prepend views by passing a value to the 3rd parameter that is less than 500 :

```
// appends 'hello/greeting' to every occurrence of 'hello/world'
elgg_extend_view('hello/world', 'hello/greeting');

// prepends 'hello/greeting' to every occurrence of 'hello/world'
elgg_extend_view('hello/world', 'hello/greeting', 450);
```

All view extensions should be registered in your plugin's `init,system` event handler in `start.php`.

Altering view input

It may be useful to alter a view's `$vars` array before the view is rendered.

Before each view rendering the `$vars` array is filtered by the *plugin hook* `["view_vars", $view_name]`. Each registered handler function is passed these arguments :

- `$hook` - the string `"view_vars"`
- `$view_name` - the view name being rendered (the first argument passed to `elgg_view()`)
- `$returnvalue` - the modified `$vars` array
- `$params` - an array containing :
  - `vars` - the original `$vars` array, unaltered
  - `view` - the view name
  - `viewtype` - The *viewtype* being rendered

Altering view input example

Here we'll alter the default pagination limit for the comments view :

```
elgg_register_plugin_hook_handler('view_vars', 'page/elements/comments', 'myplugin_
↪alter_comments_limit');

function myplugin_alter_comments_limit($hook, $type, $vars, $params) {
    // only 10 comments per page
    $vars['limit'] = elgg_extract('limit', $vars, 10);
    return $vars;
}
```

Altering view output

Sometimes it is preferable to alter the output of a view instead of overriding it.

The output of each view is run through the *plugin hook* `["view", $view_name]` before being returned by `elgg_view()`. Each registered handler function is passed these arguments :

- `$hook` - the string `"view"`
- `$view_name` - the view name being rendered (the first argument passed to `elgg_view()`)
- `$result` - the modified output of the view
- `$params` - an array containing :
  - `viewtype` - The *viewtype* being rendered

To alter the view output, the handler just needs to alter `$returnvalue` and return a new string.

Altering view output example

Here we'll eliminate breadcrumbs that don't have at least one link.

```
elgg_register_plugin_hook_handler('view', 'navigation/breadcrumbs', 'myplugin_alter_
↪breadcrumb');

function myplugin_alter_breadcrumb($hook, $type, $returnvalue, $params) {
    // we only want to alter when viewtype is "default"
    if ($params['viewtype'] !== 'default') {
        return $returnvalue;
    }

    // output nothing if the content doesn't have a single link
    if (false === strpos($returnvalue, '<a ')) {
        return '';
    }

    // returning nothing means "don't alter the returnvalue"
}
```

Replacing view output completely

You can pre-set the view output by setting `$vars['__view_output']`. The value will be returned as a string. View extensions will not be used and the `view` hook will not be triggered.

```
elgg_register_plugin_hook_handler('view_vars', 'navigation/breadcrumbs', 'myplugin_no_
↪page_breadcrumbs');

function myplugin_no_page_breadcrumbs($hook, $type, $vars, $params) {
    if (elgg_in_context('pages')) {
        return ['__view_output' => ""];
    }
}
```

Note : For ease of use you can also use a already existing default hook callback to prevent output `\Elgg\Values::preventViewOutput`

Displaying entities

If you don't know what an entity is, *check this page out first*.

The following code will automatically display the entity in `$entity` :

```
echo elgg_view_entity($entity);
```

As you'll know from the data model introduction, all entities have a *type* (object, site, user or group), and optionally a subtype (which could be anything - "blog", "forumpost", "banana").

`elgg_view_entity` will automatically look for a view called `type/subtype`; if there's no subtype, it will look for `type/type`. Failing that, before it gives up completely it tries `type/default`.

RSS feeds in Elgg generally work by outputting the `object/default` view in the "rss" viewtype.

For example, the view to display a blog post might be `object/blog`. The view to display a user is `user/default`.

Full and partial entity views

`elgg_view_entity` actually has a number of parameters, although only the very first one is required. The first three are :

- `$entity` - The entity to display
- `$viewtype` - The viewtype to display in (defaults to the one we're currently in, but it can be forced - eg to display a snippet of RSS within an HTML page)
- `$full_view` - Whether to display a *full* version of the entity. (Defaults to `true`.)

This last parameter is passed to the view as `$vars['full_view']`. It's up to you what you do with it; the usual behaviour is to only display comments and similar information if this is set to true.

Listing entities

This is then used in the provided listing functions. To automatically display a list of blog posts (*see the full tutorial*), you can call :

```
echo elgg_list_entities([
    'type' => 'object',
    'subtype' => 'blog',
]);
```

This function checks to see if there are any entities; if there are, it first displays the `navigation/pagination` view in order to display a way to move from page to page. It then repeatedly calls `elgg_view_entity` on each entity before returning the result.

Note that `elgg_list_entities` allows the URL to set its `limit` and `offset` options, so set those explicitly if you need particular values (e.g. if you're not using it for pagination).

Elgg knows that it can automatically supply an RSS feed on pages that use `elgg_list_entities`. It initializes the `["head","page"]` plugin hook (which is used by the header) in order to provide RSS autodiscovery, which is why you can see the orange RSS icon on those pages in some browsers.

Entity listings will default try to load entity owners and container owners. If you want to prevent this you can turn this off.

```
echo elgg_list_entities([
    'type' => 'object',
    'subtype' => 'blog',

    // disable owner preloading
    'preload_owners' => false,
]);
```

See also *this background information on Elgg's database*.

If you want to show a message when the list does not contain items to list, you can pass a `no_results` message or `true` for the default message. If you want even more controle over the `no_results` message you can also pass a Closure (an anonymous function).

```
echo elgg_list_entities([
    'type' => 'object',
    'subtype' => 'blog',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'no_results' => elgg_echo('notfound'),
]);
```

Rendering a list with an alternate view

You can define an alternative view to render list items using `'item_view'` parameter.

In some cases, default entity views may be unsuitable for your needs. Using `item_view` allows you to customize the look, while preserving pagination, list's HTML markup etc.

Consider these two examples :

```
echo elgg_list_entities([
    'type' => 'group',
    'relationship' => 'member',
    'relationship_guid' => elgg_get_logged_in_user_guid(),
    'inverse_relationship' => false,
    'full_view' => false,
]);
```

```
echo elgg_list_entities([
    'type' => 'group',
    'relationship' => 'invited',
    'relationship_guid' => (int) $user_guid,
    'inverse_relationship' => true,
    'item_view' => 'group/format/invitationrequest',
]);
```

In the first example, we are displaying a list of groups a user is a member of using the default group view. In the second example, we want to display a list of groups the user was invited to.

Since invitations are not entities, they do not have their own views and can not be listed using `elgg_list_*`. We are providing an alternative item view, that will use the group entity to display an invitation that contains a group name and buttons to access or reject the invitation.

Rendering a list as a table

Since 2.3 you can render lists as tables. Set `$options['list_type'] = 'table'` and provide an array of TableColumn objects as `$options['columns']`. The service `elgg()->table_columns` provides several methods to create column objects based around existing views (like `page/components/column/*`), properties, or methods.

In this example, we list the latest `my_plugin` objects in a table of 3 columns : entity icon, the display name, and a friendly format of the time.

```
echo elgg_list_entities([
    'type' => 'object',
    'subtype' => 'my_plugin',

    'list_type' => 'table',
    'columns' => [
        elgg()->table_columns->icon(),
        elgg()->table_columns->getDisplayName(),
        elgg()->table_columns->time_created(null, [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'format' => 'friendly',
        ]),
    ],
]);
```

See the `Elgg\Views\TableColumn\ColumnFactory` class for more details on how columns are specified and rendered. You can add or override methods of `elgg()->table_columns` in a variety of ways, based on views, properties/methods on the items, or given functions.

Icons

Elgg has support for two kind of icons : generic icons to help with styling (eg. show delete icon) and Entity icons (eg. user avatar).

Generic icons

As of Elgg 2.0 the generic icons are based on the FontAwesome library. You can get any of the supported icons by calling `elgg_view_icon($icon_name, $vars);` where :

- `$icon_name` is the FontAwesome name (without `fa-`) for example `user`
- `$vars` is optional, for example you can set an additional class

`elgg_view_icon()` calls the view `output/icon` with the given icon name and outputs all the correct classes to render the FontAwesome icon. If you wish to replace an icon with another icon you can write a `view_vars`, `output/icon` hook to replace the icon name with your replacement.

For backwards compatibility some older Elgg icon names are translated to a corresponding FontAwesome icon.

Entity icons

To view an icon belowing to an Entity call `elgg_view_entity_icon($entity, $size, $vars);` where :

- `$entity` is the `ElggEntity` you wish to show the icon for
- `$size` is the requestes size. Default Elgg supports `large`, `medium`, `small`, `tiny` and `topbar` (`master` is also available, but don't use it)
- `$vars` in order to pass additional information to the icon view

`elgg_view_entity_icon()` calls a view in the order :

- `icon/<type>/<subtype>`
- `icon/<type>/default`
- `icon/default`

So if you wish to customize the layout of the icon you can overrule the corresponding view.

An example of displaying a user avatar is

```
// get the user
$user = elgg_get_logged_in_user_entity();

// show the small icon
echo elgg_view_entity_icon($user, 'small');

// don't add the user_hover menu to the icon
echo elgg_view_entity_icon($user, 'small', [
        'use_hover' => false,
]);
```

Related

Page structure best practice

Elgg pages have an overall pageshell and a main content area. In Elgg 1.0+, we've marked out a space « the canvas » for items to write to the page. This means the user always has a very consistent experience, while giving maximum flexibility to plugin authors for laying out their functionality.

Think of the canvas area as a big rectangle that you can do what you like in. We've created a couple of standard canvases for you :

- one column
- two column
- content
- widgets

are the main ones. You can access these with the function :

```
$canvas_area = elgg_view_layout($canvas_name, array(
  'content' => $content,
  'section' => $section
));
```

The content sections are passed as an `array` in the second parameter. The array keys correspond to sections in the layout, the choice of layout will determine which sections to pass. The array values contain the html that should be displayed in those areas. Examples of two common layouts :

```
$canvas_area = elgg_view_layout('one_column', array(
  'content' => $content
));
```

```
$canvas_area = elgg_view_layout('one_sidebar', array(
  'content' => $content,
  'sidebar' => $sidebar
));
```

You can then, ultimately, pass this into the `elgg_view_page` function :

```
echo elgg_view_page($title, $canvas_area);
```

You may also have noticed that we've started including a standard title area at the top of each plugin page (or at least, most plugin pages). This is created using the following wrapper function, and should usually be included at the top of the plugin content :

```
$start_of_plugin_content = elgg_view_title($title_text);
```

This will also display any submenu items that exist (unless you set the second, optional parameter to false). So how do you add submenu items ?

In your plugin_init function, include the following call :

```
if (elgg_get_context() == "your_plugin") {
   // add a site navigation item
   $item = new ElggMenuItem('identifier', elgg_echo('your_plugin:link'), $url);
   elgg_register_menu_item('page', $item);
}
```

The submenu will then automatically display when your page is rendered. The "identifier" is a machine name for the link, it should be unique per menu.

Simplecache

Voir aussi :

— *Performance*
— *Views*

The Simplecache is a mechanism designed to alleviate the need for certain views to be regenerated dynamically. Instead, they are generated once, saved as a static file, and served in a way that entirely bypasses the Elgg engine.

If Simplecache is turned off (which can be done from the administration panel), these views will be served as normal, with the exception of site CSS.

The criteria for whether a view is suitable for the Simplecache is as follows :
— The view must not change depending on who or when it is being looked at
— The view must not depend on variables fed to it (except for global variables like site URL that never change)

Regenerating the Simplecache

You can regenerate the Simplecache at any time by :
— Loading `/upgrade.php`, even if you have nothing to upgrade
— In the admin panel click on "Flush the caches"
— Enabling or disabling a plugin
— Reordering your plugins

Using the Simplecache in your plugins

Registering views with the Simplecache

You can register a view with the Simplecache with the following function at init-time :

```
elgg_register_simplecache_view($viewname);
```

Accessing the cached view

If you registered a JavaScript or CSS file with Simplecache and put in the view folder as `your_view.js` or `your_view.css` you can very easily get the url to this cached view by calling `elgg_get_simplecache_url($view)`. For example :

```
$js = elgg_get_simplecache_url('your_view.js');
$css = elgg_get_simplecache_url('your_view.css');
```

Page/elements/foot vs footer

`page/elements/footer` is the content that goes inside this part of the page :

```
<div class="elgg-page-footer">
        <div class="elgg-inner">
                <!-- page/elements/footer goes here -->
        </div>
</div>
```

It's content is visible to end users and usually where you would put a sitemap or other secondary global navigation, copyright info, powered by elgg, etc.

`page/elements/foot` is inserted just before the ending `</body>` tag and is mostly meant as a place to insert scripts that don't already work with `elgg_register_js(array('location' => 'footer'));` or `elgg_require_js('amd/module');`. In other words, you should never override this view and probably don't need to extend it either. Just use the `elgg_*_js` functions instead

3.3.32 Walled Garden

Elgg supports a « Walled Garden » mode. In this mode, almost all pages are restricted to logged in users. This is useful for sites that don't allow public registration.

Activating Walled Garden mode

To activate Walled Garden mode in Elgg, go to the Administration section. On the right sidebar menu, under the « Configure » section, expand « Settings, » then click on « Advanced. »

From the Advanced Settings page, find the option labelled « Restrict pages to logged-in users. » Enable this option, then click « Save » to switch your site into Walled Garden mode.

Exposing pages through Walled Gardens

Many plugins extend Elgg by adding pages. Walled Garden mode will prevent these pages from being viewed by logged out users. Elgg uses *plugin hook* to manage which pages are visible through the Walled Garden.

Plugin authors must register pages as public if they should be viewable through Walled Gardens :

- by setting `'walled' => false` in route configuration
- by responding to the `public_pages`, `walled_garden` plugin hook. The returned value is an array of regexp expressions for public pages.

The following code shows how to expose http://example.org/my_plugin/public_page through a Walled Garden. This assumes the plugin has registered a *route* for `my_plugin/public_page`.

```
// Preferred way
elgg_register_route('my_plugin:public_page', [
    'path' => '/my_plugin/public_page',
    'resource' => 'my_plugin/public_page',
    'walled' => false,
]);

// Legacy approach
elgg_register_plugin_hook_handler('public_pages', 'walled_garden', 'my_plugin_walled_
↪garden_public_pages');

function my_plugin_walled_garden_public_pages($hook, $type, $pages) {
   $pages[] = 'my_plugin/public_page';
   return $pages;
}
```

3.3.33 Web services

Build an HTTP API for your site.

Elgg provides a powerful framework for building web services. This allows developers to expose functionality to other web sites and desktop applications along with doing integrations with third-party web applications. While we call the API RESTful, it is actually a REST/RPC hybrid similar to the APIs provided by sites like Flickr and Twitter.

To create an API for your Elgg site, you need to do 4 things :
- enable the web services plugin
- expose methods
- setup API authentication
- setup user authentication

Additionally, you may want to control what types of authentication are available on your site. This will also be covered.

Contents

Security

It is crucial that the web services are consumed via secure protocols. Do not enable web services if your site is not served via HTTPs. This is especially important if you allow API key only authentication.

If you are using third-party tools that expose API methods, make sure to carry out a thorough security audit. You may want to make sure that API authentication is required for ALL methods, even if they require user authentication. Methods that do not require API authentication can be easily abused to spam your site.

Ensure that the validity of API keys is limited and provide mechanisms for your API clients to renew their keys.

Exposing methods

The function to use to expose a method is `elgg_ws_expose_function()`. As an example, let's assume you want to expose a function that echos text back to the calling application. The function could look like this

```
function my_echo($string) {
    return $string;
}
```

Since we are providing this function to allow developers to test their API clients, we will require neither API authentication nor user authentication. This call registers the function with the web services API framework :

```
elgg_ws_expose_function(
        "test.echo",
        "my_echo",
        [
                "string" => [
                        'type' => 'string',
                ]
        ],
        'A testing method which echos back a string',
        'GET',
        false,
        false
);
```

If you add this code to a plugin and then go to http://yoursite.com/services/api/rest/json/?method=system.api.list, you should now see your test.echo method listed as an API call. Further, to test the exposed method from a web browser, you could hit the url : http://yoursite.com/services/api/rest/json/?method=test.echo&string=testing and you should see JSON data like this :

```
{"status":0,"result":"testing"}
```

Plugins can filter the output of individual API methods by registering a handler for `'rest:output',$method` plugin hook.

Response formats

JSON is the default format, however XML and serialized PHP can be fetched by enabling the `data_views` plugin and substituting `xml` or `php` in place of `json` in the above URLs.

You can also add additional response formats by defining new viewtypes.

Parameters

Parameters expected by each method should be listed as an associative array, where the key represents the parameter name, and the value contains an array with `type`, `default` and `required` fields.

Values submitted with the API request for each parameter should match the declared type. API will throw on exception if validation fails.

Recognized parameter types are :
- `integer` (or `int`)
- `boolean` (or `bool`)
- `string`
- `float`
- `array`

Unrecognized types will throw an API exception.

You can use additional fields to describe your parameter, e.g. `description`.

```
elgg_ws_expose_function(
        'test.greet',
        'my_greeting',
        [
                'name' => [
                        'type' => 'string',
```

(suite sur la page suivante)

(suite de la page précédente)

```
                        'required' => true,
                        'description' => 'Name of the person to be greeted by the API
↪',
                ],
                'greeting' => [
                        'type' => 'string',
                        'required' => false,
                        'default' => 'Hello',
                        'description' => 'Greeting to be used, e.g. "Good day" or "Hi"
↪',
                ],
        ],
        'A testing method which greets the user with a custom greeting',
        'GET',
        false,
        false
);
```

Note : If a missing parameter has no default value, the argument will be `null`. Before Elgg v2.1, a bug caused later arguments to be shifted left in this case.

Receive parameters as associative array

If you have a large number of method parameters, you can force the execution script to invoke the callback function with a single argument that contains an associative array of parameter => input pairs (instead of each parameter being a separate argument). To do that, set `$assoc` to `true` in `elgg_ws_expose_function()`.

```
function greet_me($values) {
        $name = elgg_extract('name', $values);
        $greeting = elgg_extract('greeting', $values, 'Hello');
        return "$greeting, $name";
}

elgg_ws_expose_function(
        "test.greet",
        "greet_me",
        [
                "name" => [
                        'type' => 'string',
                ],
                "greeting" => [
                        'type' => 'string',
                        'default' => 'Hello',
                        'required' => false,
                ],
        ],
        'A testing method which echos a greeting',
        'GET',
        false,
        false,
        true // $assoc makes the callback receive an associative array
);
```

Note : If a missing parameter has no default value, `null` will be used.

API authentication

You may want to control access to some of the functions that you expose. Perhaps you are exposing functions in order to integrate Elgg with another open source platform on the same server. In that case, you only want to allow that other application access to these methods. Another possibility is that you want to limit what external developers have access to your API. Or maybe you want to limit how many calls a developer can make against your API in a single day.

In all of these cases, you can use Elgg's API authentication functions to control access. Elgg provides two built-in methods to perform API authentication : key based and HMAC signature based. You can also add your own authentication methods. The key based approach is very similar to what Google, Flickr, or Twitter. Developers can request a key (a random string) and pass that key with all calls that require API authentication. The keys are stored in the database and if an API call is made without a key or a bad key, the call is denied and an error message is returned.

Key-based authentication

As an example, let's write a function that returns the number of users that have viewed the site in the last x minutes.

```
function count_active_users($minutes=10) {
    $seconds = 60 * $minutes;
    $count = count(find_active_users($seconds, 9999));
    return $count;
}
```

Now, let's expose it and make the number of minutes an optional parameter :

```
elgg_ws_expose_function(
        "users.active",
        "count_active_users",
        [
                "minutes" => [
                        'type' => 'int',
                        'required' => false,
                ],
        ],
        'Number of users who have used the site in the past x minutes',
        'GET',
        true,
        false
);
```

This function is now available and if you check `system.api.list`, you will see that it requires API authentication. If you hit the method with a web browser, it will return an error message about failing the API authentication. To test this method, you need an API key. Fortunately, there is a plugin called apiadmin that creates keys for you. It is available in the Elgg plugin repository. It will return a public and private key and you will use the public key for this kind of API authentication. Grab a key and then do a GET request with your browser on this API method passing in the key string as the parameter `api_key`. It might look something like this : http://yoursite.com/services/api/rest/xml/?method=users.active&api_key=1140321cb56c71710c38feefdf72bc462938f59f.

Signature-based authentication

The *HMAC Authentication* is similar to what is used with OAuth or Amazon's S3 service. This involves both the public and private key. If you want to be very sure that the API calls are coming from the developer you think they are coming from and you want to make sure the data is not being tampered with during transmission, you would use this authentication method. Be aware that it is much more involved and could turn off developers when there are other sites out there with key-based authentication.

User authentication

So far you have been allowing developers to pull data out of your Elgg site. Now we'll move on to pushing data into Elgg. In this case, it is going to be done by a user. Maybe you have created a desktop application that allows your Users to post to the wire without going to the site. You need to expose a method for posting to the wire and you need to make sure that a user cannot post using someone else's account. Elgg provides a token-based approach for user authentication. It allows a user to submit their username and password in exchange for a token using the method `auth.gettoken`. This token can then be used for some amount of time to authenticate all calls to the API before it expires by passing it as the parameter `auth_token`. If you do not want to have your users trusting their passwords to 3rd-party applications, you can also extend the current capability to use an approach like OAuth.

Let's write our wire posting function :

```
function my_post_to_wire($text) {

    $text = substr($text, 0, 140);

    $access = ACCESS_PUBLIC;

    // returns guid of wire post
    return thewire_save_post($text, $access, "api");
}
```

Exposing this function is the same as the previous except we require user authentication and we're going to make this use POST rather than GET HTTP requests.

```
elgg_ws_expose_function(
        "thewire.post",
        "my_post_to_wire",
        [
                "text" => [
                        'type' => 'string',
                ],
        ],
        'Post to the wire. 140 characters or less',
        'POST',
        true,
        true
);
```

Please note that you will not be able to test this using a web browser as you did with the other methods. You need to write some client code to do this.

Building out your API

As soon as you feel comfortable with Elgg's web services API framework, you will want to step back and design your API. What sort of data are you trying to expose ? Who or what will be API users ? How do you want them to get access to authentication keys ? How are you going to document your API ? Be sure to take a look at the APIs created by popular Web 2.0 sites for inspiration. If you are looking for 3rd party developers to build applications using your API, you will probably want to provide one or more language-specific clients.

Determining the authentication available

Elgg's web services API uses a type of pluggable authentication module (PAM) architecture to manage how users and developers are authenticated. This provides you the flexibility to add and remove authentication modules. Do you want to not use the default user authentication PAM but would prefer using OAuth ? You can do this.

The first step is registering a callback function for the *rest, init* plugin hook :

```
register_plugin_hook('rest', 'init', 'rest_plugin_setup_pams');
```

Then in the callback function, you register the PAMs that you want to use :

```
function rest_plugin_setup_pams() {
    // user token can also be used for user authentication
    register_pam_handler('pam_auth_usertoken');

    // simple API key check
    register_pam_handler('api_auth_key', "sufficient", "api");

    // override the default pams
    return true;
}
```

When testing, you may find it useful to register the `pam_auth_session` PAM so that you can easily test your methods from the browser. Be careful not to use this PAM on a production site because it could open up your users to a CSRF attack.

Related

HMAC Authentication

Elgg's RESTful API framework provides functions to support a HMAC signature scheme for API authentication. The client must send the HMAC signature together with a set of special HTTP headers when making a call that requires API authentication. This ensures that the API call is being made from the stated client and that the data has not been tampered with.

The HMAC must be constructed over the following data :

- The public API key identifying you to the Elgg api server as provided by the APIAdmin plugin
- The private API Key provided by Elgg (that is companion to the public key)
- The current unix time in seconds
- A nonce to guarantee two requests the same second have different signatures
- URL encoded string representation of any GET variable parameters, eg `method=test.test&foo=bar`
- If you are sending post data, the hash of this data

Some extra information must be added to the HTTP header in order for this data to be correctly processed :

- **X-Elgg-apikey** - The public API key
- **X-Elgg-time** - Unix time used in the HMAC calculation

— **X-Elgg-none** - a random string
— **X-Elgg-hmac** - The HMAC as base64 encoded
— **X-Elgg-hmac-algo** - The algorithm used in the HMAC calculation - eg, sha1, md5 etc.

If you are sending POST data you must also send :

— **X-Elgg-posthash** - The hash of the POST data
— **X-Elgg-posthash-algo** - The algorithm used to produce the POST data hash - eg, md5
— **Content-type** - The content type of the data you are sending (if in doubt use application/octet-stream)
— **Content-Length** - The length in bytes of your POST data

Elgg provides a sample API client that implements this HMAC signature : send_api_call(). It serves as a good reference on how to implement it.

3.3.34 Widgets

Widgets are content areas that users can drag around their page to customize the layout. They can typically be customized by their owner to show more/less content and determine who sees the widget. By default Elgg provides plugins for customizing the profile page and dashboard via widgets.

Contents

Structure

To create a widget, create two views :

— `widgets/widget/edit`
— `widgets/widget/content`

`content.php` is responsible for all the content that will output within the widget. The `edit.php` file contains any extra edit functions you wish to present to the user. You do not need to add access level as this comes as part of the widget framework.

Note : Using HTML checkboxes to set widget flags is problematic because if unchecked, the checkbox input is omitted from form submission. The effect is that you can only set and not clear flags. The « input/checkboxes » view will not work properly in a widget's edit panel.

Register the widget

Once you have created your edit and view pages, you need to initialize the plugin widget.

The easiest way to do this is to add the `widgets` section to your `elgg-plugin.php` config file.

```
return [
        'widgets' => [
                'filerepo' => [
                        'context' => ['profile'],
                ],
        ]
];
```

Alternatively you can also use an function to add a widget. This is done within the plugins `init()` function.

```
// Add generic new file widget
elgg_register_widget_type([
    'id' => 'filerepo',
    'name' => elgg_echo('widgets:filerepo:name'),
    'description' => elgg_echo('widgets:filerepo:description'),
    'context' => ['profile'],
]);
```

Note : The only required attribute is the `id`.

Multiple widgets

It is possible to add multiple widgets for a plugin. You just initialize as many widget directories as you need.

```
// Add generic new file widget
elgg_register_widget_type([
    'id' => 'filerepo',
    'name' => elgg_echo('widgets:filerepo:name'),
    'description' => elgg_echo('widgets:filerepo:description'),
    'context' => ['profile'],
]);

// Add a second file widget
elgg_register_widget_type([
    'id' => 'filerepo2',
    'name' => elgg_echo('widgets:filerepo2:name'),
    'description' => elgg_echo('widgets:filerepo2:description'),
    'context' => ['dashboard'],
]);

// Add a third file widget
elgg_register_widget_type([
    'id' => 'filerepo3',
    'name' => elgg_echo('widgets:filerepo3:name'),
    'description' => elgg_echo('widgets:filerepo3:description'),
    'context' => ['profile', 'dashboard'],
]);
```

Make sure you have the corresponding directories within your plugin views structure :

```
'Plugin'
    /views
        /default
            /widgets
               /filerepo
                  /edit.php
                  /content.php
               /filerepo2
                  /edit.php
                  /content.php
               /filerepo3
                  /edit.php
                  /content.php
```

Magic widget name and description

When registering a widget you can omit providing a name and a description. If a translation in the following format is provided, they will be used. For the name : `widgets:<widget_id>:name` and for the description `widgets:<widget_id>:description`. If you make sure these translation are available in a translation file, you have very little work registering the widget.

```
elgg_register_widget_type(['id' => 'filerepo']);
```

How to restrict where widgets can be used

The widget can specify the context that it can be used in (just profile, just dashboard, etc.).

```
elgg_register_widget_type([
    'id' => 'filerepo',
    'context' => ['profile', 'dashboard', 'other_context'],
]);
```

Allow multiple widgets on the same page

By default you can only add one widget of the same type on the page. If you want more of the same widget on the page, you can specify this when registering the widget :

```
elgg_register_widget_type([
    'id' => 'filerepo',
    'multiple' => true,
]);
```

Register widgets in a hook

If, for example, you wish to conditionally register widgets you can also use a hook to register widgets.

```
function my_plugin_init() {
    elgg_register_plugin_hook_handler('handlers', 'widgets', 'my_plugin_conditional_
↪widgets_hook');
}

function my_plugin_conditional_widgets_hook($hook, $type, $return, $params) {
    if (!elgg_is_active_plugin('file')) {
        return;
    }

    $return[] = \Elgg\WidgetDefinition::factory([
        'id' => 'filerepo',
    ]);

    return $return;
}
```

Modify widget properties of existing widget registration

If, for example, you wish to change the allowed contexts of an already registered widget you can do so by re-registering the widget with `elgg_register_widget_type` as it will override an already existing widget definition. If you want even more control you can also use the `handlers, widgets` hook to change the widget definition.

```
function my_plugin_init() {
    elgg_register_plugin_hook_handler('handlers', 'widgets', 'my_plugin_change_widget_
↪definition_hook');
}

function my_plugin_change_widget_definition_hook($hook, $type, $return, $params) {
    foreach ($return as $key => $widget) {
        if ($widget->id === 'filerepo') {
            $return[$key]->multiple = false;
        }
    }

    return $return;
}
```

Default widgets

If your plugin uses the widget canvas, you can register default widget support with Elgg core, which will handle everything else.

To announce default widget support in your plugin, register for the `get_list, default_widgets` plugin hook :

```
elgg_register_plugin_hook_handler('get_list', 'default_widgets', 'my_plugin_default_
↪widgets_hook');
```

In the plugin hook handler, push an array into the return value defining your default widget support and when to create default widgets. Arrays require the following keys to be defined :

— name - The name of the widgets page. This is displayed on the tab in the admin interface.

— widget_context - The context the widgets page is called from. (If not explicitly set, this is your plugin's id.)
— widget_columns - How many columns the widgets page will use.
— event - The Elgg event to create new widgets for. This is usually `create`.
— entity_type - The entity type to create new widgets for.
— entity_subtype - The entity subtype to create new widgets for. The can be ELGG_ENTITIES_ANY_VALUE to create for all entity types.

When an object triggers an event that matches the event, entity_type, and entity_subtype parameters passed, Elgg core will look for default widgets that match the widget_context and will copy them to that object's owner_guid and container_guid. All widget settings will also be copied.

```
function my_plugin_default_widgets_hook($hook, $type, $return, $params) {
    $return[] = array(
        'name' => elgg_echo('my_plugin'),
        'widget_context' => 'my_plugin',
        'widget_columns' => 3,

        'event' => 'create',
        'entity_type' => 'user',
        'entity_subtype' => ELGG_ENTITIES_ANY_VALUE,
    );

    return $return;
}
```

3.4 Tutoriels

Suivez toutes les étapes requises pour personnaliser Elgg.

Les instructions sont suffisamment détaillées pour que vous n'ayez pas besoin de beaucoup d'expérience de développement avec Elgg.

3.4.1 Hello world

Ce tutoriel vous montre comment créer un nouveau plugin qui consiste en une nouvelle page qui affiche le texte « Hello world ».

Avant toute chose, vous devez *installer Elgg*.

Dans ce tutoriel, nous allons supposer que l'URL de votre site est `https://elgg.example.com`.

Tout d'abord, créez un répertoire qui va contenir les fichiers du plugin. Il devrait être placé dans le répertoire `mod/` situé dans le répertoire d'installation d'Elgg. Dans notre cas, créez `mod/hello/`.

Fichier Manifest

Elgg a besoin que votre plugin dispose d'un fichier manifest qui contient des informations sur le plugin. A cette fin, créez un fichier nommé `manifest.xml` dans le répertoire de votre plugin, et collez ce code dedans :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
    <name>Hello world</name>
    <id>hello</id>
    <author>Your Name Here</author>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <version>0.1</version>
    <description>Hello world, testing.</description>
    <requires>
        <type>elgg_release</type>
        <version>2.0</version>
    </requires>
</plugin_manifest>
```

Voici le minimum d'informations qui doivent être présentes dans un fichier manifest :

- `<name>` est le nom du plugin tel qu'il sera affiché
- `<id>` doit correspondre au nom du répertoire que vous venez de créer
- `<requires>` doit indiquer la version minimum d'Elgg dont votre plugin a besoin
- `<author>`, `<version>` et `<description>` devraient avoir des valeurs appropriées mais peuvent être renseignés librement

Initialiseur

Puis créez `start.php` dans le répertoire `mod/hello/` et copiez ce code dedans :

```
<?php

elgg_register_event_handler('init', 'system', 'hello_world_init');

function hello_world_init() {

}
```

Le code ci-dessus indique à Elgg qu'il devrait appeler la fonction `hello_world_init()` une fois que le coeur du système Elgg est initialisé.

Enregistrer une route

L'étape suivante est d'enregistrer une route dont l'objectif est de traiter les requêtes que les utilsiateurs font vers l'URL `https://elgg.example.com/hello`.

Modifiez `elgg-plugin.php` pour qu'il ressemble à ceci :

```
<?php

return [
    'routes' => [
                'default:hello' => [
                        'path' => '/hello',
                        'resource' => 'hello',
                ],
        ],
];
```

Cet enregistrement indique à Elgg d'appeler la vue ressource `hello` quand un utilisateur navigue vers `https://elgg.example.com/hello`.

Voir le fichier

Créez `mod/hello/views/default/resources/hello.php` avec ce contenu :

```
<?php

$body = elgg_view_layout('content', [
    'title' => 'Hello world!',
    'content' => 'My first page!',
    'filter' => '',
]);

echo elgg_view_page('Hello', $body);
```

Le code crée un tableau de paramètres à passer à la fonction `elgg_view_layout()`, comprenant :

- Le titre de la page
- Le contenu de la page
- Un filter qui est laissé vide puisqu'il n'y a pour le moment rien à filtrer

Ceci crée l'agencement général (layout) de base pour la page. Cet agencement est ensuite passé à travers `elgg_view_page()` qui assemble et génère la page complète.

Dernière étape

Pour terminer, activez le plugin depuis la page d'administration d'Elgg : *https ://elgg.example.com/admin/plugins`* (le nouveau plugin apparaît en bas)

Vous pouvez maintenant vous rendre sur l'adresse `https://elgg.example.com/hello/` et vous devriez voir votre nouvelle page !

3.4.2 Personnaliser la page d'accueil

Pour remplacer la page d'accueil, surchargez simplement la vue d'Elgg `resources/index` en créant un fichier à `/views/default/resources/index.php`.

Toute sortie de cette vue deviendra votre nouvelle page d'accueil.

Vous pouvez prendre une approche similaire avec n'importe quelle autre page d'Elgg ou des plugins officiels.

3.4.3 Construire un plugin de Blog

Ce tutorial va vous apprendre à créer un plugin de blog simple. Les fonctionnalités de base du blog seront de créer des articles, les enregistrer et les afficher. Le plugin duplique des fonctionnalités présentes dans le plugin `blog` empaqueté. Vous pouvez désactiver le plugin `blog` empaqueté si vous le souhaitez, mais ce n'est pas nécessaire dans la mesure où les fonctionnalités ne seront pas en conflit les unes avec les autres.

Contents

Prérequis :
— *Installation d'Elgg*

Créez le répertoire du plugin et le fichier manifest

Tout d'abord, choisissez un nom simple et descriptif pour votre plugin. Dans ce tutoriel, le nom sera `my_blog`. Puis créez un répertoire pour votre plugin dans le répertoire `/mod/` qui se trouve dans le répertoire d'installation d'Elgg. D'autres plugins sont également situés dans `/mod/`. Dans notre cas, le nom de ce répertoire devrait être `/mod/my_blog/`. Ce répertoire constitue la racine de votre plugin, et tous les fichiers que vous allez créer pour votre nouveau plugin vont l'être quelque part dans ce répertoire.

Ensuite, créez le fichier `manifest.xml` à la racine du plugin :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
    <name>My Blog</name>
    <id>my_blog</id>
    <author>Your Name Here</author>
    <version>0.1</version>
    <description>Adds blogging capabilities.</description>
    <requires>
        <type>elgg_release</type>
        <version>2.0</version>
    </requires>
</plugin_manifest>
```

Voyez la documentation sur les *Plugins* pour plus d'informations sur le fichier manifest.

Créez le formulaire pour la création d'un nouvel article de blog

Créez un fichier */mod/my_blog/views/default/forms/my_blog/save.php`* qui contiendra le corps du formulaire. Le formulaire doit avoir des champs de saisie pour le titre, le corps et les tags de l'article de blog my_blog. Il n'est pas nécessaire d'ajouter les balises <form>.

```
echo elgg_view_field([
    '#type' => 'text',
    '#label' => elgg_echo('title'),
    'name' => 'title',
    'required' => true,
]);

echo elgg_view_field([
    '#type' => 'longtext',
    '#label' => elgg_echo('body'),
    'name' => 'body',
    'required' => true,
]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
echo elgg_view_field([
    '#type' => 'tags',
    '#label' => elgg_echo('tags'),
    '#help' => elgg_echo('tags:help'),
    'name' => 'tags',
]);

$submit = elgg_view_field(array(
    '#type' => 'submit',
    '#class' => 'elgg-foot',
    'value' => elgg_echo('save'),
));
elgg_set_form_footer($submit);
```

Notez comment le formulaire appelle `elgg_view_field()` pour afficher les champs de saisie. Cette fonction d'aide permet de maintenir la cohérence dans les balises des champs de de saisie, et est utilisée comme raccourci pour afficher des éléments des champs, tels que le label, l'aide et le champs de saisie. Voir la *Forms + Actions*.

Vous pouvez voir une liste complète des vues d'entrée dans le répertoire `/vendor/elgg/elgg/views/default/input/`.

Il est recommandé que vous rendiez votre plugin traduisible en utilisant `elgg_echo()` à chaque fois qu'une chaîne de texte sera affichée à l'utilisateur. Lisez-en plus sur l'*Internationalisation*.

Créez une page pour écrire les articles de blog

Créez le fichier `/mod/my_blog/views/default/resources/my_blog/add.php`. Cette page va afficher le formulaire que vous avez créé dans la section précédente.

```
<?php
// make sure only logged in users can see this page
gatekeeper();

// set the title
$title = "Create a new my_blog post";

// start building the main column of the page
$content = elgg_view_title($title);

// add the form to the main column
$content .= elgg_view_form("my_blog/save");

// optionally, add the content for the sidebar
$sidebar = "";

// layout the page
$body = elgg_view_layout('one_sidebar', array(
   'content' => $content,
   'sidebar' => $sidebar
));

// draw the page, including the HTML wrapper and basic page layout
echo elgg_view_page($title, $body);
```

La fonction `elgg_view_form("my_blog/save")` affiche le formulaire que vous avez créé dans al section précédente. Elle ajoute automatiquement au formulaire la balise `<form>` avec les attributs nécessaires ainsi que des

jetons anti-csrf.

L'action du formulaire sera "<?= elgg_get_site_url() ?>action/my_blog/save".

Créez le fichier d'action pour enregistrer l'article de blog

Le fichier d'action va enregistrer l'article de blog my_blog dans la base de données. Créez le fichier /mod/my_blog/actions/my_blog/save.php :

```
<?php
// get the form inputs
$title = get_input('title');
$body = get_input('body');
$tags = string_to_tag_array(get_input('tags'));

// create a new my_blog object and put the content in it
$blog = new ElggObject();
$blog->title = $title;
$blog->description = $body;
$blog->tags = $tags;

// the object can and should have a subtype
$blog->subtype = 'my_blog';

// for now, make all my_blog posts public
$blog->access_id = ACCESS_PUBLIC;

// owner is logged in user
$blog->owner_guid = elgg_get_logged_in_user_guid();

// save to database and get id of the new my_blog
$blog_guid = $blog->save();

// if the my_blog was saved, we want to display the new post
// otherwise, we want to register an error and forward back to the form
if ($blog_guid) {
   system_message("Your blog post was saved.");
   forward($blog->getURL());
} else {
   register_error("The blog post could not be saved.");
   forward(REFERER); // REFERER is a global variable that defines the previous page
}
```

Comme vous pouvez le voir dans le code ci-dessus, les objets Elgg disposent de plusieurs champs intégrés. Le titre de l'article my_blog est conservé dans le champs `title` tandis que le contenu est conservé dans le champs `description`. Il y a également un champs pour des tags, qui sont stockés sous la forme de métadonnées.

Les objets dans Elgg sont une sous-classe de quelque chose appelé « entité » (entity). Les utilisateurs, sites et groupes sont également des sous-classes d'entité. Le sous-type (subtype) d'une entité permet un contrôle granulaire pour les listings et l'affichage, c'est pourquoi chaque entité devrait avoir un sous-type. Dans ce tutoriel, le sous-type « `my_blog` » permet d'identifier un article my_blog post, mais toute chaîne de caractères alphanumérique peut constituer un sous-type valide. Lorsque vous choisissez des sous-types, veuillez vous assurer d'en choisir un qui soit cohérent avec votre plugin.

La méthode `getURL` charge l'URL du nouvel article. Il est recommandé de surcharger cette méthode. La surcharge sera faite dans le fichier `start.php`.

Créer elgg-plugin.php

Le fichier `/mod/my_blog/elgg-plugin.php` est utilisé pour déclarer diverses fonctionnalités du plugin. Il peut, par exemple, être utilisé pour configurer des entités, des actions, des widgets et des routes.

```
<?php

return [
        'entities' => [
                [
                        'type' => 'object',
                        'subtype' => 'my_blog',
                        'searchable' => true,
                ],
        ],
        'actions' => [
                'my_blog/save' => [],
        ],
        'routes' => [
                'view:object:blog' => [
                        'path' => '/my_blog/view/{guid}/{title?}',
                        'resource' => 'my_blog/view',
                ],
                'add:object:blog' => [
                        'path' => '/my_blog/add/{guid?}',
                        'resource' => 'my_blog/add',
                ],
                'edit:object:blog' => [
                        'path' => '/my_blog/edit/{guid}/{revision?}',
                        'resource' => 'my_blog/edit',
                        'requirements' => [
                                'revision' => '\d+',
                        ],
                ],
        ],
];
```

Créez le fichier start.php

Le fichier `/mod/my_blog/start.php` doit enregistrer un hook pour surcharger la génération d'URL.

```
<?php

function my_blog_init() {
    // register a hook handler to override urls
    elgg_register_plugin_hook_handler('entity:url', 'object', 'my_blog_set_url');
}

return function() {
    // register an initializer
    elgg_register_event_handler('init', 'system', 'my_blog_init');
}
```

L'enregistrement de l'action de sauvegarde va la rendre disponible via `/action/my_blog/save`. Par défaut, toutes les actions ne sont disponibles qu'aux utilisateurs identifiés. Si vous souhaitez rendre une action disponible aux seuls administrateurs ou l'ouvrir y compris à des visiteurs non identifiés, vous pouvez passer respectivement "admin" ou "public" comme troisième paramètre de `elgg_register_action`.

La fonction de surcharge d'URL va extraire l'ID de l'entité et l'utiliser pour créer une URL simple pour la page qui va afficher cette entité. Dans ce cas l'entité devrait bien sûr être un article my_blog. Ajoutez cette fonction à votre fichier `start.php` :

```
function my_blog_set_url($hook, $type, $url, $params) {
    $entity = $params['entity'];
    if (elgg_instanceof($entity, 'object', 'my_blog')) {
        return "my_blog/view/{$entity->guid}";
    }
}
```

Le gestionnaire de page permet de servir la page qui génère le formulaire et la page qui affiche l'article. La prochaine section va montrer comment créer la page qui affiche l'article. Ajoutez cette fonction à votre fichier *start.php*` :

```
function my_blog_page_handler($segments) {
    if ($segments[0] == 'add') {
        echo elgg_view_resource('my_blog/add');
        return true;
    }

    else if ($segments[0] == 'view') {
        $resource_vars['guid'] = elgg_extract(1, $segments);
        echo elgg_view_resource('my_blog/view', $resource_vars);
        return true;
    }

    return false;
}
```

La variable `$segments` contient les différentes parties de l'URL une fois séparées par /.

Les fonctions de gestionnaires de pages doivent retourner `true` ou `false`. `true` signifie que la page existe et a bien été traitée par le gesitonnaire de pages. `false` signifie que la page n'existe pas et que l'utilisateur va être redirigé vers la page d'erreur 404 du site (la page demandée n'existe pas ou n'a pas été trouvée). Dans cet exemple, l'URL doit contenir `/my_blog/add` ou `/my_blog/view/id` où id est un identifiant valide d'une entité avec le sous-type `my_blog`. Vous trouverez plus d'informations sur les gestionnaires de pages sur *Page handler*.

Créez une page pour afficher l'article de blog

Pour pouvoir afficher un article my_blog sur sa propre page, vous devez créer une page d'affichage. Créez le fichier `/mod/my_blog/views/default/resources/my_blog/view.php` :

```
<?php

// get the entity
$guid = elgg_extract('guid', $vars);
$my_blog = get_entity($guid);

// get the content of the post
$content = elgg_view_entity($my_blog, array('full_view' => true));

$params = [
    'title' => $my_blog->getDisplayName(),
    'content' => $content,
    'filter' => '',
];
```

(suite sur la page suivante)

(suite de la page précédente)

```
$body = elgg_view_layout('content', $params);

echo elgg_view_page($my_blog->getDisplayName(), $body);
```

Cette page a beaucoup de points communs avec la page `add.php`. Les principales différences sont que les informations sont extraires depuis l'entité my_blog, et qu'au lieu d'afficher un formulaire, la fonction `elgg_view_entity` est appelée. Cette fonction donne les informations de l'entité à ce qu'on appelle la vue de l'objet.

Créez la vue de l'objet

Quand `elgg_view_entity` est appelé ou quand des articles my_blogs sont affichés dans une liste par exemple, la vue de l'objet va générer le contenu approprié. Créez le fichier `/mod/my_blog/views/default/object/my_blog.php` :

```
<?php

echo elgg_view('output/longtext', array('value' => $vars['entity']->description));
echo elgg_view('output/tags', array('tags' => $vars['entity']->tags));
```

Comme vous pouvez le voir dans la section précédente, chaque article my_blog est passé à la vue de l'objet sous la forme `$vars['entity']`. (`$vars` est un tableau utilisé dans le système de vues pour passer des variables à une vue.)

La dernière ligne prend les tags de l'article my_blog et les affiche automatiquement sous la forme d'une série de liens cliquables. La recherche est gérée automatiquement.

(Si vous vous interrogez sur le « `default` » dans `/views/default/`, vous pouvez créer des vues alternatives. RSS, OpenDD, FOAF, mobile et d'autres sont divers types de vues valides.)

Tester le plugin

Rendez-vous sur la page d'administration d'Elgg, listez les plugins, et activez le plugin `my_blog`.

La page pour créer un nouvel article my_blog devrait désormais être accessible via `https://elgg.example.com/my_blog/add`, et après avoir enregistré l'article, vous devriez le voir affiché sur sa propre page.

Afficher une liste des articles de blog

Créons également une page qui liste les entrées my_blog qui ont été créées

Créez `/mod/my_blog/views/default/resources/my_blog/all.php` :

```
<?php
$titlebar = "All Site My_Blogs";
$pagetitle = "List of all my_blogs";

$body = elgg_list_entities(array(
    'type' => 'object',
    'subtype' => 'my_blog',
));

$body = elgg_view_title($pagetitle) . elgg_view_layout('one_column', array('content'␣
↪=> $body));
```

(suite sur la page suivante)

(suite de la page précédente)

```
echo elgg_view_page($titlebar, $body);
```

La fonction `elgg_list_entities` prend les derniers articles my_blog et les passe à la vue d'affichage de l'objet. Notez que cette fonction ne retourne que les articles que l'utilisateur a le droit de voir, aussi le contrôle d'accès est géré de manière transparente. La fonction (et ses cousines) gère également de manière transparente la pagination, et crée même un flux RSS pour votre my_blogs si vous avez défini cette vue.

La fonction de liste peut également restreindre les articles my_blog à ceux d'un utilisateur spécifique. Par exemple, la fonction `elgg_get_logged_in_user_guid` récupère l'identifiant unique (GUID) de l'utilisateur connecté, et en le passant à `elgg_list_entities` la liste n'affichera que les articles de l'utilisateur connecté :

```
echo elgg_list_entities(array(
    'type' => 'object',
    'subtype' => 'my_blog',
    'owner_guid' => elgg_get_logged_in_user_guid()
));
```

Puis vous devez modifier votre gestionnaire de page my_blog pour récupérer la nouvelle page quand l'URL est `/my_blog/all`. Modifiez la fonction `my_blog_page_handler` dans `start.php` pour qu'elle ressemble à ceci :

```
function my_blog_page_handler($segments) {
    switch ($segments[0]) {
        case 'add':
           echo elgg_view_resource('my_blog/add');
           break;

        case 'view':
            $resource_vars['guid'] = elgg_extract(1, $segments);
            echo elgg_view_resource('my_blog/view', $resource_vars);
            break;

        case 'all':
        default:
           echo elgg_view_resource('my_blog/all');
           break;
    }

    return true;
}
```

Maintenant, si l'URL contient `/my_blog/all`, l'utilisateur verra une page « Tous les My_Blogs du site ». Grâce au cas par défaut, la liste de tous les my_blogs sera également affichée si l'URL n'est pas valide, comme par exemple``/my_blog`` ou `/my_blog/xyz`.

Vous pouvez aussi choisir de mettre à jour la vue de l'objet pour gérer différents types d'affichage, faute de quoi la liste de tous les my_blogs va également afficher le contenu complet de tous les my_blogs. Modifiez `/mod/my_blog/views/default/object/my_blog.php` pour qu'il ressemble à ceci :

```
<?php
$full = elgg_extract('full_view', $vars, FALSE);

// full view
if ($full) {
    echo elgg_view('output/longtext', array('value' => $vars['entity']->description));
    echo elgg_view('output/tags', array('tags' => $vars['entity']->tags));
```

(suite sur la page suivante)

(suite de la page précédente)

```
// list view or short view
} else {
    // make a link out of the post's title
    echo elgg_view_title(
        elgg_view('output/url', array(
            'href' => $vars['entity']->getURL(),
            'text' => $vars['entity']->getDisplayName(),
            'is_trusted' => true,
    )));
    echo elgg_view('output/tags', array('tags' => $vars['entity']->tags));
}
```

Désormais, si `full_view` est à `true` (tel qu'il avait été préalablement défini dans *cette section*), la vue de l'objet va afficher le contenu et les tags de l'article (le titre est affiché par `view.php`). Sinon la vue de l'objet ne va afficher que le titre et les tags de l'article.

FIN

Il y a tant d'autres choses qui peuvent être faites, mais espérons que ceci vous donne une bonne idée de comment démarrer.

3.4.4 Intégrer un éditeur de texte visuel (Rich Text Editor)

Construisez votre propre plugin wysiwyg.

Elgg est distribué avec un plugin pour CKEditor, et précédemment distribué avec le support de TinyMCE. Cependant, s'il y a un éditeur wysiwyg que vous préférez, vous pourriez utiliser ce tutoriel pour construire le vôtre.

Tous les formulaires dans Elgg devraient essayer d'utiliser les vues de saisie situées dans `views/default/input`. Si ces vues sont utilisées, il est plus aisé pour les auteurs de plugins de remplacer une vue, ici `input/longtext`, par leur vue avec wysiwyg.

Ajoutez la bibliothèque de code WYSIWYG

Maintenant vous devez charger TinyMCE dans un répertoire de votre plugin. Nous recommandons vivement que vous utilisiez `composer` pour gérer les bibliothèques tierces, car il est beaucoup plus facile de gérer la maintenance et les montées de version de cette manière

```
.. code-block:: sh
```

composer nécessite bower-asset/tinymce

Dites à Elgg quand et comment charger TinyMCE

Maintenant que vous avez :
- créé le fichier start
- initialisé le plugin
- chargé le code wysiwyg

Il est temps de dire à Elgg comment appliquer TinyMCE aux champs de saisie de texte.

Nous allons faire ceci en étendant la vue input/longtext et en incluant un peu de JavaScript. Créez une vue tinymce/longtext et ajoutez le code suivant :

```
<?php

    /**
     * Elgg long text input with the tinymce text editor intacts
     * Displays a long text input field
     *
     * @package ElggTinyMCE
     *
     *
     */

?>
<!-- include tinymce -->
<script language="javascript" type="text/javascript" src="<?php echo $vars['url']; ?>
↪mod/tinymce/tinymce/jscripts/tiny_mce/tiny_mce.js"></script>
<!-- intialise tinymce, you can find other configurations here http://wiki.moxiecode.
↪com/examples/tinymce/installation_example_01.php -->
<script language="javascript" type="text/javascript">
   tinyMCE.init({
    mode : "textareas",
    theme : "advanced",
    theme_advanced_buttons1 : "bold,italic,underline,separator,strikethrough,
↪justifyleft,justifycenter,justifyright, justifyfull,bullist,numlist,undo,redo,link,
↪unlink,image,blockquote,code",
    theme_advanced_buttons2 : "",
    theme_advanced_buttons3 : "",
    theme_advanced_toolbar_location : "top",
    theme_advanced_toolbar_align : "left",
    theme_advanced_statusbar_location : "bottom",
    theme_advanced_resizing : true,
    extended_valid_elements : "a[name|href|target|title|onclick],
↪img[class|src|border=0|alt|title|hspace|vspace|width|height|align|onmouseover|onmouseout|name],
↪
hr[class|width|size|noshade],font[face|size|color|style],span[class|align|style]"
});
</script>
```

Puis, dans la fonction init de votre plugin, étendez la vue input/longtext

```
function tinymce_init() {
    elgg_extend_view('input/longtext', 'tinymce/longtext');
}
```

Et voilà ! Désormais chaque fois que quelqu'un utilise input/longtext, TinyMCE sera chargé et appliqué à cette zone de texte.

3.4.5 Widget basique

Créez un widget qui va afficher “Hello, World !” ainsi que n’importe quel texte souhaité par l’utilisateur.

Dans Elgg, les widgets sont ces composants que vous pouvez déplacer sur votre page de profil ou le tableau de bord d’administration.

Ce tutoriel suppose que vous êtes familier(ère) des concepts de base d’Elgg tels que :
- *Views*
- *Plugins*

Vous devriez les revoir si cela devient confus en cours de route.

Contents

Ajouter le code de la vue du widget

Elgg scanne automatiquement certains répertoires des plugins pour trouver des fichiers spécifiques. Les vues *Views* facilitent l’ajout de votre propre code d’affichage, ou la possibilité de faire d’autres choses telles que surcharger le comportement par défaut d’Elgg. Pour le moment, nous allons simplement ajouter le code d’affichage pour votre widget. Créez un fichier `/views/default/widgets/helloworld/content.php`. “helloworld” sera le nom de votre widget à l’intérieur du plugin hello. Dans ce fichier ajoutez le code :

```
<?php

echo "Hello, world!";
```

Ceci va ajouter ces mots au canevas du widget lorsqu’il sera rendu. Elgg se charge de charger le widget.

Enregistrer votre widget

Elgg a besoin qu’on lui indique explicitement que le plugin contient un widget pour qu’il vérifie le répertoire des vues du widget. Ceci est fait en appelant la fonction elgg_register_widget_type(). Modifiez `/start.php` et ajoutez dedans ces lignes :

```
<?php

function hello_init() {
    elgg_register_widget_type([
        'id' => 'helloworld',
        'name' => 'Hello, world!',
        'description' => 'The "Hello, world!" widget',
    ]);
}

return function() {
    elgg_register_event_handler('init', 'system', 'hello_init');
}
```

Rendez-vous maintenant sur votre page de profil avec un navigateur web et ajoutez le widget “hello, world”. Il devrait afficher “Hello, world !”.

Note : Pour de vrais widgets, c'est toujours une bonne idée de respecter *Internationalization*.

Permettre les personnalisation par l'utilisateur

Cliquez sur le lien d'adition dans la barre d'outils du widget que vous avez créé. Vous allez noter que le seul contrôle qu'il vous offre par défaut est sur le niveau d'accès (sur qui peut voir le widget).

Supposez que vous voulez permettre à l'utilisateur de contrôler quel message d'accueil est affiché dans le widget. De la même manière qu'Elgg charge automatiquement `content.php` pour afficher un widget, il charge `edit.php` quand un utilisateur tente de modifier un widget. Ajoutez le code suivant dans `/views/default/widgets/helloworld/edit.php` :

```
<div>
    <label>Message:</label>
    <?php
        //This is an instance of the ElggWidget class that represents our widget.
        $widget = $vars['entity'];

        // Give the user a plain text box to input a message
        echo elgg_view('input/text', array(
            'name' => 'params[message]',
            'value' => $widget->message,
            'class' => 'hello-input-text',
        ));
    ?>
</div>
```

Notez la relation entre les valeurs passées aux champs "name" (nom) et "value" (valeur) d'input/text. Le nom du champ de la boîte de saisie de texte est `params[message]` parce qu'Elgg va automatiquement gérer les variables des widgets placées dans le tableau `params`. Le nom de la variable PHP correspondante sera `message`. Si nous voulions utiliser le champ `greeting` au lieu de `message` nous passerions respectivement les valeurs `params[greeting]` et `$widget->greeting`.

La raison pour laquelle nous définissons l'option "value" du tableau est que ceci indique à la vue d'édition de se souvenir de ce que l'utilisateur a saisi la dernière fois qu'il a modifié la valeur du texte du message.

Maintenant pour afficher le message de l'utilisateur nous devons modifier content.php pour qu'il utilise cette variable *message*. Editez `/views/default/widgets/helloworld/content.php` et modifiez-le pour :

```
<?php

$widget = $vars['entity'];

// Always use the corresponding output/* view for security!
echo elgg_view('output/text', array('value' => $widget->message));
```

Vous devriez maintenant pouvoir saisir un message dans la boîte de texte et le voir apparaître dans le widget.

3.5 Docs de conception

Acquérez une compréhension profonde de comment fonctionne Elgg et de pourquoi il est construit de cette manière.

3.5.1 Actions

Les actions sont la première modalité d'interaction des utilisateurs avec un site Elgg.

Aperçu

Dans Elgg, une action est le code qui est exécuté pour faire des modifications dans la base de données quand un utilisateur effectue quelque chose. Par exemple, le fait de se connecter, de publier un commentaire, ou de créer un article de blog sont des actions. Le script d'action traite le processus d'entrée, effectue les modifications appropriées dans la base de données, et fournit un retour à l'utilisateur à propos de l'action.

Gestionnaire d'action

Les actions sont enregistrées durant le processus de démarrage (boot) en appelant `elgg_register_action()`. Toutes les URLs d'action commencent par `action/` et sont servies par le contrôleur front end d'Elgg à travers le service action. Cette approche est différente d'application PHP traditionnelles qui envoient l'information à un fichier spécifique. Le service action réalise des *CSRF vérifications de sécurité*, et appelle le fichier de script d'action enregistré, puis renvoient éventuellement l'utilisateur vers une nouvelle page. En utilisant le service action au lieu d'un seul fichier de script, Elgg fournit automatiquement une sécurité et une extensibilité améliorées.

Voyez *Forms + Actions* pour plus de détails sur comment enregistrer et construire une action. Pour regarder les actions du noyau, regardez dans le répertoire /actions.

3.5.2 Base de données

Une discussion solide sur le design du modèle de données Elgg et ses motivations.

Contents

Aperçu

Dans Elgg, tout fonctionne sur un modèle de données unifié construit sur des unités atomiques de données appelées entités

Il est recommandé que les plugins n'interagissent pas directement avec la base de données, afin de créer un système plus stable et une meilleure expérience utilisateur parce que les contenus créés par différents plugins peut être mélangés ensemble de manières cohérentes. Avec cette approche, les plugins sont plus rapides à développer, et sont en même temps plus puissants.

Chaque entité dans le système hérite de la classe `ElggEntity`. Cette classe contrôle les permissions d'accès, la propriété, le conteneur, et fournit une API cohérente pour accéder aux propriétés des entités et les modifier.

Vous pouvez étendre les entités avec des informations supplémentaires de deux manières :

`Metadata` : Ces informations décrivent l'entité, elles sont habituellement ajoutées par l'auteur de l'entité quand l'entité est créée ou modifiée. Des exemples de métadonnées incluent les tags, un numéro ISBN ou un ID tierce-partie, une adresse, des coordonnées géographiques, etc. Imaginez les métadonnées comme un simple système de stockage clef-valeur.

`Annotations` : Ces informations étendent l'entité avec des propriétés, habituellement ajoutées par une tierce-partie. De telles propriétés comprennent les notations, likes et votes.

La principale différence entre métadonnées et annotations :
— les métadonnées n'ont pas de propriétaire, alors que les annotations en ont
— les métadonnées ne disposent pas de contrôle d'accès, alors que les annotations en disposent
— les métadonnées sont préchargées lorsque l'entité est construite, alors que les annotations ne sont chargées que sur demande

Ces différences peuvent avoir des implications sur les performances et votre logique applicative, aussi considérez avec soin la manière dont vous souhaiter attacher des données à vos entités.

Dans certains cas, il peut être préférable d'éviter d'utiliser ces métadonnées et annotations et de créer de nouvelles entités à la place, puis de les attacher via `container_guid` ou une relation.

Modèle de données (datamodel)

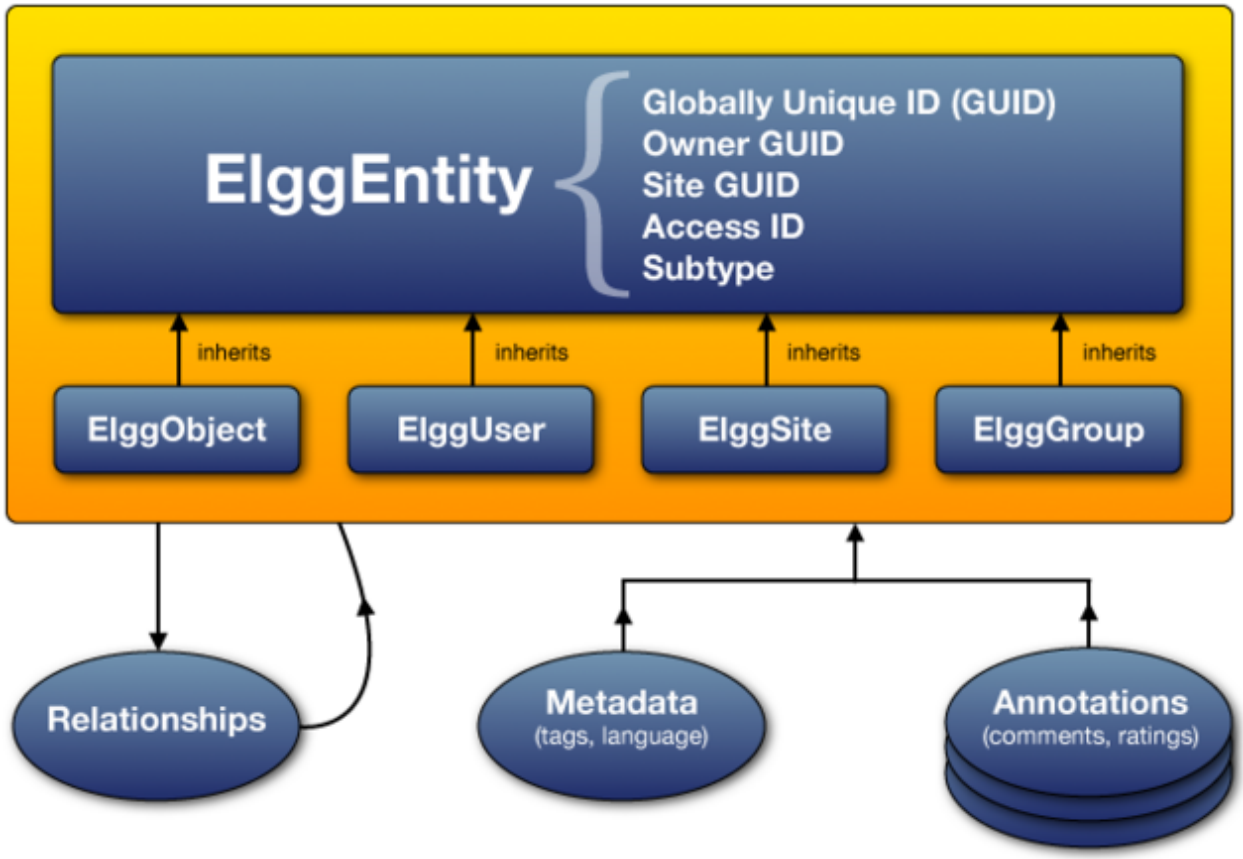

Fig. 9 – Le diagramme du modèle de données d'Elgg

Entités

`ElggEntity` est la classe de base pour le modèle de données d'Elgg et supporte un jeu de propriétés et méthodes communes.

- Un identifiant global unique (Globally Unique IDentifier - Voir *GUIDs*).
- Permissions d'accès . (Quand un plugin demande des données, il n'accède jamais à des données que l'utilisateur actuel n'a pas la permission de voir.)
- Un sous-type arbitraire (plus d'informations ci-dessous).
- Un propriétaire (owner).
- Le site auquel appartient cette entité.
- Un conteneur, utilisé pour associer le contenu avec un groupe ou un utilisateur.

Types

Les *véritables* entités seront des instances de quatre sous-classes différentes, chacune ayant une propriété **type** distincte et leur propres propriétés et méthodes additionnelles.

Type	Classe PHP	Représente
objet	`ElggObject`	Les plupart des contenus créés par les utilisateurs, tels que des articles de blog, des chargements de fichiers, et des signets.
groupe	`ElggGroup`	Un groupe organisé d'utilisateurs avec sa propre page de profil
utilisateur	`ElggUser`	Un utilisateur du système
site	`ElggSite`	Le site servi par l'installation Elgg

Chaque type a sa propre API étendue. Par ex. les utilisateurs peuvent être en contact avec d'autres utilisateurs, un groupe peut avoir des membres, tandis que les objets peuvent être « likés » et commentés.

Sous-types (subtypes)

Chaque entité doit définir un **sous-type** (subtype), que les plugins utilisent pour spécialiser encore l'entité. Elgg rend facile le fait de rechercher des entités spécifiques d'un ou de plusieurs sous-types donnés, ainsi que de leur assigner des comportements et vues spéciales.

Les sous-types sont communément donnés aux instances de `ElggEntity` pour définir le type de contenu créé. Par ex. le plugin blog crée des objets avec le sous-type `"blog"`.

Par défaut, les utilisateurs, les groupes et les sites ont respectivement le sous-type `user`, `group` et `site`.

Les plugins peuvent utiliser des classes d'entités personnalisées qui étendent l classe de base du type. Pour cela, ils doivent enregistrer leur classe au démarrage (par ex. dans le gestionnaire `'init', 'system'`), en utilisant `elgg_set_entity_class()`. Par exemple, le plugin blog pourraity utiliser `elgg_set_entity_class('object', 'blog', \ElggBlog::class)`.

Les plugins peuvent utiliser `elgg-plugin.php` pour définir une classe d'entité via le paramètre raccourci `entities`.

Trucs à savoir sur les sous-types

— Avant que la méthode `save()` d'une entité soit appelée, le sous-type doit être défini en écrivant une chaîne de caractère dans la propriété `subtype`.
— *Le sous-type ne peut pas être changé après l'enregistrement.*

GUIDs

Un GUID est un entier qui définit de manière unique chaque entité dans une installation Elgg (un IDentifiant Global Unique - Globally Unique IDentifier). Il est assigné automatiquement la première fois qu'une entité est enregistré, et ne peut jamais être changé.

Certaines fonctions de l'API Elgg focntionnenet avec des GUIDs au lieu d'objets `ElggEntity`.

ElggObject

Le type d'entité `ElggObject` représente un type de contenu arbitraire au sein d'une installation Elgg ; des choses telles que des articles de blog, des fichiers, etc.

Au-delà des propriétés standards de ElggEntity, ElggObjects supporte également :
- `title` Le titre de l'objet (texte sans HTML)
- `description` Une description de l'objet (HTML)

La plupart des autrse données à propos de l'objet sont généralement stockées via des métadonnées.

ElggUser

Le type d'entité `ElggUser` représente les utilisateurs au sien d'une installation Elgg. Il seront définis comme désactivés jusqu'à-ce que leur compte ait été activé (à moins qu'ils n'aient été créés à partir du panneau d'administration).

Au-delà des propriétés standards de ElggEntity, ElggUsers supporte également :
- `name` Le nom de l'utilisateur en texte brut. Par ex. « Hugh Jackman »
- `username` Leur nom d'utilisateur. Par ex. « hjackman »
- `password` Une version hachée de leur mot de passe
- `email` Leur adresse email
- `language` Leur code de langue par défaut.
- `code` Leur code de session (déplacé vers une table séparée dans la 1.9).
- `last_action` Le timestamp UNIX de la dernière fois qu'ils ont chargé une page
- `prev_last_action` La précédente valeur de `last_action`
- `last_login` Le timestamp UNIX de leur dernière connexion
- `prev_last_login` la précédente valeur de `last_login`

ElggSite

Le type d'entité `ElggSite` représente votre installation Elgg (via l'URL de votre site).

Au-delà des propriétés standards de ElggEntity, ElggSites supporte également :
- `name` Le nom du site
- `description` Une description du site
- `url` L'adresse du site

ElggGroup

Le type d'entité `ElggGroup` représente une association d'utilisateurs Elgg. Les utilisateurs peuvent rejoindre, quitter les groupes, et y publier du contenu.

Au-delà des propriétés standards de ElggEntity, ElggGroups supporte également :
- `name` Le nom du groupe (texte sans HTML)
- `description` Une description du groupe (HTML)

`ElggGroup` a des méthodes additionnelles pour gérer le contenu et les adhésions.

Le plugin Groups

A ne pas confondre avec le type d'entité `ElggGroup`, Elgg vient avec un plugin appelé « Groups » qui fournit une UI/UX par défaut pour que les utilisateurs du site interagissent avec les groupes. Chaque groupe dispose d'un forum de discussion et d'une page de profil qui relient les utilisateurs au contenu dans le groupe..

Vous pouvez modifier l'expérience utilisateur via les moyens traditionnels d'extension de plugin, ou remplacer complètement le plugin Groups par votre propre plugin.

Écrire un plugin conscient des groupes

Les développeurs de plugins ne devraient pas trop s'inquiéter d'écrire une fonctionnalité consciente des groupes, mais il y a quelques points clefs :

Ajouter du contenu

En passant le groupe en tant que `container_guid` via un champ de saisie caché, vous pouvez utiliser un seul formulaire et une seule action pour ajouter du contenu à la fois pour un utilisateur ou pour un groupe.

Utilisez `ElggEntity->canWriteToContainer()` pour déterminer si l'utilisateur actuel a ou n'a pas le droit d'ajouter du contenu dans un groupe.

Soyez attentif au fait que vous allez devoir passer le GUID du conteneur ou le nom d'utilisateur à la page responsable de la publication et la valeur associée, de sorte que ceci puisse être ensuite conservé dans votre formulaire sous forme de champ de saisie caché, pour un passage plus aisé vers vos actions. Au sein d'une action « create », vous allez avoir besoin de récupérer ce champ de saisie et de l'enregistrer sous forme de propriété de votre nouvel élément (la valeur par défaut est le conteneur actuel de l'utilisateur) :

```
$user = elgg_get_logged_in_user_entity();
$container_guid = (int)get_input('container_guid');

if ($container_guid) {
    $container = get_entity($container_guid);

    if (!$container->canWriteToContainer($user->guid)) {
        // register error and forward
    }
} else {
    $container_guid = elgg_get_logged_in_user_guid();
}

$object = new ElggObject;
$object->container_guid = $container_guid;

...

$container = get_entity($container_guid);
forward($container->getURL());
```

Jongler entre utilisateurs et groupes

En fait, `[[Engine/DataModel/Entities/ElggGroup|ElggGroup]]` simule la plupart des méthodes de `[[Engine/DataModel/Entities/ElggUser|ElggUser]]`. Vous pouvez récupérer l'icône, le nom, etc. en utilisant les mêmes appels, et si vous demandez les contacts du groupe, vous récupérez ses membres. Ceci a été designé spécifiquement pour que vous puissiez alterner entre groupes et utilisateurs aisément dans votre code.

Propriété

Les entités ont une propriété GUID `owner_guid`, qui définit son propriétaire. Typiquement ceci renvoie au GUID d'un utilisateur, quoique les sites et les utilisateurs eux-même n'ont souvent pas de propriétaire (valeur de 0).

La propriété d'une entité détermine, d'une part, si vous pouvez ou non accéder à, ou modifier cette entité.

Conteneurs (containers)

Afin de rechercher aisément du contenu par groupe ou utilisateur, le contenu est généralement défini comme « contenu » soit par l'utilisateur qui l'a publié, soit par le groupe dans lequel l'utilisateur l'a publié. Ceci signifie que la propriété `container_guid` des nouveaux objets sera définie avec la valeur du ElggUser actuel ou du ElggGroup cible.

Par ex., trois articles de blog peuvent avoir des propriétaires différentes, mais être tous contenus dans le groupe où ils ont été publiés.

Note : Ceci n'est pas toujours vrai. Les entités Commentaires sont contenues par l'objet commenté, et dans certains plugins tierce-partie la conteneur peut être utilisé pour modéliser des relations parents-enfants entre entités (par ex. un objet « dossier » qui contient un objet fichier).

Annotations

Les annotations sont des éléments de données attachées à une entité qui permet aux utilisateurs de laisser des évaluations, ou d'autres types de réactions pertinentes. Un plugin de sondage pourrait enregistrer des votes sous forme d'annotations.

Les annotations sont stockées sous formes d'instances de la classe `ElggAnnotation`.

Chaque annotation dispose de :
- Un type d'annotation interne (tel que *comment*)
- Une valeur (qui peut être une chaîne de caractères ou un entier)
- Un niveau d'accès distinct de celui de l'entité à laquelle il est attaché
- Un propriétaire

Comme pour les métadonnées, les valeurs sont stockées sous forme de chaines de caractères à moins que la valeur donnée soit un entier PHP (`is_int($value)` vaut true), ou à moins que `$vartype` soit spécifié manuellement comme `integer`.

Ajouter une annotation

La manière la plus simple d'ajouter une annotation est d'utiliser la méthode `annotate` sur une entité, qui est définie comme :

```
public function annotate(
    $name,           // The name of the annotation type (eg 'comment')
    $value,          // The value of the annotation
    $access_id = 0,  // The access level of the annotation
    $owner_id = 0,   // The annotation owner, defaults to current user
    $vartype = ""    // 'text' or 'integer'
)
```

Par exemple, pour donner une notation à une entité, vous pouvez appeler :

```
$entity->annotate('rating', $rating_value, $entity->access_id);
```

Lire les annotations

Pour récupérer les annotations d'une entité, vous pouvez appeler la méthode suivante :

```
$annotations = $entity->getAnnotations(
    $name,    // The type of annotation
    $limit,   // The number to return
    $offset,  // Any indexing offset
    $order,   // 'asc' or 'desc' (default 'asc')
);
```

Si votre type d'annotation utilise largement des valeurs entières, plusieurs fonctions mathématiques utiles sont fournies :

```
$averagevalue = $entity->getAnnotationsAvg($name);  // Get the average value
$total = $entity->getAnnotationsSum($name);         // Get the total value
$minvalue = $entity->getAnnotationsMin($name);      // Get the minimum value
$maxvalue = $entity->getAnnotationsMax($name);      // Get the maximum value
```

Fonctions d'aide utiles

Commentaires

Si vous souhaitez fournir une fonctionnalité de commentaire sur les objets de votre plugin, la fonction suivante va fournir le listing complet, le formulaire et les actions :

```
function elgg_view_comments(ElggEntity $entity)
```

Métadonnées

Les métadonnées dans Elgg vous permettent de stocker les données supplémentaires d'une `entité` au-delà des champs pré-définis que cette entité supporte. Par exemple, `ElggObjects` ne supporte que les champs d'entités basiques plus un titre et une description, mais vous pouvez souhaiter inclure également des tags ou un numéro ISBN. De manière similaire, vous pourriez souhaiter que les utilisateurs puissent enregistrer une date de naissance.

Sous le capot, les métadonnées sont stockées sous forme d'instance de la classe `ElggMetadata`, mais vous n'avez pas à vous soucier de cela en pratique (quoique si vous êtes intéressé, voyez la référence de la classe `ElggMetadata`). Ce que vous avez besoin de savoir :

— Une métadonnée a un propriétaire, qui peut être différent du propriétaire de l'entité à laquelle elle est attachée
— Vous pouvez potentiellement avoir plusieurs éléments de chaque type de métadonnée attachés à la même entité
— Comme pour les annotations, les valeurs sont stockées sous forme de chaîne de caractères à moins que la valeur donnée soit un entier PHP (`is_int($value)` vaut true), ou à moins que le `$value_type` soit défini manuellement comme `integer` (voyez ci-dessous).

Note : A partir de Elgg 3.0, les métadonnées n'ont plus d'"`access_id`.

Le cas simple

Ajouter des métadonnées

Pour ajouter une métadonnée à une entité, appelez simplement :

```
$entity->metadata_name = $metadata_value;
```

Par exemple, pour ajouter une date d'anniversaire à un utilisateur :

```
$user->dob = $dob_timestamp;
```

Ou pour ajouter des tags à un objet :

```
$object->tags = array('tag one', 'tag two', 'tag three');
```

Lorsque vous ajoutez une métadonnée de cette manière :

— Le propriétaire est assigné à l'utilisateur actuellement identifié
— Réassigner une métadonnée va remplacer l'ancienne valeur

Ceci convient pour la plupart des objectifs. Faites attention à bien noter quels attributs sont des métadonnées et lesquels sont natifs au type d'entité avec lequel vous travaillez. Vous n'avez pas besoin d'enregistrer une entité après avoir ajouté ou modifié des métadonnées. Vous devez enregistrer une entité si vous avez changé l'un des ses attributs natifs. A titre d'exemple, si vous avez changé l'id d'accès d'un ElggObject, vous devez l'enregistrer, faute de quoi la modification ne sera pas conservée dans la base de données.

Note : A partir de Elgg 3.0, la propriété `access_id` des métadonnées est ignorée

---

Lire les métadonnées

Pour récupérer une métadonnée, traitez-la comme une propriété d'une entité :

```
$tags_value = $object->tags;
```

Notez que ceci va retourner la valeur absolue de la métadonnée. Pour récupérer des métadonnées sous forme d'objet ElggMetadata, vous aurez besoin d'utiliser les méthodes décrites dans la section *contrôle plus fin* ci-dessous.

Si vous stockez de multiples valeurs dans cette métadonnée (comme dans l'exemple « tags » ci-dessus), vous obtiendrez un tableau de toutes ces valeurs. Si vous stockez seulement une valeur, vous récupérerez une chaîne de caractères ou un entier. Stocker un tableau avec seulement une valeur vous retournera une chaîne de caractères. Par ex.

```
$object->tags = array('tag');
$tags = $object->tags;
// $tags will be the string "tag", NOT array('tag')
```

Pour récupérer toujours un tableau, castez simplement vers un tableau :

```
$tags = (array)$object->tags;
```

Lire des métadonnées comme des objets

`elgg_get_metadata` est la meilleure fonction pour récupérer les métadonénes sous forme d'objets ElggMetadata :

Par ex., pour récupérer la date de naissance d'un utilisateur

```
elgg_get_metadata(array(
    'metadata_name' => 'dob',
    'metadata_owner_guid' => $user_guid,
));
```

Ou pour récupérer toutes les objets des métadonnées :

```
elgg_get_metadata(array(
    'metadata_owner_guid' => $user_guid,
    'limit' => 0,
));
```

Erreurs courantes

« Ajouter » des métadonnées

Notez que vous ne pouvez pas « ajouter » des valeurs aux tableaux de métadonnées comme si c'étaient des tableaux php normaux. Par exemple, le code suivant ne fera pas ce qu'il semblerait qu'il devrait faire.

```
$object->tags[] = "tag four";
```

Essayer de stocker des tableaux indexés

Elgg ne supporte pas le stockage de tableaux indexés (paires clef/valeur) dans les métadonnées. Par exemple, le code suivant ne fonctionne pas comme vous pourriez le penser de prime abord :

```
// Won't work!! Only the array values are stored
$object->tags = array('one' => 'a', 'two' => 'b', 'three' => 'c');
```

Au lieu de cela, vous pouvez conserver cette information de cette manière :

```
$object->one = 'a';
$object->two = 'b';
$object->three = 'c';
```

Conserver des GUIDs dans des métadonnées

Quoiqu'il existe certains cas pour stocker les GUIDs d'entités dans des métadonnées, les relations *Relationships* sont une construction bien plus adaptée pour relier les entités les unes aux autres.

Relations

Les relations vous permettent d'associer des entités ensemble. Exemples : un artiste a des fans, un utilisateur est membre d'une organisation, etc.

La classe `ElggRelationship` modélise une relation directe entre deux entités, en faisant la déclaration :

> « **{subject}** est un **{noun}** de **{target}**. »

Nom de l'API	Modèles	Représente
`guid_one`	Le sujet	Quelle entité est reliée
`relationship`	Le nom	Le type de relation
`guid_two`	La cible	L'entité à laquelle le sujet est relié

Le type de relation peut également être un verbe, faisant la déclaration :

> « **{sujet} {verbe} {cible}**. »
> Par ex. Utilisateur A « aime » l'article de blog B

Chaque relation a une direction. Imaginez un archer qui tire une flèche vers une cible ; la flèche se déplace dans une direction, en reliant le sujet (l'archer) à à la cible.

Une relation n'implique pas de réciprocité. **A** suit **B** n'implique pas que **B** suive **A**.

Les **Relations_ n'ont pas de contrôle d'accès.** Elles ne sont jamais cachées d'une vue et peuvent être modifiées par le code à n'importe quel niveau de privilège, avec l'avertissement que *les entités* d'une relation peuvent être invisible en raison du contrôle d'accès !

Travailler avec des relations

Créer une relation

Par exemple pour établir que « **$utilisateur** est un **fan** de **$artiste** » (utilisateur est le sujet, artiste est la cible) :

```
// option 1
$success = add_entity_relationship($user->guid, 'fan', $artist->guid);

// option 2
$success = $user->addRelationship($artist->guid, 'fan');
```

Ceci déclenche l'événement [create, relationship], en lui passant l'objet `ElggRelationship` créé. Si un gestionnaire retourne `false`, la relation ne sera pas créée et `$success` vaudra `false`.

Vérifier une relation

Par exemple pour vérifier que « **$utilisateur** est un **fan** de **$artiste** » :

```
if (check_entity_relationship($user->guid, 'fan', $artist->guid)) {
    // relationship exists
}
```

Notez que si la relation existe, `check_entity_relationship()` retourne un objet `ElggRelationship` :

```
$relationship = check_entity_relationship($user->guid, 'fan', $artist->guid);
if ($relationship) {
    // use $relationship->id or $relationship->time_created
}
```

Supprimer une relation

Par exemple pour pouvoir affirmer que « **$utilisateur** n'est désormais plus un **fan** de **$artiste** » :

```
$was_removed = remove_entity_relationship($user->guid, 'fan', $artist->guid);
```

Ceci déclenche l'événement [delete, relationship], en lui passant l'objet `ElggRelationship` associé. Si un gestionnaire retourne `false`, la relation sera conservée, `$was_removed` vaudra `false`.

Autres fonctions utiles :

- `delete_relationship()` : supprimer par ID
- `remove_entity_relationships()` : supprime ceux qui sont liés à une entité

Trouver des relations et des entités associées

Voici ci-dessous quelques fonctions pour récupérer des relations entre objets et/ou des entités reliées :
- `get_entity_relationships()` : récupère les relations par entité sujet ou entité cible
- `get_relationship()` : récupère un objet relation par ID
- `elgg_get_entities()` : récupère les entités dans les relations de différentes manières

Par ex. retrouver les utilisateurs qui ont rejoint votre groupe en janvier 2014.

```
$entities = elgg_get_entities(array(
    'relationship' => 'member',
    'relationship_guid' => $group->guid,
    'inverse_relationship' => true,

    'relationship_created_time_lower' => 1388534400, // January 1st 2014
    'relationship_created_time_upper' => 1391212800, // February 1st 2014
));
```

Contrôle d'accès

Les contrôle d'accès granulaires sont l'un des principes de design fondamentaux dans Elgg, et une fonctionnalité qui a été au cœur du système tout au long de son développement. L'idée est simple : un utilisateur devrait avoir un contrôle total sur qui peut voir un élément de donnée qu'il ou elle a créé.

Niveau d'accès dans le modèle de données

Pour parvenir à cela, chaque entité et annotation contient une propriété `access_id`, qui correspond à l'un des niveau d'accès prédéfinis ou à une entrée dans la table `access_collections` de la base de données.

Niveaux d'accès pré-définis

- `ACCESS_PRIVATE` (valeur : 0) Privé.
- `ACCESS_LOGGED_IN` (value : 1) Logged in users.
- `ACCESS_PUBLIC` (value : 2) Public data.

Niveaux d'accès définis par l'utilisateur

Vous pouvez définir des groupes d'accès additionnels et les assigner à une entité, ou une annotation. Un certain nombre de fonctions ont été définies pour vous y aider ; voyez *Access Control Lists* pour plus d'informations.

Comment les accès affectent la récupération de données

Toutes les récupérations de données fonctionnent au-dessus de la couche de la base de données - par exemple `elgg_get_entities` ne retournera que les éléments que l'utilisateur actuel a l'autorisation de voir. Il n'est pas possible de récupérer des éléments auxquels l'utilisateur actuel n'a pas accès. Ceci rend très difficile de créer un trou de sécurité pour récupérer des données.

Accès en écriture

Les règles suivantes gouvernent les accès en écriture :
- Le propriétaire d'une entité peut toujours la modifier
- Le propriétaire d'un conteneur peut modifier tout ce qui est dedans (notez que cela ne signifie pas que le propriétaire d'un groupe peut modifier toutes les publications dans ce groupe)
- Les administrateurs peuvent tout éditer

Vous pouvez surcharger ce comportement en utilisant un *hook plugin* appelé `permissions_check`, qui passe l'entité en question à toute focntion qui a annoncé qu'elles veut être référencée. Retourner `true` va autoriser l'accès en écriture ; retourner `false` va l'interdire. Voyez *les références des plugin hook pour les permissions_check* pour plus de détails.

Schéma

La base de données contient un certain nombre de tables primaires et secondaires. Vous pouvez suivre les évolutions du schéma dans `engine/schema/migrations/`

Tables principales

Voici la description des tables principales. Gardez à l'esprit que pour une installation d'Elgg données les tables vont avoir un préfixe (typiquement « elgg_ »).

Table : entities (entités)

Ceci est la table principale *Entities* qui contient les utilisateurs, sites, objets et groupes Elgg. Quand vous installez Elgg la première fois elle est automatiquement peuplée avec votre premier site.

Elles contient les champs suivants :
- **guid** Un compteur auto-incrémenté qui poduit un GUID qui identifie de manière unique cette entité dans le système
- **type** Le type d'entity - object, user, group ou site
- **subtype** Le sous-type d'entité
- **owner_guid** Le GUID de l'entité du propriétaire
- **container_guid** Le GUID dans laquelle cette entité est contenue - soit un utilisateur (user) soit un groupe (group)
- **access_id** Contrôles d'accès sur cette entité
- **time_created** Le timestamp Unix de création de l'entité
- **time_updated** Le timestamp Unix de la dernière modification de l'entité
- **enabled** Si oui ("yes") l'entité est accessible, si non ("no") l'entité a été désactivée (Elgg traite cela comme si elle avait été supprimée sans toutefois la supprimer réellement de la base de données)

Table : metadata

Cette table contient des métadonnées *Metadata*, des informations supplémentaires attachées à une entité.

- **id** Un IDentifiant unique
- **entity_guid** L'entité à laquelle cette métadonnée est attachée
- **name** Le nom de la chaîne
- **value** La valeur de la chaîne
- **value_type** La classe de la valeur, soit texte (text) ou un entier (integer)
- **time_created** Le timestamp Unix de la date de création de la métadonnée
- **enabled** Si oui ("yes") l'élément est accessible, si non ("no") l'élément a été désactivé

Table : annotations

Cette table contient les *Annotations*, ce qui est distinct des *Metadata*.

- **id** Un IDentifiant unique
- **entity_guid** L'entité à laquelle cette métadonnée est attachée
- **name** Le nom de la chaîne
- **value** La valeur de la chaîne
- **value_type** La classe de la valeur, soit texte (text) ou un entier (integer)
- **owner_guid** Le GUID du propriétaire qui a défini cette annotation
- **access_id** Un Contrôle d'Accès sur cette annotation
- **time_created** Le timestamp Unix de la date de création de l'annotation.
- **enabled** Si oui ("yes") l'élément est accessible, si non ("no") l'élément a été désactivé

Table : relationships (relations)

Cette table définit les *Relationships*, qui relient une entité avec une autre.

- **guid_one** Le GUID de l'entité sujet.
- **relationship** Le type de relation.
- **guid_two** Le GUID de l'entité cible.

Tables secondaires

Table : access_collections

Cette table définit les Collections d'Accès, qui donnent accès aux utilisateurs aux *Entités* ou aux *Annotations*.

- **id** Un IDentifiant unique
- ***name** Le nom de la collection d'accès
- **owner_guid** Le GUID de l'entité propriétaire (par ex. un utilisateur ou un groupe)
- **subtype** le sous-type de la collection d'accès (par ex. *friends* ou *group_acl*)

3.5.3 Événements et Hooks des plugins

Contents

Aperçu

Elgg a un événement système qui peut être utilisé pour remplacer ou étendre les fonctionnalités du noyau.

Les plugins influencent le système en créant des gestionnaires ou handlers (appelables tels que des fonctions et des méthodes) et en les enregistrant pour gérer deux types d'événements : *Evénements Elgg (Events)* et *Hooks des plugins*.

Quand un événement est déclenché, un jeu des gestionnaires (handlers) est exécuté par ordre de priorité. Les arguments sont passés à chaque gestionnaire qui peut influencer le processus. Après exécution, la fonction déclencheuse (« trigger ») retourne une valeur qui dépend du comportement des gestionnaires.

Voir aussi :

- *List of events in core*
- *List of plugin hooks in core*

Evénements Elgg (Events) vs. Hooks des Plugins

Les principales différences entre *Evénements Elgg (Events)* et *Hooks des plugins* sont :

1. La majorité des événements Elgg peuvent être annulé ; à moins que l'événement soit un événement « after », un gestionnaire qui retourne *false* peut annuler l'événement, et plus aucun autre gestionnaire n'est appelé.
2. Les hooks plugin ne peuvent pas être annulé ; tous les gestionnaires sont toujours appelés.
3. Les hooks lugin passent une valeur arbitraire à travers les gestionnaires, leur donnant à chacun une chance de la modifier en cours de route.

Evénements Elgg (Events)

Les événements Elgg Events sont déclenchés quand un objet Elgg est créé, modifié ou supprimé ; et à différentes étapes importantes du chargement du framework Elgg. Exemples : lors de la création d'un article de blog ou de l'identification d'un utilisateur.

Contrairement aux *Hooks des plugins*, *la plupart des événements Elgg peuvent être annulé*, ce qui interrompt l'exécution des gestionnaires, et potentiellement annulent certaines actions dans le noyau d'Elgg.

Chaque événement Elgg a un nom et un type d'objet (system, user, object, nom de relation, annotation, group) qui décrit le type d'objet passé aux gestionnaires.

Événements Avant (Before) et Après (After)

Certains événements sont séparés en « before » (avant) et « after » (après). Ceci évite la confusion autour de l'état fluctuant du système. Par ex. Est-ce que l'utilisateur est identifié au cours de l'événement [login, user] ?

Les événements Avant ont des noms qui se terminent par « :before » et sont exécutés avant qu'il ne se passe quelque chose. Comme pour les événements traditionnels, les gestionnaires peuvent annuler l'événement en retournant `false`.

Les événements Après, avec des noms qui se terminent par « :after », sont exécutés après qu'il se soit passé quelque chose. Au contraire des événements traditionnels, ces gestionnaires *ne peuvent pas* annuler ces événements ; tous les gestionnaires seront toujours appelés.

Là où des événements :before et :after sont disponibles, les développeurs sont encouragés à faire la transition vers eux, même si d'anciens événements resteront supportés pour des raisons de compatibilité descendante.

Gestionnaires d'événement Elgg (Event Handlers)

Les gestionnaires d'événement Elgg sont appelables avec l'un des prototypes suivants :

```
<?php

/**
 * @param \Elgg\Event $event The event object
 *
 * @return bool if false, the handler is requesting to cancel the event
 */
function event_handler1(\Elgg\Event $event) {
    ...
}

/**
 * @param string $event       The name of the event
 * @param string $object_type The type of $object (e.g. "user", "group")
 * @param mixed  $object      The object of the event
 *
 * @return bool if false, the handler is requesting to cancel the event
 */
function event_handler2($event, $object_type, $object) {
    ...
}
```

Dans `event_handler1`, l'objet `Event` a diverses méthodes pour récupérer le nom, le type d'objet, et l'objet de l'événement. Voyez l'interface `Elgg\Event` pour plus de détails.

Dans tous les cas, si un gestionnaire retourne `false`, l'événement est annulé, ce qui empêche l'exécution des autres gestionnaires. Toutes les autres valeurs de retour sont ignorées.

Note : Si le type d'événement est « object » ou « user », utilisez le type de données `\Elgg\ObjectEvent` ou `\Elgg\UserEvent` à la place, ce qui clarifie le type de retour de la méthode `getObject()`.

Enregistrez un gestionnaire d'événement (Elgg Event)

Enregistrez votre gestionnaire pour un événement en utilisant `elgg_register_event_handler` :

```
<?php

elgg_register_event_handler($event, $object_type, $handler, $priority);
```

Paramètres :

- **$event** Le nom de l'événement.
- **$object_type** Le type d'objet (par ex. « user » ou « object ») ou "all" pour tous les types pour lesquels l'événement est déclenché.
- **$handler** Le callback ou la fonction gestionnaire.
- **$priority** La priorité - 0 en premier et la valeur par défaut est 500.

Objet ne fait pas ici référence à un `ElggObject` mais plutôt à une chaîne de caractères qui décrit n'importe quel objet dans le framework : le système, un utilisateur, un objet, une relation, une annotation, un groupe.

Exemple :

```
<?php

// Register the function myPlugin_handle_create_object() to handle the
// create object event with priority 400.
elgg_register_event_handler('create', 'object', 'myPlugin_handle_create_object', 400);
```

Avertissement : Si vous gérez l'événement « update » d'un objet, évitez d'appeler `save()` dans votre gestionnaire d'événement. Tout d'abord ce n'est probablement pas nécessaire car l'objet est enregistré après que l'événement soit terminé, mais aussi parce que `save()` appelle un autre événement « update » et ne rend plus disponible `$object->getOriginalAttributes()`.

Classes invocables comme gestionnaires

Vous pouvez utiliser une classe avec une méthode `__invoke()` comme gestionnaire. Enregistrez simplement le nom de la classe et elle sera instanciée (sans argument) pour toute la durée de vie de l'événement (ou du hook).

```
<?php

namespace MyPlugin;

class UpdateObjectHandler {
    public function __invoke(\Elgg\ObjectEvent $event) {

    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
// in init, system
elgg_register_event_handler('update', 'object', MyPlugin\UpdateObjectHandler::class);
```

Déclencher un événement (Elgg Event)

Vous pouvez déclencher un événement personnalisé en utilisant `elgg_trigger_event` :

```
<?php

if (elgg_trigger_event($event, $object_type, $object)) {
    // Proceed with doing something.
} else {
    // Event was cancelled. Roll back any progress made before the event.
}
```

Pour les événements avec des états ambigus, tels que l'identification d'un utilisateur, vous devriez utiliser *Événements Avant (Before) et Après (After)* en appelant `elgg_trigger_before_event` ou `elgg_trigger_after_event`. Ceci clarifie pour le gestionnaire d'événement l'état auquel attendre et quels événements peuvent être annulés.

```
<?php

// handlers for the user, login:before event know the user isn't logged in yet.
if (!elgg_trigger_before_event('login', 'user', $user)) {
    return false;
}

// handlers for the user, login:after event know the user is logged in.
elgg_trigger_after_event('login', 'user', $user);
```

Paramètres :

- **$event** Le nom de l'événement.
- **$object_type** Le type d'objet (par ex. « user » ou « object »).
- **$object** L'objet (par ex. une instance de `ElggUser` ou `ElggGroup`)

La fonction va retourner `false` si n'importe lequel des gestionnaires sélectionnés retourne `false` et si l'événement est interruptible, sinon elle retournera `true`.

Hooks des plugins

Les Hooks Plugin fournissent un moyen pour les plugins de déterminer ou de modifier collaborativement une valeur. Par exemple, pour décider si un utilisateur a la permission de modifier une entité ou d'ajouter des options de configurations supplémentaires à un plugin.

Un hook plugin a une valeur passée à la fonction déclencheuse (« trigger »), et chaque gestionnaire a la possibilité de modifier cette valeur avant qu'elle soit passée au prochain gestionnaire. Après que le dernier gestionnaire a été exécuté, la valeur finale est retournée par le trigger.

Gestionnaires des Hooks plugin

Les gestionnaires de hooks sont appelables avec l'un des prototypes suivants :

```
<?php

/**
 * @param \Elgg\Hook $hook The hook object
 *
 * @return mixed if not null, this will be the new value of the plugin hook
 */
function plugin_hook_handler1(\Elgg\Hook $hook) {
    ...
}

/**
 * @param string $hook    The name of the plugin hook
 * @param string $type    The type of the plugin hook
 * @param mixed  $value   The current value of the plugin hook
 * @param mixed  $params  Data passed from the trigger
 *
 * @return mixed if not null, this will be the new value of the plugin hook
 */
function plugin_hook_handler2($hook, $type, $value, $params) {
    ...
}
```

Dans `plugin_hook_handler1`, l'objet `Hook` a diverses méthodes pour récupérer le nom, le type, la valeur et les paramètres du hook. Voyez l'interface `Elgg\Hook` pour plus de détails.

Dans les deux cas, si le gestionnaire ne retourne aucune valeur (ou explicitement la valeur `null`), la valeur du hook plugin n'est pas modifiée. Dans le cas contraire, la valeur retournée devient la nouvelle valeur du hook plugin, et il deviendra alors disponible via `$hook->getValue()` (ou `$value`) dans le prochain gestionnaire.

Enregistrez un gestionnaire de Hook plugin

Enregistrez votre gestionnaire pour un hook plugin en utilisant `elgg_register_plugin_hook_handler` :

```
<?php

elgg_register_plugin_hook_handler($hook, $type, $handler, $priority);
```

Paramètres :

- **$hook** Le nom du hook plugin.
- **$type** Le type de hook, ou "all" pour tous les types.
- **$handler** Le callback ou la fonction gestionnaire.
- **$priority** La priorité - 0 en premier et la valeur par défaut est 500.

Type peut avoir des sens différents. Cela peut signifier un type d'entité Elgg ou quelque chose de spécifique au nom du hook plugin.

Exemple :

```
<?php

// Register the function myPlugin_hourly_job() to be called with priority 400.
elgg_register_plugin_hook_handler('cron', 'hourly', 'myPlugin_hourly_job', 400);
```

Déclencher un Hook plugin

Vous pouvez déclencher un plugin hook personnalisé en utilisant `elgg_trigger_plugin_hook` :

```
<?php

// filter $value through the handlers
$value = elgg_trigger_plugin_hook($hook, $type, $params, $value);
```

Paramètres :

- **$hook** Le nom du hook plugin.
- **$type** Le type de hook, ou "all" pour tous les types.
- **$params** Des données arbitraires passées par le déclencheur aux gestionnaires.
- **$value** La valeur initiale du hook plugin.

Avertissement : Les arguments *$params* et *$value* sont inversés entre les gestionnaires de hook plugin et les focntions de déclenchement !

Dé-enregistrer des gestionnaires d'événement ou de hook

Les fonctions `elgg_unregister_event_handler` et `elgg_unregister_plugin_hook_handler` peuvent être utilisées pour retirer des gestionnaires déjà enregistrés par un autre plugin ou le noyau d'Elgg. Les paramètres sont dans le même ordre que pour les fonctions d'enregistrement, excepté qu'il n'y a pas de paramètre de priorité.

```
<?php

elgg_unregister_event_handler('login', 'user', 'myPlugin_handle_login');
```

Les fonctions anonymes ou les objets invocables ne peuvent pas être enregistrés, mais des fonctions de rappel (callback) de méthode dynamique peuvent être dé-enregistrés en donnant la version statique de la fonction de rappel :

```
<?php

$obj = new MyPlugin\Handlers();
elgg_register_plugin_hook_handler('foo', 'bar', [$obj, 'handleFoo']);

// ... elsewhere

elgg_unregister_plugin_hook_handler('foo', 'bar', 'MyPlugin\Handlers::handleFoo');
```

Même si le gestionnaire d'événement référence un appel à une méthode dynamique, le code ci-dessus va bien supprimer le gesitonnaire.

Ordre d'appel des gestionnaires

Les gestionnaires sont d'abord appelés par ordre de priorité, puis par ordre d'enregistrement.

Note : Avant Elgg 2.0, l'enregistrement avec le mot-clef `all` provoquait un appel tardif des gestionnaires, même s'ils avaient été enregistrés avec des priorités plus faibles.

3.5.4 Internationalisation

Elgg 1.0 et supérieur diffèrent des versions précédentes dans le sens où ces versions utilisent un tableau de valeur texte personnalisé plutôt que gettext. Ceci améliore la performance du système et la fiabilité du système de traduction.

TODO : plus svp

3.5.5 AMD

Aperçu

Il existe deux systèmes JavaScript dans Elgg : le système 1.8 devenu obsolète, et le nouveau système compatible AMD (Asynchronous Module Definition) introduit dans la 1.9.

Discussions au sujet des bénéfices de l'utilisation d'AMD dans Elgg.

Pourquoi AMD ?

Nous avons beaucoup travaillé pour rendre le JavaScript d'Elgg plus maintenable et plus utile. Nous avons fait quelques grands pas dans la 1.8 en introduisant l'objet et la bibliothèque JavaScript « `elgg` », mais avons rapidement réalisé que l'approche que nous prenions n'était pas susceptible de passer à l'échelle.

La taille du JS sur le web est rapidement croissante, et le JS dans Elgg s'accroît également. Nous voulons qu'Elgg soit capable d'offrir une solution qui rende le développement JS aussi productif et maintenable que possible pour aller plus loin.

Les raisons de choisir AMD sont nombreuses et bien documentées. Mettons en avant seulement les raisons les plus pertinentes dans la mesure où elles sont liées à Elgg spécifiquement.

1. Gestion des dépendances simplifiée

Les modules AMD se chargent de manière asynchrone et s'exécutent dès que leurs dépendances sont disponibles, ce qui élimine le besoin de préciser des « priorités » et des emplacements « location » lorsqu'on enregistre une bibliothèque JS dans Elgg. De plus, vous n'avez pas besoin de vous soucier de charger explicitement les dépendances d'un module dans PHP. Le chargeur AMD (RequireJS en l'occurrence) prend soin de tous ces tracas pour vous. Il est également possible d'avoir des dépendances texte avec le plugin text de RequireJS, de sorte que le templating côté client devrait être un plaisir.

2. AMD fonctionne dans tous les navigateurs. Dès maintenant.

Les développeurs Elgg ont déjà écrit beaucoup de JavaScript. Nous savons que vous voulez en écrire plus. Nous ne pouvons pas accepter d'attendre 5-10 ans pour qu'une solution native de modules JS soit disponible dans tous les navigateurs avant que nous puissions organiser notre JavaScript d'une manière maintenable.

3. Vous n'avez pas besoin d'une étape de build pour développer avec AMD.

Nous apprécions le cycle édition-mise à jour (« edit-refresh »)du web. Nous voulions faire en sorte que toute personne qui développe avec Elgg puisse continuer à expérimenter ce plaisir. Des formats de modules synchrones comme Closure ou CommonJS n'étaient tout simplement pas une option pour nous. Mais même si AMD n'a pas besoin d'une étape de build, *il reste cependant très adapté pour le build*. A cause du wrapper `define()`, il est possible de concaténer de multiples modules dans un seul fichier et de livrer l'ensemble en une seule fois dans un environnement de production. [1]

AMD est un système de chargement de modules largement éprouvé et bien pensé pour le web d'aujourd'hui. Nous avons beaucoup de gratitude pour le travail qui a été mis dedans, et sommes excités de l'offrir comme la solution standard pour le développement JavaScript dans Elgg à partir de Elgg 1.9.

3.5.6 Sécurité

L'approche d'Elgg pour les diverses questions de sécurité est commune à toutes les applications web.

Astuce : Pour signaler une vulnérabilité potentielle dans Elgg, envoyez un email à security@elgg.org.

Contents

1. Ceci n'est pas encore supporté par le noyau Elgg, mais nous sommes en train de nous pencher dessus dans la mesure où la réduction des allers-retours est critique pour une bonne première expérience, particulièrement sur les appareils mobiles.

Mots de passe

Validation du mot de passe

La seule restriction qu'Elgg impose sur le mot de passe est qu'il doit comporter au moins 6 caractères par défaut, quoique ceci puisse être changé dans `/elgg-config/settings.php`. Des critères additionnels peuvent être ajoutés en enregistrant un plugin hook pour `registeruser:validate:password`.

Hachage du mot de passe

Les mots de passe ne sont jamais stockés en clair, seulement sous forme de valeur de hachage avec sel produit par bcrypt. Ceci est effectué via la fonction standard `password_hash()`. Sur des systèmes plus anciens, le polyfill `password-compat` est utilisé, mais l'algorithme est identique.

Les installations Elgg créées avant la version peuvent avoir des chaînes de hachage de mot de passe « historiques » résiduelles créées en utilisant MD5 avec sel. Elles ont été migrées vers bcrypt lorsque les utilisateurs se connectent, et seront totalement retirées lorsque le système est mis à niveau vers Elgg 3.0. Dans le même temps, nous sommes heureux d'assister les propriétaires de sites à retirer manuellement ces chaînes de hachage historique, même si cela force ces utilisateurs à réinitialiser leur mot de passe.

Mitigation du mot de passe

Elgg a un mécanisme de mitigation des mots de passe qui rend les attaques par dictionnaire depuis l'extérieur très difficiles. Un utilisateur n'est autorisé qu'à 5 tentatives de connexion par période de 5 minutes.

Réinitialisation du mot de passe

Si un utilisateur oublie son mot de passe, la génération d'un nouveau mot de passe aléatoire peut être demandée. Après la demande, un email est envoyé avec une URL unique. Quand l'utilisateur visite cette URL, un nouveau mot de passe est envoyé par email à l'utilisateur.

Sessions

Elgg utilise la gestion des session PHP avec des gestionnaires personnalisés. Les données de sessions sont stockées dans la base de données. Le cookie de session contient l'id de session qui lie l'utilisateur au navigateur. Les métadonnées de l'utilisateur sont conservées dans la session, notamment le GUID, le nom d'utilisateur et l'adresse email.

La durée de vie de la session est contrôlée par la configuraiton PHP du serveur et additionnellement à travers les options dans le fichier `/elgg-config/settings.php`.

Fixation de session

Elgg protège contre la fixation de session en régénérant l'id de session lorsqu'un utilisateur se déconnecte.

Cookie « Se souvenir de moi »

Afin de permettre aux utilisateurs de rester identifiés pour une période plus longue que le navigateur ait été fermé ou pas, Elgg utilise un cookie (nommé par défaut `elggperm`) qui contient ce qui peut être considéré comme un super identifiant de session. Cet identifiant est conservé dans la tabel des cookies. Quand une session est initiée, Elgg vérifie la présence du cookie `elggperm`. S'il existe et que le code de session dans le cookie correspond au code dans la table des cookies, l'utilisateur correspondant est automatiquement connecté.

Lorsqu'un utilisateur change son mot de passe tous les cookies permanents sont supprimés de la base de données.

La durée de vie des cookies persistants peut être contrôlée dans le fichier */elgg-config/settings.php*. La durée de vie par défaut est de 30 jours. Les entrées de la base de données pour les cookies persistants seront supprimées après l'expiration de la durée de vie.

Authentification alternative

Note : Cette section est très lacunaire

Pour remplacer le système d'authentification par défaut des utilisateurs d'Elgg, un plugin pourrait remplacer l'action par défaut `login` par sa propre action. Une meilleure alternative est d'enregistrer un gestionnaire d'authentification (PAM) en utilisant `register_pam_handler()` qui gère l'authentification des utilisateurs sur la base de nouvelles exigences.

HTTPS

Note : Vous devez activer le support de SSL sur votre serveur pour que chacune de ces techniques fonctionne.

Vous pouvez servir l'ensemble de votre site par SSL en changeant l'URL du site pour inclure « https » au lieu de seulement « http ».

XSS

Le filtrage est utilisé dans Elgg pour rendre les attaques XSS plus difficiles. L'objectif du filtrage est de supprimer les JavaScript et autres saisies dangereuses des utilisateurs.

Le filtrage est assuré à travers la fonction `filter_tags()`. Cette fonction prend une chaîne de caractères et retourne une chaîne filtrée. Elle déclenche le hook plugin `validate, input`.

Par défaut Elgg est fourni avec le code de filtrage htmLawed. Les développeurs peuvent ajouter n'importe quel code additionnel ou de remplacement sous forme de plugin.

La fonction `filter_tags()` est appelée pour chaque saisie utilisateur dès lors que la saisie est obtenue à travers un appel à `get_input()`. Si pour quelque raison un développeur souhaite ne pas appliquer le filtrage par défaut sur certaines saisies utilisateur, la fonction `get_input()` a un paramètre pour désactiver le filtrage.

CSRF / XSRF

Elgg génère des jetons de sécurité pour empêcher la contrefaçon de requêtes inter-sites. Ceux-ci sont intégrés dans tous les formulaires et les requêtes AJAX modificatrices d'état dès lors que la bonne API est utilisée. Lisez-en plus dans le guide de développement *Forms + Actions*.

URLs signées

Il est possible de protéger les URLs avec une signature unique. Lisez-en plus dans le the *Forms + Actions* Guide de dévelopement.

Injection SQL

L'API d'Elgg's assainit toutes les entrées avant d'effectuer des requêtes en base de données. Lisez-en plus dans la documentation de design *Base de données*.

Vie privée

Elgg utilise un système d'ACL pour contrôler quels utilisateurs ont accès à divers éléments de contenu. Lisez-en plus dans la *Base de données* documentation de design.

Sécurisation

Les administrateurs du site peuvent configurer les paramètres qui vont aider à sécuriser le site. Lisez-en plus dans le Guide d'administration *Sécurité*.

3.5.7 Loggable

Loggable iest une interface héritée par toute classe qui veut que les événements liés à ses objets membres soient enregistrés dans le journal système. `ElggEntity` et `ElggExtender` héritent tous deux de `Loggable`.

Loggable définit plusieurs méthodes de classe qui sont utilisées pour l'enregistrement du journal système par défaut, et peuvent être utilisées pour définir vos propres journaux (ainsi que pour d'autres objectifs) :

- `getSystemLogID()` Retourne un identifiant unique pour l'objet à des fins de conservation dans le journal système. C'est généralement le GUID de l'objet
- `getClassName()` Retourne le nom de la classe de l'objet
- `getType()` Retournr le type d'objet
- `getSubtype()` Récupère le sous-type de l'objet
- `getObjectFromID($id)` Pour un ID donné, retourne l'objet qui lui est associé

Détails de la base de données

Le journal système par défaut est enregistré dans la table `system_log` de la *base de données*. Il contient les champs suivants :

- **id** - Un identifiant numérique unique de la ligne
- **object_id** - Le GUID de l'entité sur laquelle l'action est effectuée
- **object_class** - La classe de l'entitésur laquelle l'action est effectuée (par ex. ElggObject)
- **object_type** - Le type d'entité sur laquelle l'action est effectuée (par ex. object)
- **object_subtype** - Le sous-type d'entité sur laquelle l'action est effectuée (par ex. blog)
- **event** - L'événement enregistré (par ex. create ou update)

— **performed_by_guid** - Le GUID de l'entité agissante (l'utilisateur qui réalise l'action)
— **owner_guid** - Le GUID de l'utilisateur à qui appartient l'entité sur laquelle l'action est effectuée
— **access_id** - Le niveau d'accès associé avec cette entrée de journal
— **time_created** - Le timestamp UNIX de la date de l'événement

3.6 Guides du contributeur

Participez à rendre Elgg encore meilleur.

Elgg est un projet communautaire. Il dépend du soutien de volontaires pour réussir. Voici plusieurs manières d'aider :

3.6.1 Écrire du code

Comprenez les standards et process d'Elgg pour que vos propositions de modification soient acceptées aussi vite que possible.

Contents

Agrément de licence

En proposant un patch vous accordez de publier le code sous une licence GPLv2 et une licence MIT.

Demandes de fusion (Pull Requests)

Les Pull requests (PRs) sont le meilleur moyen de contribuer au noyau d'Elgg. L'équipe de développement les utilise y compris pour les modifications les plus mineures.

Pour de nouvelles fonctionnalités, soumettez une demande de fonctionnalité ou parlez-en avec nous en premier lieu et assurez-vous que l'équipe du noyau approuve votre proposition avant de passer beaucoup de temps sur du code.

Listes de vérification (checklists)

Utilisez ces listes de vérification réduites pour les nouveaux PRs sur github afin de garantir des contributions de haute qualité et d'aider tout le monde à comprendre le statut des PRs ouverts.

Demandes de correction de bug (bugfix PR) :

```
- [ ] Commit messages are in the standard format
- [ ] Includes regression test
- [ ] Includes documentation update (if applicable)
- [ ] Is submitted against the correct branch
- [ ] Has LGTM from at least one core developer
```

Demande de fonctionnalité (feature PR) :

```
- [ ] Commit messages are in the standard format
- [ ] Includes tests
- [ ] Includes documentation
- [ ] Is submitted against the correct branch
- [ ] Has LGTM from at least two core developers
```

Choisir une branche vers laquelle publier

Le tableau suivant suppose que la dernière version stable est la 2.1.

Type de changement	Branche vers laquelle publier
Correctif de sécurité	Ne le faites pas ! Envoyez un email à security@elgg.org pour des conseils.
Correctif de bug	1.12 (ou 2.1 si le correctif pour 1.12 est trop complexe)
Performance	2.x
Dépréciation	2.x
Fonctionnalité mineure	2.x
Fonctionnalité majeure	master (maître)
A n'importe quel changement non rétro-compatible	master (maître)

Si vous ne savez pas quelle branche choisir, demandez !

La différence entre des changements mineurs et majeurs est subjective et soumise à l'appréciation de l'équipe du noyau.

Format du message de commit

Nous exigeons un format particulier afin de permettre de publier plus souvent, avec des journaux des modifications et un historique des sources améliorés. Suivez simplement les étapes suivantes :

1. Commencez par le `type` en sélectionnant la *dernière catégorie qui s'applique* dans cette liste :
   - **docs** - *uniquement* la documentation a été mise à jour
   - **chore** - (corvée) ceci comprend les remaniements (refactoring), les changements de style de code, l'ajout de tests manquants, ce qui concerne Travis, etc.
   - **perf** - l'objectif principal est d'améliorer la performance
   - **fix** - ceci corrige un bug
   - **deprecate** - la modification rend obsolète toute partie de l'API
   - **feature** - ceci ajoute une nouvelle fonctionnalité pour les utilisateurs ou les développeurs
   - **security** - la modification affecte une question de sécurité d'une manière ou d'une autre. *Merci de ne pas pousser ce commit vers un repository public.* Contactez plutôt security@elgg.org.

   Par ex., si votre commit fait des remaniements pour régler (fix) un bug, cela reste un « fix ». Cependant, si ce bug est lié à la sécurité, le type doit être « security » et vous devriez envoyer un email à security@elgg.org avant de procéder. En cas de doute, faites au mieux, et un relecteur (reviewer) vous fournira des conseils.
2. Entre parenthèses, ajoutez le `component`, une courte chaîne qui décrit les sous-syst-me en train d'être modifié.

   Quelques exemples : `views`, `i18n`, `seo`, `a11y`, `cache`, `db`, `session`, `router`, `<plugin_name>`.
3. Ajoutez une virgule, un espace, et un court résumé `summary` des modifications, qui va apparaître dans le journal des modifications (changelog).

   Aucune ligne ne devrait dépasser 100 caractères en longueur, aussi gardez un résumé concis.

Bon résumé	Mauvais résumé (problème)
les propriétaires de pages (pages owners) voient leur propre bloc propriétaire sur les pages	correction de bug (vague)
la vue du graphique en barres n'échoue plus si "foo" n'est pas défini	met à jour views/default/bar.php de sorte que la vue bar ne soit plus… (info redondante)
réduit la mise en page de la rivière pour rentrer sur un iPhone	modifie la mise en page de la rivière (vague)
elgg_foo() gère les tableaux pour $bar	dans elgg_foo() vous pouvez maintenant passer un tableau pour $bar et la fonction sera… (déplacez les précisions vers la description)
supprime la couleur du lien de l'entête des commentaires dans la rivière	corrige la bdd pour que… (info redondante)
requiert un titre non-vide pour enregistrer les pages	peut enregistrer des pages sans titre (résume de manière confuse l'ancien comportement)

4. (recommandé) Sautez une ligne et ajoutez une `description` des modification. Incluez leur raison d'être, toute information sur la compatibilité ascendante ou descendante, et toute raison pour laquelle cette modification devait être faite d'une certaine manière. Exemple :

   > Nous accélérons la migration des tables en utilisant une seule requête INSERT INTO … SELECT au lieu de ligne par ligne. Cette migration se produit durant la mise à niveau vers Elgg 1.9.

   A moins que votre modification soit triviale/évidente, une description est requise.

5. Si le commit correspond à une demande (« issue ») Github, sautez une ligne et ajoutez `Fixes #` suivi par le numéro de demande. Par ex. `Fixes #1234`. Vous pouvez inclure de multiples demande en les séparant par des virgules.

   GitHub va fermer automatiquement la demande quand le commit est fusionné. Si vous souhaitez simplement faire référence à une demande, utilisez `Refs #`.

Quand c'est terminé, votre message de commit aura le format :

```
type(component): summary

Optional body
Details about the solution.
Opportunity to call out as breaking change.

Closes/Fixes/Refs #123, #456, #789
```

Voici un exemple d'un bon message de commit :

```
perf(upgrade): speeds up migrating remember me codes

We speed up the Remember Me table migration by using a single INSERT INTO ... SELECT␣
↪query instead of row-by-row.
This migration takes place during the upgrade to 1.9.

Fixes #6204
```

Pour valider les messages de commit localement, vérifiez que `.scripts/validate_commit_msg.php` est exécutable, et faites une copie ou un lien symbolique dans le dossier `.git/hooks/commit-msg`.

```
chmod u+x .scripts/validate_commit_msg.php
ln -s .scripts/validate_commit_msg.php .git/hooks/commit-msg/validate_commit_msg.php
```

Réécrire des messages de commits

Si votre PR ne se conforme pas au format standard de message de commit, nous vous demanderons de le réécrire.

Pour modifier seulement le dernier commit :

1. Amender le commit : `git commit --amend` (git ouvre le message dans un éditeur de texte).
2. Changer le message et enregistrer/quitter l'éditeur.
3. Forcez l'envoi de votre branche : `git push -f your_remote your_branch` (votre PR sera mis à jour).
4. Renommez le titre du PR pour correspondre

Sinon vous pourrez avoir besoin d'effectuer un « rebase » interactif :

1. Faites un rebase des derniers N commits : `git rebase -i HEAD~N` où N est le nombre. (Git va ouvrir le fichier `git-rebase-todo` pour modification)
2. Pour les commits qui ont besoin d'être modifiés, modifiez `pick` en `r` (pour « reword », reformulation) et enregistrez/quittez l'éditeur.
3. Modifiez le(s) message(s) de commit, enregistrez/quitter l'éditeur (git va prsenter un fichier pour chaque commit qui a besoin d'être reformulé).
4. `git push -f your_remote your_branch` pour forcer un push de la branche (qui met à jour votre PR).
5. Renommez le titre du PR pour correspondre

Standards de développements

Elgg utilise des jeux de standards qui sont basés en partie sur les standards PEAR et PSR2. Vous pouvez voir le jeu de règles dans `vendor/elgg/sniffs/elgg.xml`.

Pour vérifier les violations des standards de votre code (à condition que vous ayez installé Elgg avec les dépendances dev), exécutez :

```
phpcs --standard=vendor/elgg/sniffs/elgg.xml -s path/to/dir/to/check
```

Pour corriger automatiquement des violations réparables, exécutez :

```
phpcbf --standard=vendor/elgg/sniffs/elgg.xml path/to/dir/to/fix
```

Pour vérifier les répertoires du noyau, vous pouvez utiliser le raccourci `composer lint` et `composer lint-fixer`.

Tester

Elgg dispose de tests automatisés à la fois pour PHP et JavaScript. Toutes les nouvelles contributions doivent intégrer les tests appropriés.

Voir aussi :

Écrire des tests

Recommandations générales

Découpez les tests par comportements que vous souhaitez tester et utilisez des noms qui décrivent le comportement. Par ex. :

— Pas si bon : Une seule grosse méthode `testAdd()`.
— Mieux : Méthodes `testAddingZeroChangesNothing` et `testAddingNegativeNumberSubtracts`

Efforcez-vous d'utiliser des *designs à base de composants* qui permettent de tester en isolement, sans de grands graphes de dépendances ou d'accès à la base de données. L'injection de dépendances est ici critique.

Tests PHP

PHPUnit

Située dans `engine/tests/phpunit`, c'est notre suite de tests préférée. Elle n'utilise pas d'accès à la base de données, et n'a qu'un accès superficiel à l'API des entités.

— Si possible, nous vous encourageons à créer des composants qui sont testables dans cette suite.
— Envisagez de séparer le stockage de votre composant de sorte que la logique métier puisse être testée ici.
— Dépendez des classes `Elgg\Filesystem\*` plutôt que d'utiliser les fonctions du système de fichiers PHP.

SimpleTest

Le reste des fichiers dans `engine/tests` constitue notre suite d'intégration, pour tout ce qui demande un accès à la base de données ou aux APIs des entités.

— Notre objectif à long terme est de les minimiser et de les convertir en PHPUnit

Tester les interactions entre services

Dans l'idéal, vos tests devraient construire vos propres graphes d'objet isolés pour des manipulations directes, mais ce n'est pas toujours possible.

Si votre test dépend du Fournisseur de Service Elgg (`_elgg_services()` retourne un `Elgg\Di\ServiceProvider`), réalisez que cela maintient une instance singleton pour la plupart des services qu'il fournit, et que beaucoup de services conservent également leurs propres références locales pour ces services.

Du fait de ces références locales, replacer les services sur le SP (SP = « Service Provider », Fournisseur de Service) au sein d'un test n'aura souvent pas l'effet désiré. Au lieu de cela, vous pourriez avoir besoin d'utiliser la fonctionnalité disponible dans les services eux-mêmes :

— Les services `events` et `hooks` ont des méthodes `backup()` et `restore()`.
— Le service `logger` a des méthodes `disable()` et `enable()`.

Tests Jasmine

Les fichiers de test doivent être nommés `*Test.js` et devraient être placés soit dans `js/tests/` ou à côté de leurs fichiers sources dans `views/default/**.js`. Karma va automatiquement sélectionner les nouveaux fichiers `*Test.js` et exécuter ces tests.

Test générique

```
define(function(require) {
        var elgg = require('elgg');

        describe("This new test", function() {
                it("fails automatically", function() {
                        expect(true).toBe(false);
                });
        });
});
```

Effectuer les tests

Elgg utilise Karma avec Jasmine pour effectuer des tests unitaires JS.

Vous devez avoir nodejs et yarn installés.

Tout d'abord installez toute les dépendances de développement :

```
yarn
```

Exécutez les tests une seule fois puis quittez :

```
yarn test
```

Vous pouvez également exécuter les tests en continu pendant le développement de sorte qu'ils s'exécutent à chaque enregistrement :

```
karma start js/tests/karma.conf.js
```

Déboguer les tests JS

Vous pouvez exécuter la suite de test au sein des outils de développement de Chrome :

```
yarn run chrome
```

Ceci va renvoyer une URL telle que `http://localhost:9876/`.

1. Ouvrez l'URL dans Chrome, et cliquez sur « Debug ».
2. Ouvrez les outils de développement de Chrome et l'onglet Console.
3. Rechargez la page

Si vous modifiez un test, vous devrez quitter Karma avec `Ctrl-c` et le rédémarrer.

Meilleures pratiques de développement

Rendez votre code plus facile à lire, plus facile à maintenir, et plus facile à déboguer. Un usage consistant de ces recommandations signifie moins de travail de devinettes pour les développeurs, ce qui signifie des développeurs plus heureux, et plus productifs.

Développement en général

Ne Vous Répétez Pas

Si vous copiez-collez des morceaux significatifs de code, considérez qu'il y a une opportunité de réduire la duplication en introduisant une fonction, un argument supplémentaire, une vue, ou une nouvelle classe de composant.

Par ex. Si vous trouvez des vues qui sont identiques à l'exception d'une seule valeur, remaniez-les en une seule vue qui accepte une option.

Note : Dans la publicaiton d'une correction de bugs, *un peu de duplication est préférable à de la refactorisation*. Corrigez les bugs de la manière la plus simple possible, et refactorisez pour réduire la duplication dans la branche de la prochaine version mineure.

Adoptez SOLID et GRASP

Utilisez ces principes pour le design OO pour résoudre des problèmes en utilisant des composants couplés librement, et essayez de rendre tous les composants et le code d'intégration testables.

Les espacements sont gratuits

N'ayez pas peur d'utiliser des blocs de code séparés. Utilisez un espace unique pour séparer les paramètres des fonctions et les concaténations de chaînes.

Nom des variables

Utilisez des noms de variables auto-documentés. `$group_guids` est mieux que `$array`.

Evitez les doubles négations. Préférez `$enable = true` à `$disable = false`.

Nom des interfaces

Utilisez le motif `Elgg\{Namespace}\{Name}`.

N'ajoutez pas de préfixe `I` ou de suffixe `Interface`.

Nous n'utilisons aucun préfixe ou suffixe de sorte que nous sommes encouragés à :

- nommer les classes d'implémentation de manière plus descriptive (le nom « par défaut » est déjà pris).
- faites du typage sur les interfaces, parce que c'est la manière la plus courte et la plus simple de faire.

Nommez les implementations comme `Elgg\{Namespace}\{Interface}\{Implementation}`.

Fonctions

Autant que possible, ayez des fonctions/méthodes qui renvoient un type unique. Utilisez des valeurs vides telle que `array()`, `""`, ou `0` pour indiquer l'absence de résultat.

Faites attention aux cas où des valeurs de retour valides (telles que `"0"`) pourraient être interprétées comme vides.

Les fonctions qui ne déclenchent pas une exception lors d'une erreur devraient retourner `false` lorsqu'elles échouent.

Note : Particularly low-level, non-API functions/methods (e.g. entity_row_to_elggstar), which should not fail under normal conditions, should throw instead of returning false.

Les fonctions qui ne renvoient qu'un booléen devraient être préfixées par `is_` ou `has_` (par ex., `elgg_is_logged_in()`,`elgg_has_access_to_entity()`).

Syntaxe ternaire

Acceptable seulement pour des déclarations sur une seule ligne, non imbriquée

Minimisez la complexité

Minimisez les blocs imbriqués et les chemins d'exécution distinct au sein du code. Faites un Return rapide pour réduire les niveaux imbriqués et la charge cognitive lors de la lecture du code.

Utilisez les commentaires à bon escient

Les bons commentaires décrivent le « pourquoi. » Le bon code décrit le « comment. » Par ex. :

Mauvais :

```
// increment $i only when the entity is marked as active.
foreach ($entities as $entity) {
        if ($entity->active) {
                $i++;
        }
}
```

Bon :

```
// find the next index for inserting a new active entity.
foreach ($entities as $entity) {
        if ($entity->active) {
                $i++;
        }
}
```

Ajoutez toujours un commentaire s'il n'est pas évident que quelque chose doit être fait d'une certaine manière. D'autres développeurs qui regardent le code devraient être découragés de réorganiser le code d'une manière qui le casserait.

```
// Can't use empty()/boolean: "0" is a valid value
if ($str === '') {
    register_error(elgg_echo('foo:string_cannot_be_empty'));
    forward(REFERER);
}
```

Commitez de manière efficace

— Péchez par excès de commits atomiques qui sont précisément ciblés sur la modification d'un seul aspect du système.
— Eviter de mélanger des modifications qui ne sont pas liées ou des modifications importants des espaces. Les commits avec de nombreux changements sont effrayants et rendent les demandes de fusion (Pull Requests) difficiles à évaluer.
— Utilisez des outils git visuels pour façonner des diffs extrêmement précis et lisibles.

Incluez des tests

A chaque fois que c'est possible, *incluez des tests unitaires* pour le code que vous ajoutez ou modifiez.

Gardez les corrections de bugs simples

Évitez la tentation de refactoriser le code pour la publicaiton d'une correction de bug. Faire cela a tendance à introduire des régressions, à casser la fonctionnalité dans ce qui devrait être une version stable.

Recommandations PHP

Voici les standards de développement requis pour le noyau d'Elgg et tous les plugins associés. Les développeurs de plugins sont fortement encouragés à adopter ces standards.

Les développeurs devraient lire en premier le Guide des Standards de Développement PSR-2.

Les standards d'Elgg étendent PSR-2, mais diffèrent des manières suivantes :
— Indentez en utilisant le caractère tabulation, pas des espaces.
— Les parenthèses ouvrantes pour les classes, méthodes, et fonctions doivent être sur le même ligne.
— Si une ligne dépasse 100 caractères, envisagez de la remanier (par ex. en introduisant des variables).
— La compatibilité avec PSR-1 est encouragée, mais pas strictement requise.

Documentation

— Incluez des commentaires PHPDoc sur les fonctions et les classes (toutes les méthodes ; les propriétés déclarées quand c'est approprié), y compris les type et description de tous les paramètres.
— Dans les listes de déclarations `@param`, le début des noms de variables et des descriptions doivent être alignées.
— Annotate classes, methods, properties, and functions with `@internal` unless they are intended for public use, are already of limited visibility, or are within a class already marked as `@internal`.
— Utilisez `//` ou `/* */` pour les commentaires.
— Utilisez seulement les commentaires `//` à l'intérieur du corps des fonctions et méthodes.

Nommage

- Utilisez des traits de soulignements (« underscores ») pour séparer les mots dans les noms de fonctions, de variables, et les propriétés. Les noms de méthodes utilisent la syntaxe camelCase.
- Les noms de fonctions pour un usage public doivent commencer par `elgg_`.
- Tous les autres noms de fonctions doivent commencer par `_elgg_`.
- Nommez les globales et les contantes en `MAJUSCULE` (`ACCESS_PUBLIC`, `$CONFIG`).

Divers

Pour les pré-requis PHP, voyez `composer.json`.

N'utilisez pas les balises PHP raccourcies `<?` ou `<%`. Il est possible d'utiliser `<?=` dans la mesure où cela est toujours activé à partir de PHP 5.4.

Quand vous créez des chaînes de caractères avec des variables :
- utilisez des chaînes de caractères avec des guillemets doubles
- encadrez les variables avec des accolades seulement quand c'est nécessaire.

Mauvais (difficile à lire, mauvais usage des guillemets et des {}) :

```
echo 'Hello, '.$name."!  How is your {$time_of_day}?";
```

Bon :

```
echo "Hello, $name!  How is your $time_of_day?";
```

Supprimez les espaces de fin de ligne. Un moyen simple de faire cela avant votre commit est d'exécuter `php . scripts/fix_style.php` à partir de la racine de l'installation.

Validation de la valeur

Quand vous travaillez avec des saisies utilisateur, préparez les saisies en dehors de la méthode de validation.

Mauvais :

```
function validate_email($email) {
        $email = trim($email);

        // validate
}

$email = get_input($email);

if (validate_email($email)) {
        // the validated email value is now out of sync with an actual input
}
```

Bon :

```
function validate_email($email) {
        // validate
}

$email = get_input($email);
$email = trim($email);
```

(suite sur la page suivante)

(suite de la page précédente)

```
if (validate_email($email)) {
        // green light
}
```

Utilisez les exceptions

N'ayez pas peur d'utiliser les exceptions. Elles sont plus faciles à gérer que des sorties de fonction mixtes :

Mauvais :

```
/**
* @return string|bool
*/
function validate_email($email) {
        if (empty($email)) {
                return 'Email is empty';
        }

        // validate

        return true;
}
```

Bon :

```
/**
* @return void
* @throw InvalidArgumentException
*/
function validate_email($email) {
        if (empty($email)) {
                throw new InvalidArgumentException('Email is empty');
        }

        // validate and throw if invalid
}
```

Documenter les valeurs de retour

N'utilisez pas `@return void` sur des méthodes qui renvoient une valeur ou null.

Mauvais :

```
/**
* @return bool|void
*/
function validate_email($email) {
        if (empty($email)) {
                return;
        }

        // validate
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return true;
}
```

Bon :

```
/**
* @return bool|null
*/
function validate_email($email) {
        if (empty($email)) {
                return null;
        }

        // validate

        return true;
}
```

Recommandations CSS

Save the css in files with a `.css` extension.

Utilisez des abréviations quand c'est possible

Mauvais :

```
background-color: #333333;
background-image:  url(...);
background-repeat:  repeat-x;
background-position:  left 10px;
padding: 2px 9px 2px 9px;
```

Bon :

```
background: #333 url(...) repeat-x left 10px;
padding: 2px 9px;
```

Utilisez les traits d'union « -« , pas les underscores « _ »

Mauvais :

```
.example_class {}
```

Bon :

```
.example-class {}
```

Note : You should prefix your ids and classnames with text that identifies your plugin.

Une propriété par ligne

Mauvais :

```
color: white;font-size: smaller;
```

Bon :

```
color: white;
font-size: smaller;
```

Déclarations des propriétés

Celles-ci devraient être espacées comme ceci : `propriété: valeur;`

Mauvais :

```
color:value;
color :value;
color : value;
```

Bon :

```
color: value;
```

Préfixes fournisseurs (vendor)

- Regroupez les préfixes fournisseurs pour la même propriété
- La version la plus longue des préfixes fournisseurs en premier
- Incluez toujours la version sans préfixe fournisseur
- Ajoutez une ligne supplémentaire entre les groupes avec préfixes fournisseurs et les autres propriétés

Mauvais :

```
-moz-border-radius: 5px;
border: 1px solid #999999;
-webkit-border-radius: 5px;
width: auto;
```

Bon :

```
border: 1px solid #999999;

-webkit-border-radius: 5px;
-moz-border-radius: 5px;
border-radius: 5px;

width: auto;
```

Regroupez les sous-propriétés

Mauvais :

```
background-color: white;
color: #0054A7;
background-position: 2px -257px;
```

Bon :

```
background-color: white;
background-position: 2px -257px;
color: #0054A7;
```

Recommandations JavaScript

Les mêmes standards de formatage que PHP s'appliquent.

Toutes les fonctions devraient être dans l'espace de nom `elgg`.

Les expressions d'une fonction devraient se terminer par un point-virgule.

```
elgg.ui.toggles = function(event) {
        event.preventDefault();
        $(target).slideToggle('medium');
};
```

APIs obsolètes

De temps en temps, des fonctions et des classes doivent être dépréciées au profit de nouveaux remplacements. Dans la mesure où les auteurs de plugins tierce-partie s'appuient sur une API cohérente, la compatibilité ascendante doit être maintenue, mais ne sera pas maintenue indéfiniment dans la mesure où les auteurs de plugins sont supposés mettre à jour leurs plugins. Afin de maintenir la compatibilité ascendante, les API obsolètes vont suivre les recommandations suivantes :

— Les versions mineures (1.x) qui déprécient une API doivent inclure une fonction/classe d'emballage (« wrapper ») -ou tout autre moyen approprié- qui maintient la compatibilité ascendante, y compris tout bug de la fonction/classe originelle. Cette couche de compatibilité utilise `elgg_deprecated_notice('...', '1.11')` pour journaliser le fait que cette fonction est obsolète.

— La révision majeure suivante (2.0) supprime la couche de compatibilité. Toute utilisation de l'API obsolète devrait être corrigé auparavant.

3.6.2 Base de données

Contribuer aux modifications du schéma de la base de données

Contents

Migrations de la base de données

Elgg utilise Phinx pour gérer les migrations de la base de données.

Créer une migration

Pour créer une nouvelle migration, exécutez ce qui suit dans votre console :

```
phinx create -c engine/conf/migrations.php MigrationClassName
```

Ceci va générer un squelette de migration horodaté dans `engine/schema/migrations/`. Suivez la documentation de Phinx pour appeler les méthodes nécessaires pour modifier les tables de la base de données.

Exécuter une migration

Les migrations sont exécutées à chaque fois que vous exécutez `upgrade.php`. Si vous préférez exécuter les migrations manuellement, vous pouvez le faire via la ligne de commande :

```
phinx migrate -c engine/conf/migrations.php
```

Vérifiez la documentation Phinx pour des drapeaux additionnels qui vous autorisent à effectuer une seule migration ou un jeu de migrations dans un intervalle de temps.

3.6.3 Contribuer à la Documentation

La nouvelle documentation devrait s'intégrer correctement avec le reste de la documentation d'Elgg.

Contents

Tester les documentations localement

Elgg dispose d'un script grunt qui construit automatiquement les docs, les ouvre dans une fenêtre de navigateur, et les recharge automatiquement pendant que vous faites des modifications (le rechargement ne prend que quelques secondes). Vous aurez besoin d'avoir yarn et sphinx installés pour pouvoir utiliser ces scripts.

```
cd path/to/elgg/
yarn
grunt
```

C'est aussi simple que cela ! Grunt va continuer à s'exécuter, à vérifier les docs pour des modifications, et à reconstruire automatiquement.

Suivez l'organisation du document existant

L'organisation actuelle n'est pas nécessairement La Bonne Manière d'organiser les documentations, mais la cohérence est mieux que l'aléatoire.

intro/*

C'est tout ce que les tout nouveaux utilisateurs ont besoin de savoir (installation, fonctionnalités, licence, etc.)

admin/*

Guides pour les administrateurs. Orienté tâches.

guides/*

Guides de l'API pour les développeurs de plugins. Style recette de cuisine. Exemple solide. Elément de code solide. Cassé par les services (actions, i18n, routage, bd, etc.). Ceci devrait discuter seulement de l'API publique et de son comportement, pas des détails d'implémentation ou du raisonnement.

design/*

Documentation de conception (design docs) pour les personnes qui veulent avoir une meilleure compréhension de comment/pourquoi le noyau est construit de cette manière. Ceci devrait discuter des détails d'implémentaiton internes des divers services, de quels compromis ont été faits, et du raisonnement derrière la décision finale. Devrait être utile aux personnes qui veulent contribuer ou pour la communication entre les développeurs du noyau.

contribute/*

Guides de contributeurs pour les diverses manières dont des personnes peuvent participer au sein du projet.

appendix/*

Des informations plus détaillées/méta/de contexte à propos du projet (historique, feuille de route, etc.)

Utilisez « Elgg » d'une manière grammaticalement correcte

Elgg n'est pas un acronyme, l'écrire en majuscules (ELGG ou E-LGG) est incorrect. Veuillez ne pas le faire.

En Français, Elgg ne prend pas d'article quand il est utilisé comme un nom. Voici quelques exemples à imiter :

— "J'utilise Elgg pour mon site web"
— "Installez Elgg pour mettre votre communauté en ligne"

Quand il est utilisé comme adjectif, l'article s'applique au nom principal, aussi vous devriez en utiliser un. Par exemple :

— « Allez sur le site de la communauté Elgg pour trouver de l'aide. »
— « J'ai construis un réseau avec Elgg hier »

Ce conseil peut ne pas être valable pour d'autres langues que l'anglais.

Évitez les pronoms à la première personne

Faites référence au lecteur comme « vous ». Ne vous incluez pas dans la narration habituelle.

Avant :

> Quand nous aurons terminé l'installation d'Elgg, nous irons rechercher quelques plugins !

Après :

> Quand vous aurez terminé l'installation d'Elgg, recherchez quelques plugins !

Pour faire référence à vous-même (à éviter si possible), utilisez votre nom et écrivez à la troisième personne. Ceci permet aux futurs lecteurs/éditeurs de savoir de qui les opinions sont exprimées.

Avant :

> Je pense que le meilleur moyen pour faire X est d'utiliser Y.

Après :

> Evan pense que le meilleur moyen pour faire X est d'utiliser Y.

Supprimer le délayage

Avant :

> Si vous souhaitez utiliser une bibliothèque javascript tierce-partie au sein du framework Elgg, vous devriez prendre soin d'appeler la fonction `elgg_register_js` pour l'enregistrer.

Après :

> Pour utiliser une bibliothèque javascript tierce-partie, appelez `elgg_register_js` pour l'enregistrer.

Préférez les dates absolues aux dates relatives

Il est difficile de dire quand une phrase ou un paragraphe particuliers ont été écrits, aussi les dates relatives deviennent vite incompréhensibles. Les dates absolues donnent également au lecteur une bonne indication de si un projet a été abandonné, ou si un conseil est toujours d'actualité.

Avant :

> Récemment le truc a été machin. Bientôt, la chose sera machin aussi.

Après :

> Récemment (à partir de septembre 2013), le truc a été machin. Il est prévu que la chose soit également machin d'ici octobre 2013.

Ne rappelez pas au lecteur de contribuer

Concentrez-vous pour ne traiter que du sujet en question. Des sollicitations constantes pour du travail gratuit sont agaçantes et donnent l'impression que le projet est dans le besoin. Si des personnes souhaitent contribuer au projet, elles peuvent visiter le guide du contributeur.

3.6.4 Internationaliser la documentation

Quand vous modifiez la documentation, pensez à mettre à jour les modèles de traduction de la documentation avant de faire un commit :

```
cd docs/
make gettext
```

Pour plus d'information, voyez http://www.sphinx-doc.org/en/stable/intl.html#translating-with-sphinx-intl

Points d'attention spécifiques

Quand vous traduisez la documentation soyez conscients de la syntaxe spécifique dans les fichiers de documentation.

Liens de traduction

— Traduisez le texte dans des liens anonymes (par ex., `` `prononciation`__ ``), mais maintenez l'ordre de tous les liens anonymes dans un unique bloc. S'il y a deux liens anonymes à traduire dans un seul bloc de traduction, il ne doivent pas être inversés l'un par rapport à l'autre.
— Traduisez le texte des liens nommés (par ex., `` `site de démo`_ ``) mais seulement si vous maintenez le noms en utilisant la bonne syntaxe rST. Dans ce cas ce devrait être `` `traduction de "site de démo" <demo site_>`_ ``.

NE traduisez PAS

— Tout ce qui se situe entre des caractères trait vertical ne devrait pas être traduit (par ex., master).
— Le code, à moins que ce ne soit un commentaire dans le code.

3.6.5 Traductions

Les traductions démultiplient l'impact que Elgg peut avoir en le rendant accessible à un plus grand pourcentage du monde.

La communauté sera toujours reconnaissante à ceux qui travaillent dur pour fournir des traductions de haute qualité pour l'UI et la documentation de Elgg.

Transifex

Toutes les traductions pour le projet Elgg sont organisées à travers Transifex.

https://www.transifex.com/organization/elgg

Les auteurs de plugins sont encouragés à coordonner les traductions via Transifex également, de sorte que la communauté entière puisse être unie, et que cela facilite la possibilité pour les traducteurs de contribuer à la traduction de n'importe quel plugin dans l'écosystème Elgg.

Récupérer les traductions

Les traductions faites dans Transifex ont besoin d'être récupérées périodiquement dans le dépôt de code d'Elgg. Ceci peut être effectué avec le script `.scripts/languages.php` livré avec le code source de Elgg.

Les pré-requis pour exécuter le script sont :

- Accès à la ligne de commande
- Git
- Transifex CLI tool

Le script va effectuer les étapes suivantes :

1. Créer une nouvelle branche git nommée `{branch}_i18n_{timestamp}`
2. Récupérer les traductions pour toutes les langues qui ont 95% des chaînes traduites
3. Retirer les éventuels codes de langue invalides
4. Commiter les modifications vers la branche

Après cela, faites un push de la branche vers Github et faites une nouvelle Pull Request.

Par exemple, si vous voulez récupérer les traductions pour la branche `3.x`, exécutez les commandes suivantes :

```
php .scripts/languages.php 3.x
git push -u your_fork 3.x_i18n_1515151617
```

Exécutez la commande sans paramètre pour avec des informations plus détaillées sur son utilisation.

Configuration Transifex

La configuration pour Transifex se trouve dans le code source d'Elgg dans le fichier `.tx/config`.

Ce fichier définit :

- Le projet Transifex associé avec la version majeure d'Elgg
- L'emplacement des tous les fichiers qui ont du contenu traduisible

Lisez la documentation Transifex pour plus d'information.

Nouvelle version majeure d'Elgg

Chaque version majeure d'Elgg doit disposer de son propre projet dans Transifex. De cette manière nous pouvons nous assurer que les chaînes ajoutées et retirées entre les versions ne créent pas de conflit l'une avec l'autre. Par exemple une clef de traduction retirée dans Elgg 3 ne devrait pas être retirée des traductions faites pour Elgg 2. De même, une nouvelle chaîne ajoutée seulement pour Elgg 3 ne devrait pas être incluses dans les traductions prévues pour Elgg 2.

Le processus pour mettre en place une nouvelle version majeure est le suivant :

1. Récupérez les dernières traductions depuis Transifex vers la précédente version majeure
2. Fusionnez la branche git de la version précédente vers la nouvelle afin de s'assurer que toutes les dernières clefs de traduction sont présentes
3. Créez un nouveau projet Transifex sur https://www.transifex.com/elgg/
4. Mettez à jour le fichier `.tx/config` dans la branche de développement de la nouvelle version majeure
   - Mettez à jour la configuration pour pointer vers le nouveau projet Transifex
   - Retirez la configuration des plugins retirés
   - Ajoutez la configuration pour les nouveaux plugins
5. Poussez les sources de traduction vers le nouveau projet Transifex avec la commande :

```
tx push -s
```

6. Copiez le nouveau fichier de configuration temporairement (pas de commit) vers la version majeure précédente, et poussez les traductions existantes depuis là vers le nouveau projet :

```
tx push -t -f --no-interactive
```

Plus tard, une fois que la branche dédiée (par ex. `3.x`) a été créée pour la version majeure, configurez Transifex pour récupérer automatiquement les nouvelles clefs à partir d'elle dans https://www.transifex.com/elgg/elgg-core-3/content/. De cette manière vous n'avez pas à répéter manuellement l'étape 5 à chaque fois que de nouvelles clefs de traduction sont ajoutées.

Il est important de toujours avoir une branche `n.x` en plus des branches créées pour des versions mineures spécifiques (`n.1`, `n.2`, etc.). De cette manière les URLs des sources automatiquement mises à jour n'ont pas besoin d'être mises à jour à chaque fois qu'une nouvelle branche mineure est créée.

3.6.6 Signaler des problèmes

Signalez bugs et demandes de fonctionnalités sur https://github.com/Elgg/Elgg/issues. Voyez ci-dessous pour des recomandations.

LIMITES DE RESPONSABILITÉ

Attention : Les problèmes de sécurité devraient être signalés à **security @ elgg . org** ! Merci de ne publier aucun faille de sécurité sur github ! !

Note : Les demandes d'aide relèvent du site de la communité. Les tickets avec des demandes d'aide seront fermés.

Important : Nous ne pouvons donner aucune garantie sur quand votre ticket sera résolu.

Rapports de bugs

Avant de soumettre un rapport de bug :
- Recherchez un ticket existant à propos du problème que vous rencontrez. Ajouter toute information supplémentaire à cet endroit.
- Vérifiez que le problème est reproductible
  - Sur la dernière version d'Elgg
  - Avec tous les plugins tierce-partie désactivés

Liste de contrôle pour un bon signalement de bug :
- Comportement attendu et comportement constaté
- Des étapes claires pour reproduire le problème
- La version d'Elgg que vous utilisez
- Les navigateurs affectés par ce problème

Demandes de fonctionnalité

Avant de soumettre une demande de fonctionnalité :
- Vérifiez sur le site de la communauté si un plugin existe avec les fonctionnalités dont vous avez besoin.
- Envisagez si vous le pouvez de *développer un plugin* qui fasse ce dont vous avez besoin.
- Recherchez parmi les tickets fermés pour voir si quelqu'un d'autre a suggéré la même fonctionnalité, mais que cela a été refusé. Vous devrez pouvoir expliquer pourquoi votre suggestion devrait être examinée cette fois-ci.

Liste de contrôle pour une bonne demande de fonctionnalité :
- Explication détaillée de la fonctionnalité
- Cas d'usages réels
- API proposée

3.6.7 Devenir un soutien financier

Tous les fonds recueillis par l'intermédiaire du réseau de supporters Elgg vont directement pour :
- Développement du noyau Elgg
- La fourniture d'infrastructures (elgg.org, github, etc)

Il s'agit d'un excellent moyen d'aider au développement d'Elgg !

Avantages

Pour seulement $50 par an pour les individus ou $150 par an pour les organisations, vous pouvez être listé comme soutien sur notre page des soutiens. Les soutiens d'Elgg apparaissent dans cette liste sauf s'ils demandent à ne pas l'être.

Les soutiens ont l'autorisation d'ajouter ce logo officiel sur leur site s'ils le souhaitent :

Limite de responsabilité

Nous avons une politique de non-remboursement sur les souscriptions des soutiens. Si vous souhaitez arrêter votre soutien, allez sur Paypal et annulez votre souscription. Vous ne serez pas débité l'année suivante.

Etre un soutien d'Elgg ne donne en aucun cas le droit à un individu ou à une organisation le droit de parler au nom du projet Elgg, de commercialiser en son nom, ou de laisser entendre qu'ils sont liées au projet Elgg. Ils peuvent, toutefois, mentionner le fait qu'ils sont soutien du projet Elgg.

Si vous avez des questions à propos de cet avertissement, écrivez à info@elgg.org.

Nous nous réservons le droit de retirer ou refuser une souscription sans avertissement préalable à notre entière discrétion. Il n'y a pas de politique de remboursement.

S'il n'y a pas d'utilisation évidente de Elgg, votre site sera relié avec l'attribut « nofollow ».

S'inscrire

Si vous souhaitez devenir un supporter d'Elgg :
- lisez l'*avertissement* ci-dessus
- sur la page des soutiens, souscrivez via Paypal
- envoyez un email à info@elgg.org avec :
  - la date de votre souscription
  - votre nom (et le nom de l'organisation, s'il y a lieu)
  - votre site web
  - votre profil sur la communauté Elgg

Une fois que tous les détails ont été reçus, nous vous ajouterons à la liste appropriée. Merci pour votre soutien !

3.6.8 Ajouter un Service à Elgg

Le *guide sur les services* présente des informations générales sur l'utilisation des services Elgg.

Pour ajouter un nouvel objet service à Elgg :

1. Annotate your class as `@internal`.
2. Ouvrez la classe `Elgg\Di\ServiceProvider`.
3. Ajoutez une annotaiton `@property-read` pour votre service au tout début. Ceci permet aux EDIs et aux analyseurs de code statique de comprendre le type de la propriété.
4. Pour le constructeur, ajoutez le code pour indiquer au fournisseur de service quoi retourner. Voyez la classe `Elgg\Di\DiContainer` pour plus d'information sur comment le conteneur DI (« Dependency Injection ») d'Elgg fonctionne.

A ce stade votre service sera disponible depuis l'objet fournisseur de service, mais ne sera pas encore accessible aux plugins.

Injectez vos dépendances

Concevez votre constructeur de classe de sorte qu'il *demande* les dépendances nécessaires plutôt que de les créer ou d'utiliser `_elgg_services()`. La méthode `setFactory()` du fournisseur de service fournit l'accès à l'instance du fournisseur de service dans la méthode de votre fabrique.

Voici un exemple de fabrique de service `foo`, qui injecte les services `config` et `db` dans le constructeur :

```
// in Elgg\Di\ServiceProvider::__construct()

$this->setFactory('foo', function (ServiceProvider $c) {
    return new Elgg\FooService($c->config, $c->db);
});
```

La liste complète des services internes peut être vue dans les déclarations `@property-read` au début de `Elgg\Di\ServiceProvider`.

Avertissement : Evitez de faire du travail dans le constructeur de votre service, en particulier si cela requiert des requêtes sur la base de données. Actuellement les tests PHPUnit tests ne peuvent pas les effectuer.

Faire qu'un service fasse partie de l'API publique

Si votre service est conçu pour être utilisé par des développeurs de plugins :

1. Faites une interface `Elgg\Services\<Name>` qui ne contient que ces méthodes nécessaires dans l'API publique.
2. Faites que la classe de votre service implémente cette interface.
3. Pour les méthodes qui sont dans l'interface, déplacez la documentation dans l'interface. Vous pouvez simplement utiliser `{@inheritdoc}` dans les PHPDocs des méthodes de votre classe effective.
4. Documentez votre service dans `docs/guides/services.rst` (ce fichier)
5. Ouvrez le test PHPUnit `Elgg\ApplicationTest` et ajoutez la clef de votre service au tableau `$names` dans `testServices()`.
6. Ouvrez la classe `Elgg\Application`.
7. Ajoutez une déclaration `@property-read` pour documenter votre service, mais utilisez votre **interface** pour le type, et *pas* le nom de la classe de votre service.
8. Ajoutez la clef de votre service au tableau dans la propriété `$public_services`, par ex. `'foo' => true,`

Désormais votre service sera disponible via l'accès à la propriété sur l'instance `Elgg\Application` :

```
// using the public foo service
$three = elgg()->foo->add(1, 2);
```

Note : Pour des exemples, voyez le service `config`, qui comprend l'interface `Elgg\Services\Config` et la classe d'implémentation effective `Elgg\Config`.

Cycle de vie d'un Service et Fabriques (« factories »)

Par défaut, les services enregistrés sur le fournisseur de service sont « partagés », ce qui signifie que le fournisseur de service va conserver l'instance créée pour le reste de la requête, et servir la même instance à tout ce qui demande la propriété.

Si vous avez besoin que les développeurs puissent construire des objets qui soient pré-coonnectés aux services Elgg, vous pouvez avoir besoin d'ajouter une méthode de fabrique publique à `Elgg\Application`. Voici un exemple qui retourne une nouvelle instance en utilisant les services d'Elgg :

```
public function createFoo($bar) {
    $logger = $this->services->logger;
    $db = $this->services->db;
    return new Elgg\Foo($bar, $logger, $db);
}
```

3.6.9 Écrire des tests

Contents

Vision

Nous voulons *rendre les tests manuels inutiles* pour les développeurs du noyau, les auteurs de plugins, et les administrateurs de sites en rendant possible et en faisant la promotion de tests rapides et automatisés à tous les niveaux de Elgg.

Nous avons hâte d'un monde dans lequel les développeurs du noyau n'ont pas besoin de faire de tests manuels pour vérifier la correction du code contribué à Elgg. De manière similaire, nous envisageons un monde dans lequel les administrateurs de site peuvent mettre à niveau et installer de nouveaux plugins avec la certitude que tout fonctionne bien ensemble.

Effectuer les tests

Suite de tests du noyau Elgg

Actuellement nos tests se partagent en deux parties :

— Les test PHPUnit sont situés dans `/tests/phpunit` – ce sont strictemetn des tests unitaires pour le moment. Pas d'intégration ni de tests de bout en bout.
— Les tests SimpleTest sont situés dans `/tests/` – Ceux-ci sont supportés, mais nous encourageons fortement la création de nouveaux tests dans PHPUnit et planifions de migrer ceux-ci vers PHPUnit dans un futur proche le plus adéquat.

Dans la mesure où il existe un fichier de configuration `phpunit.xml` à la racine d'Elgg, les tests devraient être aussi simples que :

```
git clone http://github.com/Elgg/Elgg
cd Elgg
phpunit
```

Tests des plugins

Idéalement, les plugins sont configurés d'une telle manière qu'ils peuvent faire l'objet de tests unitaires de la même manière que le noyau d'Elgg. Les développeurs de plugins sont libres d'implémenter leurs propres méthodes pour les tests unitaires, mais nous encourageons tout le monde à les rendre aussi simple que pour le noyau de Elgg :

```
git clone http://github.com/developer/elgg-plugin plugin
cd plugin
phpunit
```

Tests de bout en bout

Dans la mesure où les plugins de Elgg ont une telle capacité à remplacer, filtrer et modifier Elgg et le comportement d'autres plugins, il est important de pouvoir exécuter des tests de bout en bout sur un serveur de pré-production avec votre configuration finale avant de déployer vers la production.

Note : ToDo : Faciliter l'exécution de tous les tests d'intégration et d'acceptabilité depuis le panneau admin en utilisant la configuration courante du plugin. (sans s'inquiéter de corruption de la base de données !)

Motivation

Brièvement, les gains escomptés des tests sont :

- Une confiance accrue dans le système.
- Une plus grande liberté pour remanier le code.
- Une documentation intégrée et à jour.

Nous adorons les contributions communautaires, mais afin de maintenir la stabilit » nous ne pouvons pas accepter de contributions extérieures sans vérifier au préalable leur correction. En promouvant les tests automatisés, les développeurs du noyau peuvent éviter les ennuis d'une vérification manuelle avant d'accepter un patch. Cela signifie également que les développeurs externes n'ont pas à perdre de temps à gagner la confiance de l'équipe du noyau. SI un patch arrive et qu'il dispose de tests pour le vérifier, nous pouvons être confiants dans le fait qu'il fonctionne sans avoir besoin de se soucier de la réputation de l'auteur.

Notez que ces bénéfices peuvent également s'étendre au dépôt des plugins. Les propriétaires de sites sont encouragés à « tester les plugins minutieusement » avant de les déployer sur un site de production. A la date de mars 2013, ceci signifie tester manuellement toutes les fonctionnalités que le plugin promet d'offrir. Mais Elgg fournit un grand nombre de fonctionnalités, et il n'est pas raisonnable de tester *chacune* d'entre elles sur *chaque navigateur* que vous voulez supporter et sur *chaque appareil* que vous voulez supporter ? Mais que se passerait-il si les développeurs de plugins pouvaient écrire des tests pour leurs plugins, et si les propriétaires de sites pouvaient simplement exécuter ces tests pour tous les plugins installés pour vérifier si la fonctionnalité est bien conservée ? Alors ils ne seraient pas limités à seulement prendre des plugins de développeurs « de confiance » ou des versions « stables ». Ils pourraient voir que, en réalité, rien n'a été cassé quand ils ont mis à niveau ce plugin critique depuis la version 1.3 vers la 2.5, et passer la mise à niveau en production en toute confiance.

La raison pour laquelle ce n'est pas le cas actuellement est parce qu'Elgg lui-même n'est pas encore si facile à tester à ce niveau. Nous voulons changer cela.

Stratégie

Nous avons quelques principes pour nous guider, dont nous pensons qu'ils seront utiles pour faire de cette vision une réalité.

En bref, nous prônons :

- L'intégration continue – si Travis n'est pas content, nous non plus
- L'injection des dépendances – Pour créer un code hautement testable et modulaire
- BDD (« Behaviour Driven Development ») – Les tests devraient vérifier des fonctionnalités et fournir de la documentation, pas reprendre l'API de la Classe

Intégration continue

Nous exécutons tous nos tests sur Travis CI de sorte que nous puissions avoir un feedback en temps réel sur la correction des demandes de fusion entrantes et des développements au fur et à mesure de leur évolution. **Si Travis n'est pas d'accord, nous ne faisons pas de commit sur le dépôt.** Ceci nous permet de fusionner les demandes de fusion à un rythme rapide, dès lors qu'elles ont passé les tests. Cela nous permet également de rejeter les demandes de fusion sans investigation poussée si elles ne passent pas les tests. Nous pouvons aller au-delà de la question « est-ce que ça marche ou pas » et évoquer les choses dont les humains ont besoin de parler : la conception de l'API, l'utilité pour le projet, est-ce que cela fait partie du noyau ou pas, etc. Nous voulons qu'autant de fonctionnalités que possible fournies par le noyau d'Elgg puissent être vérifiées automatiquement par des tests exécutés sur Travis.

Injection des dépendances

Afin de maximiser la testabilité, **toutes les dépendances doivent être explicites**. Les fonctions globales, les Singletons, et les localisateurs de service sont mortels pour la testabilité parce qu'il est impossible de dire quelles dépendances se cachent sous le capot, et qu'il est encore plus difficile de simuler (« mock out ») ces dépendances. La capacité de simulation est critique parce que vous voulez que vos tests unitaires ne testent qu'une seule classe à la fois. Les échecs d'un test dans un TestCase ne devraient pas résulter d'une rupture dans une dépendance ; les échecs des tests ne devraient indiquer de rupture que dans la classe en train d'être testée. Ceci rend tout plus facile à déboguer. A la date de mars 2013, la majorité d'Elgg assume et utilise toujours un état global, et ceci a rendu Elgg et les plugins Elgg historiquement très difficiles à tester. Fort heureusement nous avons pris une autre direction depuis, et une grande partie du travail dans Elgg 1.9 a consisté à refactorer des composants du noyau pour qu'ils soient plus injectables sous forme de dépendances. Nous récoltons déjà les bénéfices de cet effort.

Développement orienté comportement

Pour nous ceci signifie que **nous nommons les tests d'après les fonctionnalités plutôt que les méthodes**. Quand vous testez des fonctionnalités, vous êtes encouragés à écrire moins de tests, plus petits, et plus logiques. Quand un test échoue, cela permet de savoir exactement quelle fonctionnalité est compromise. De plus, en nommant vos tests d'après les fonctionnalités, la liste des tests fournit une documentation des fonctionnalités que le système supporte. La documentation est typiquement quelque chose de problématique à conserver à jour, mais quand la documentation et la vérification sont une seule et même chose, il devient très facile de maintenir la documentation à jour.

Considérez ces deux méthodes de test :

- `testRegister()`
- `testCanRegisterFilesAsActionHandlers()`

En regardant seulement les noms, `testRegister` vous indique que la classe testée a probablement une méthode nommée « register ». Si ce test est passé, il vérifie probablement qu'elle se comporte correctement, mais ne vous dit pas ce qui caractérise un comportement correct, ou ce que l'auteur original du test essayait de vérifier. Si cette méthode a plusieurs usages corrects que vous devez tester, cette convention de nommage succincte vous encourage également à écrire un très long test qui teste toutes les conditions et fonctionnalités de ladite méthode. L'échec du test pourrait être causée par la compromission de n'importe laquelle d'entre elles, et cela prendra plus de temps pour savoir où se situe le véritable problème.

De l'autre côté, `testCanRegisterFilesAsActionHandlers` vous indique qu'il y a ces choses appelées « actions » qui ont besoin d'être « gérées » (« handled »), et que ces fichiers peuvent être enregistrés comme des gestionnaires valides pour les actions. Ceci expose mieux aux nouveaux venus la terminologie du projet et communique clairement l'intention du test pour ceux qui sont déjà familiers avec la terminologie.

Pour un bon exemple de ce que nous recherchons, regardez `/tests/phpunit/Elgg/ViewServiceTest.php`

3.6.10 Tâches du noyau

Certaines tâches autour d'Elgg sont réservées à l'équipe des développeurs du noyau dans la mesure où elles demandent certaines permissions spéciales. Les guides ci-dessous décrivent le processus pour ces actions.

Déplacer un plugin vers son propre dépôt

Contents

Étapes d'extraction d'un plugin

Déplacez le code vers son propre dépôt

Suivez le guide Github pour transformer un sous-dossier en un nouveau dépôt (Splitting a subfolder out into a new repository). Ceci va faire en sorte que l'historique des commits soit préservé.

Dépendances

Si le plugin a des dépendances sur n'importe quelles bibliothèques externes, assurez-vous que ces dépendances soient gérées. Par exemple, si une bibliothèque PHP est requise qui est livrée avec le noyau d'Elgg, assurez-vous de l'ajouter dans le `composer.json` de ce plugin dans la mesure où vous ne pouvez pas vous appuyer sur le fait que le noyau d'Elgg conserve cette bibliothèque.

Commitez le code

Au cours du guide GitHub un nouveau dépôt est créé pour le plugin que vous essayez de déplacer.

Dans la mesure où une tentative a déjà été faite pour extraire certains des plugins vers leur propre dépôt il se peut que le dépôt existe déjà.

Si le dépôt n'existait pas pour le plugin, assurez-vous que vous le créez dans Elgg organisation.

Si le dépôt existe déjà, le meilleur moyen de mettre à jour le code serait en faisant une Pull Request. Ceci risque cependant probablement d'échouer à cause d'une différence dans la manière dont le dépôt a été initialement créé (comme discuté dans Problème avec GitHub).

Les dépôts initiaux ont été créés avec

```
git subtree split
```

et le guide appelle

```
git filter-branch --prune-empty --subdirectory-filter
```

Ceci va laisser une différence dans les commits que Github est incapable de résoudre. Dans ce cas vous devrez forcer un push des modifications vers le dépôt du plugin Elgg existant.

Avertissement : Dans la mesure où cela va réécrire tout l'historique du dépôt du plugin, assurez-vous que vous savez que c'est bien ce que vous souhaitez faire.

Packagist

Assurez-vous que le `composer.json` du plugin contient bien toutes les informations pertinentes. Voici un exemple :

```
{
        "name": "elgg/<name of the repository>",
        "description": "<a description of the plugin>",
        "type": "elgg-plugin",
        "keywords": ["elgg", "plugin"],
        "license": "GPL-2.0-only",
        "support": {
                "source": "https://github.com/elgg/<name of the repository>",
                "issues": "https://github.com/elgg/<name of the repository>/issues"
        },
        "require": {
                "composer/installers": ">=1.0.8"
        },
        "conflict": {
                "elgg/elgg": "< <minimal Elgg required version>"
        }
}
```

La règle `conflict` est ici pour aider à éviter l'installation de ce plugin dans une version d'Elgg non supportée.

Ajoutez le dépôt à Packagist, pour les dépôts existants ceci a déjà été fait. Assurez-vous que Packagist est mis à jour correctement avec tous les commits.

Tagguer une release

Afin que Composer soit capable de mettre en cache le plugin pour une installation plus rapide, une release doit être créée sur le dépôt. A priori, la première version qui doit être tagguée est la même version que celle mentionnée dans le `manifest.xml`. Après cela le développement peut commencer, suivi par le schéma de version Semver.

Traductions

Si les traductions pour le plugin doivent être gérées par Transifex, ajoutez le plugin à Transifex.

Nettoyage du noyau Elgg

Maintenant que le plugin a été déplacé dans son propre dépôt, il est temps de faire une Pull Request sur le noyau d'Elgg pour supprimer le code originel.

Supprimez le plugin

— Supprimez le plugin du dossier `mod`
— Recherchez le nom du plugin dans le noyau pour trouver toute référence qui aurait également besoin d'être retirée (par ex. anciennes docs, tests spéciaux, etc.)

Traductions

Dans la mesure où le plugin ne fait plus partie du noyau Elgg, assurez-vous que la configuration de Transifex ne contient plus le plugin.

Plugins joints

Si le plugin est toujours livrés conjointement avec la publicaiton d'une nouvelle version de Elgg, assurez-vous d'ajouter le plugin dans `composer.json`.

Composer

Vérifiez les dépendances composer du noyau pour identifier si des exigences qui étaient spécifiques pour le plugin retiré peuvent également être retirées des dépendances du noyau.

Documentation

Ajoutez une mention dans la documentation *Upgrade Notes* qui indique que le plugin a été retiré du noyau de Elgg.

Processus de publication d'une release

Publier une nouvelle version d'Elgg.

Voici le processus suivi par l'équipe du noyau pour publier une nouvelle release d'Elgg. Nous publions cette information dans un esprit d'ouverture, et pour faciliter l'intégration de nouveaux membres de l'équipe.

Contents

Pré-requis

— Accès SSH à elgg.org
— Accès aux commits sur http://github.com/Elgg/Elgg
— Accès admin à https://elgg.org/
— Accès au compte Twitter Twitter account
— Node.js et Yarn installés
— Sphinx installé (`easy_install sphinx && easy_install sphinx-intl`)
— Client Transifex installé (`easy_install transifex-client`)
— Compte Transifex avec accès au projet Elgg

Fusionnez les commits à partir des branches les plus basses

Déterminez la branche LTS. Nous devons fusionner tous les nouveaux commits jusqu'ici en passant par les autres branches.

Pour chaque branche

Vérifiez la branche, assurez-vous qu'elle est à jour, et créez une nouvelle branche de travail avec la fusion. Par ex. ici nous fusionnons les commits de la 1.12 dans la 2.0 :

```
git checkout 2.0
git pull
git checkout -b merge112
git merge 1.12
```

Note : Si elle est déjà à jour (aucun commit à fusionner), nous pouvons nous arrêter ici pour cette branche.

S'il y a des conflits, résolvez-les, `git add .`, et `git merge`.

Faites un PR pour la branche et attendez le résultat des tests automatiques et l'approbation par un ou plusieurs autre(s) dev(s).

```
git push -u my_fork merge112
```

Une fois la fusion effectuée, nous répéterions le processus pour fusionner les commits de la 2.0 dans la 2.1.

Première nouvelle version mineure/majeure stable

Mettez à jour *Support policy* pour inclure la date de la nouvelle version mineure/majeure et complétez les vides de la version précédente.

Préparer la sortie

Mettez à jour votre clone git local.

Fusionnez les derniers commits vers le haut en commençant par la branche supportée la plus basse.

Visitez `https://github.com/Elgg/Elgg/compare/<new>...<old>` et soumettez le PR si quoi que ce soit a besoin d'être fusionné plus haut.

Installez les pré-requis :

```
yarn install elgg-conventional-changelog
easy_install sphinx
easy_install sphinx-intl
easy_install transifex-client
```

Note : Sur Windows, vous devrez exécuter ces commandes dans une console avec des privilèges admin

Exécutez le script `release.php`. Par exemple, pour publier 1.12.5 :

```
git checkout 1.12
php .scripts/release.php 1.12.5
```

Ceci crée une branche `release-1.12.5` dans votre repo local.

Ensuite, naviguez manuellement jusqu'à la page `/admin/settings/basic` et vérifiez qu'elle se charge. Si ce n'est pas le cas, il se peut qu'un fichier de traduction issu de Transifex comporte une erreur de syntaxe. Corrigez l'erreur et amendez votre commit avec le nouveau fichier :

```
# only necessary if you fixed a language file
git add .
git commit --amend
```

Ensuite, soumettez un PR via Github pour les tests automatisés et l'approbation par un autre développeur :

```
git push your-remote-fork release-1.12.5
```

Tagguez la release

Une fois approuvée et fusionnée, tagguez les release :

```
git checkout release-${version}
git tag -a ${version} -m'Elgg ${version}'
git push --tags origin release-${version}
```

Ou créez une release sur Github

— Allez sur les releases
— Cliquez sur "Create a new release"
— Saisissez la version

— Sélectionnez la bonne branche (par ex. 1.12 pour une release 1.12.x, 2.3 pour une release 2.3.x, etc.)
— Définissez le titre de la release tel que “Elgg {version}”
— Collez la partie de CHANGELOG.md relative à cette release dans la description

Un peu d’administration pour finir

— Marquez les jalons de release Github comme terminés
— Déplacez les tickets non résolus des jalons publiés vers des jalons ultérieurs

Mettez à jour le site web

— ssh vers elgg.org
— Clonez https://github.com/Elgg/elgg-scripts

Construisez le package zip

Utilisez `elgg-scripts/build/elgg-starter-project.sh` pour générer le fichier .zip. Exécutez sans argument pour voir son utilisation.

```
# login as user deploy
sudo -su deploy

# regular release
./elgg-starter-project.sh master 3.0.0 /var/www/www.elgg.org/download/

# MIT release
./elgg-starter-project.sh master 3.0.0-mit /var/www/www.elgg.org/download/
```

Note : Pour les releases Elgg 2.x utilisez la branche `2.x` du starter-project (par ex. `./elgg-starter-project.sh 2.x 2.0.4 /var/www/www.elgg.org/download/`)

— Vérifiez que `vendor/elgg/elgg/composer.json` dans le fichier zip a bien la version attendue.
— Si ce n’est pas le cas, assurez-vous que GitHub a bien le tag de release, et que le projet de démarrage a un élément compatible `elgg/elgg` dans la liste « requires » de composer.

Construire les packages zip 1.x

Utilisez `elgg-scripts/build/build.sh` pour générer le fichier .zip. Exécutez sans argument pour voir L’utilisation.

```
# regular release
./build.sh 1.12.5 1.12.5 /var/www/www.elgg.org/download/

# MIT release
./build.sh 1.12.5 1.12.5-mit /var/www/www.elgg.org/download/
```

Mettez à jour la page de téléchargement de elgg.org

— Clonez https://github.com/Elgg/community
— Ajoutez la nouvelle version dans `classes/Elgg/Releases.php`
— Committez et poussez les modifications (push)
— Mettez à jour le plugin sur www.elgg.org

```
composer update elgg/community
```

Mettre à jour elgg.org

— Cloner https://github.com/Elgg/www.elgg.org
— Modifiez la version d'Elgg requise dans `composer.json`
— Mettre à jours vendors

```
composer update
```

— Committez et poussez les modifications (push)
— Faites un pull vers le site en activité

```
cd /var/www/www.elgg.org && sudo su deploy && git pull
```

— Mettez à jour les dépendances

```
composer install --no-dev --prefer-dist --optimize-autoloader
```

— **Allez sur la panneau admin de la communauté**
  — Vider le cache APC
  — Exécuter la mise à niveau

Faites l'annonce

Ce devrait être la toute dernière chose à faire.

1. Ouvrez `https://github.com/Elgg/Elgg/blob/<tag>/CHANGELOG.md` et copiez le contenu pour cette version
2. Connectez-vous sur https://elgg.org/blog et rédigez un nouvel article de blog avec un sommaire
3. Copiez le contenu dans le CHANGELOG, supprimez le formattage, et supprimez manuellement les balises SVG
4. Ajoutez les tags `release` et `elgg2.x` où x est le nom de la branche en train d'être publiée
5. Tweetez depuis le compte Twitter account d'Elgg

3.7 Appendix

Miscellaneous information about the project.

3.7.1 Upgrade Notes

If you are upgrading your plugins and website to a new major Elgg releases, the following noteworthy changes apply.

See the administrator guides for *how to upgrade a live site*.

From 2.x to 3.0

Contents

PHP 7.0 is now required

5.6 is reaching it's end of life. PHP 7.0 is now required to install and run Elgg.

`$CONFIG` is removed!

Not exactly, however you **must** audit its usage and *should* replace it with `elgg_get_config()` and `elgg_set_config()`, as recommended since Elgg 1.9.

The global `$CONFIG` is now a proxy for Elgg's configuration container, and modifications **will fail** if you try to alter array properties directly. E.g. `$CONFIG->cool_fruit[] = 'Pear';`. The silver lining is that failures will emit NOTICEs.

Removed views

- `forms/account/settings` : usersettings extension can now extend the view `forms/usersettings/save`
- `forms/admin/site/advanced/system`
- `resources/file/download`
- `output/checkboxes` : use `output/tags` if you want the same behaviour
- `input/write_access` : mod/pages now uses the **access :collections :write** plugin hook.
- `invitefriends/form`
- `page/layouts/content` : use `page/layouts/default`
- `page/layouts/one_column` : use `page/layouts/default`
- `page/layouts/one_sidebar` : use `page/layouts/default`
- `page/layouts/two_sidebar` : use `page/layouts/default`
- `page/layouts/walled_garden` : use `page/layouts/default`
- `page/layouts/walled_garden/cancel_button`
- `page/layouts/two_column_left_sidebar`
- `page/layouts/widgets/add_panel`
- `page/elements/topbar_wrapper` : update your use of `page/elements/topbar` to include a check for a logged in user
- `pages/icon`
- `groups/group_sort_menu` : use `register, filter:menu:groups/all` plugin hook
- `groups/my_status`
- `groups/profile/stats`
- `subscriptions/form/additions` : extend `notifications/settings/other` instead
- `likes/count` : modifications can now be done to the `likes_count` menu item
- `likes/css` : likes now uses `elgg/likes.css`
- `resources/members/index`
- `messageboard/css`
- `notifications/subscriptions/personal`
- `notifications/subscriptions/collections`
- `notifications/subscriptions/form`
- `notifications/subscriptions/jsfuncs`
- `notifications/subscriptions/forminternals`
- `notifications/css`
- `pages/input/parent`
- `river/item` : use elgg_view_river_item() to render river items
- `river/user/default/profileupdate`
- `admin.js`
- `aalborg_theme/homepage.png`
- `aalborg_theme/css`
- `resources/avatar/view` : Use entity icon API
- `ajax_loader.gif`
- `button_background.gif`
- `button_graduation.png`
- `elgg_toolbar_logo.gif`
- `header_shadow.png`
- `powered_by_elgg_badge_drk_bckgnd.gif`
- `powered_by_elgg_badge_light_bckgnd.gif`
- `sidebar_background.gif`
- `spacer.gif`
- `toptoolbar_background.gif`
- `two_sidebar_background.gif`
- `ajax_loader_bw.gif` : use `graphics/ajax_loader_bw.gif`
- `elgg_logo.png` : use `graphics/elgg_logo.png`

— `favicon-128.png` : use `graphics/favicon-128.png`
— `favicon-16.png` : use `graphics/favicon-16.png`
— `favicon-32.png` : use `graphics/favicon-32.png`
— `favicon-64.png` : use `graphics/favicon-64.png`
— `favicon.ico` : use `graphics/favicon.ico`
— `favicon.svg` : use `graphics/favicon.svg`
— `friendspicker.png` : use `graphics/friendspicker.png`
— `walled_garden.jpg` : use `graphics/walled_garden.jpg`
— `core/friends/collection`
— `core/friends/collections`
— `core/friends/collectiontabs`
— `core/friends/tablelist`
— `core/friends/talbelistcountupdate`
— `lightbox/elgg-colorbox-theme/colorbox-images/*`
— `navigation/menu/page` : now uses `navigation/menu/default` and a prepare hook
— `navigation/menu/site` : now uses default view
— `page/elements/by_line` : Use `object/elements/imprint`
— `forms/admin/site/advanced/security` : the site secret information has been moved to `forms/admin/security/settings`
— `river/object/file/create` : check *River*
— `river/object/page/create` : check *River*
— `river/object/page_top/create` : check *River*
— `river/relationship/member` : check *River*
— `object/page_top` : use `object/page`
— `ajax/discussion/reply/edit` : See *Discussion replies moved to comments*
— `discussion/replies` : See *Discussion replies moved to comments*
— `object/discussion_reply` : See *Discussion replies moved to comments*
— `resources/discussion/reply/edit` : See *Discussion replies moved to comments*
— `resources/elements/discussion_replies` : See *Discussion replies moved to comments*
— `river/elements/discussion_replies` : See *Discussion replies moved to comments*
— `river/object/discussion/create`
— `river/object/discussion_reply/create` : See *Discussion replies moved to comments*
— `search/object/discussion_reply/entity` : See *Discussion replies moved to comments*
— `rss/discussion/replies` : See *Discussion replies moved to comments*
— `search/header`
— `search/layout` in both `default` and `rss` viewtypes
— `search/no_results`
— `search/object/comment/entity`
— `search/css` : Moved to `search/search.css`
— `search/startblurb`
— `bookmarks/bookmarklet.gif`
— `blog_get_page_content_list`
— `blog_get_page_content_archive`
— `blog_get_page_content_edit`
— `forms/invitefriends/invite` : use `forms/friends/invite`
— `resources/invitefriends/invite` : use `resources/friends/invite`
— `resources/reportedcontent/add`
— `resources/reportedcontent/add_form`
— `resources/site_notifications/view` : Use `resources/site_notifications/owner`
— `resources/site_notifications/everyone` : Use `resources/site_notifications/all`

Removed functions/methods

All the functions in `engine/lib/deprecated-1.9.php` were removed. See https://github.com/Elgg/Elgg/blob/2.0/engine/lib/deprecated-1.9.php for these functions. Each `@deprecated` declaration includes instructions on what to use instead. All the functions in `engine/lib/deprecated-1.10.php` were removed. See https://github.com/Elgg/Elgg/blob/2.0/engine/lib/deprecated-1.10.php for these functions. Each `@deprecated` declaration includes instructions on what to use instead.

- `elgg_register_library` : require your library files so they are available globally to other plugins
- `elgg_load_library`
- `activity_profile_menu`
- `can_write_to_container` : Use `ElggEntity->canWriteToContainer()`
- `create_metadata_from_array`
- `metadata_array_to_values`
- `datalist_get`
- `datalist_set`
- `detect_extender_valuetype`
- `developers_setup_menu`
- `elgg_disable_metadata`
- `elgg_enable_metadata`
- `elgg_get_class_loader`
- `elgg_get_metastring_id`
- `elgg_get_metastring_map`
- `elgg_register_class`
- `elgg_register_classes`
- `elgg_register_viewtype`
- `elgg_is_registered_viewtype`
- `file_delete` : Use `ElggFile->deleteIcon()`
- `file_get_type_cloud`
- `file_type_cloud_get_url`
- `get_default_filestore`
- `get_site_entity_as_row`
- `get_group_entity_as_row`
- `get_missing_language_keys`
- `get_object_entity_as_row`
- `get_user_entity_as_row`
- `update_river_access_by_object`
- `garbagecollector_orphaned_metastrings`
- `groups_access_collection_override`
- `groups_get_group_tool_options` : Use `elgg()->group_tools->all()`
- `groups_join_group` : Use `ElggGroup::join`
- `groups_prepare_profile_buttons` : Use `register, menu:title` hook
- `groups_register_profile_buttons` : Use `register, menu:title` hook
- `groups_setup_sidebar_menus`
- `groups_set_icon_url`
- `groups_setup_sidebar_menus`
- `messages_notification_msg`
- `set_default_filestore`
- `generate_user_password` : Use `ElggUser::setPassword`
- `row_to_elggrelationship`
- `run_function_once` : Use `Elgg\Upgrade\Batch` interface
- `system_messages`
- `notifications_plugin_pagesetup`
- `elgg_format_url` : Use elgg_format_element() or the « output/text » view for HTML escaping.
- `get_site_by_url`

— `elgg_override_permissions` : No longer used as handler for `permissions_check` and `container_permissions_check` hooks
— `elgg_check_access_overrides`
— `AttributeLoader` became obsolete and was removed
— `Application::loadSettings`
— `ElggEntity::addToSite`
— `ElggEntity::disableMetadata`
— `ElggEntity::enableMetadata`
— `ElggEntity::getSites`
— `ElggEntity::removeFromSite`
— `ElggEntity::isFullyLoaded`
— `ElggEntity::clearAllFiles`
— `ElggPlugin::getFriendlyName` : Use `ElggPlugin::getDisplayName()`
— `ElggPlugin::setID`
— `ElggPlugin::unsetAllUsersSettings`
— `ElggFile::setFilestore` : ElggFile objects can no longer use custom filestores.
— `ElggFile::size` : Use `getSize`
— `ElggDiskFilestore::makeFileMatrix` : Use `Elgg\EntityDirLocator`
— `ElggData::get` : Usually can be replaced by property read
— `ElggData::getClassName` : Use `get_class()`
— `ElggData::set` : Usually can be replaced by property write
— `ElggEntity::setURL` : See `getURL` for details on the plugin hook
— `ElggMenuBuilder::compareByWeight` : Use `compareByPriority`
— `ElggMenuItem::getWeight` : Use `getPriority`
— `ElggMenuItem::getContent` : Use `elgg_view_menu_item()`
— `ElggMenuItem::setWeight` : Use `setPriority`
— `ElggRiverItem::getPostedTime` : Use `getTimePosted`
— `ElggSession` has removed all deprecated methods
— `ElggSite::addEntity`
— `ElggSite::addObject`
— `ElggSite::addUser`
— `ElggSite::getEntities` : Use `elgg_get_entities()`
— `ElggSite::getExportableValues` : Use `toObject`
— `ElggSite::getMembers` : Use `elgg_get_entities()`
— `ElggSite::getObjects` : Use `elgg_get_entities()`
— `ElggSite::listMembers` : Use `elgg_list_entities()`
— `ElggSite::removeEntity`
— `ElggSite::removeObject`
— `ElggSite::removeUser`
— `ElggSite::isPublicPage` : Logic moved to the router and should not be accessed directly
— `ElggSite::checkWalledGarden` : Logic moved to the router and should not be accessed directly
— `ElggUser::countObjects` : Use `elgg_get_entities()`
— `Logger::getClassName` : Use `get_class()`
— `Elgg\Application\Database::getTablePrefix` : Read the `prefix` property
— `elgg_view_access_collections()`
— `ElggSession::get_ignore_access` : Use `getIgnoreAccess`
— `ElggSession::set_ignore_access` : Use `setIgnoreAccess`
— `profile_pagesetup`
— `pages_can_delete_page` : Use `$entity->canDelete()`
— `pages_search_pages`
— `pages_is_page` : use `$entity instanceof ElggPage`
— `discussion_comment_override` : See *Discussion replies moved to comments*
— `discussion_can_edit_reply` : See *Discussion replies moved to comments*
— `discussion_reply_menu_setup` : See *Discussion replies moved to comments*

— `discussion_reply_container_logic_override` : See *Discussion replies moved to comments*
— `discussion_reply_container_permissions_override` : See *Discussion replies moved to comments*
— `discussion_update_reply_access_ids` : See *Discussion replies moved to comments*
— `discussion_search_discussion` : See *Discussion replies moved to comments*
— `discussion_add_to_river_menu` : See *Discussion replies moved to comments*
— `discussion_prepare_reply_notification` : See *Discussion replies moved to comments*
— `discussion_redirect_to_reply` : See *Discussion replies moved to comments*
— `discussion_ecml_views_hook` : See *Discussion replies moved to comments*
— `search_get_where_sql`
— `search_get_ft_min_max`
— `search_get_order_by_sql`
— `search_consolidate_substrings`
— `search_remove_ignored_words`
— `search_get_highlighted_relevant_substrings`
— `search_highlight_words`
— `search_get_search_view`
— `search_custom_types_tags_hook`
— `search_tags_hook`
— `search_users_hook`
— `search_groups_hook`
— `search_objects_hook`
— `members_list_popular`
— `members_list_newest`
— `members_list_online`
— `members_list_alpha`
— `members_nav_popular`
— `members_nav_newest`
— `members_nav_online`
— `members_nav_alpha`
— `uservalidationbyemail_generate_code`

All functions around entity subtypes table :

— `add_subtype` : Use `elgg_set_entity_class` at runtime
— `update_subtype` : Use `elgg_set_entity_class` at runtime
— `remove_subtype`
— `get_subtype_id`
— `get_subtype_from_id`
— `get_subtype_class` : Use `elgg_get_entity_class`
— `get_subtype_class_from_id`

All caches have been consolidated into a single API layer. The following functions and methods have been removed :

— `is_memcache_available`
— `_elgg_get_memcache`
— `_elgg_invalidate_memcache_for_entity`
— `ElggMemcache`
— `ElggFileCache`
— `ElggStaticVariableCache`
— `ElggSharedMemoryCache`
— `Elgg\Cache\Pool` interface and all extending classes

As a result of system log changes :

— `system_log_default_logger` : moved to `system_log` plugin
— `system_log_listener` : moved to `system_log` plugin
— `system_log` : moved to `system_log` plugin

- `get_system_log` : renamed to `system_log_get_log` and moved to `system_log` plugin
- `get_log_entry` : renamed to `system_log_get_log_entry` and moved to `system_log` plugin
- `get_object_from_log_entry` : renamed to `system_log_get_object_from_log_entry` and moved to `system_log` plugin
- `archive_log` : renamed to `system_log_archive_log` and moved to `system_log` plugin
- `logbrowser_user_hover_menu` : renamed to `system_log_user_hover_menu` and moved to `system_log` plugin
- `logrotate_archive_cron` : renamed to `system_log_archive_cron` and moved to `system_log` plugin
- `logrotate_delete_cron` : renamed to `system_log_delete_cron` and moved to `system_log` plugin
- `logrotate_get_seconds_in_period` : renamed to `system_log_get_seconds_in_period` and moved to `system_log` plugin
- `log_browser_delete_log` : renamed to `system_log_browser_delete_log` and moved to `system_log` plugin

Deprecated APIs

- `ban_user` : Use `ElggUser->ban()`
- `create_metadata` : Use `ElggEntity` setter or `ElggEntity->setMetadata()`
- `update_metadata` : Use `ElggMetadata->save()`
- `get_metadata_url`
- `create_annotation` : Use `ElggEntity->annotate()`
- `update_metadata` : Use `ElggAnnotation->save()`
- `elgg_get_user_validation_status` : Use `ElggUser->isValidated()`
- `make_user_admin` : Use `ElggUser->makeAdmin()`
- `remove_user_admin` : Use `ElggUser->removeAdmin()`
- `unban_user` : Use `ElggUser->unban()`
- `elgg_get_entities_from_attributes` : Use `elgg_get_entities()`
- `elgg_get_entities_from_metadata` : Use `elgg_get_entities()`
- `elgg_get_entities_from_relationship` : Use `elgg_get_entities()`
- `elgg_get_entities_from_private_settings` : Use `elgg_get_entities()`
- `elgg_get_entities_from_access_id` : Use `elgg_get_entities()`
- `elgg_list_entities_from_metadata` : Use `elgg_list_entities()`
- `elgg_list_entities_from_relationship` : Use `elgg_list_entities()`
- `elgg_list_entities_from_private_settings` : Use `elgg_list_entities()`
- `elgg_list_entities_from_access_id` : Use `elgg_list_entities()`
- `elgg_list_registered_entities` : Use `elgg_list_entities()`
- `elgg_batch_delete_callback`
- `\Elgg\Project\Paths::sanitize` : Use `\Elgg\Project\Paths::sanitize()`
- `elgg_group_gatekeeper` : Use `elgg_entity_gatekeeper()`
- `get_entity_dates` : Use `elgg_get_entity_dates()`
- `messages_set_url` : Use `ElggEntity::getURL()`

Removed global vars

— `$CURRENT_SYSTEM_VIEWTYPE`
— `$DEFAULT_FILE_STORE`
— `$ENTITY_CACHE`
— `$SESSION` : Use the API provided by `elgg_get_session()`
— `$CONFIG->site_id` : Use `1`
— `$CONFIG->search_info`
— `$CONFIG->input` : Use `set_input` and `get_input`

Removed classes/interfaces

— `FilePluginFile` : replace with `ElggFile` (or load with `get_entity()`)
— `Elgg_Notifications_Notification`
— `Elgg\Database\EntityTable\UserFetchResultException.php`
— `Elgg\Database\MetastringsTable`
— `Elgg\Database\SubtypeTable`
— `Exportable` and its methods `export` and `getExportableValues` : Use `toObject`
— `ExportException`
— `Importable` and its method `import`.
— `ImportException`
— `ODD` and all classes beginning with `ODD*`.
— `XmlElement`
— `Elgg_Notifications_Event` : Use `\Elgg\Notifications\Event`
— `Elgg\Mail\Address` : use `Elgg\Email\Address`
— `ElggDiscussionReply` : user `ElggComment` see *Discussion replies moved to comments*

Schema changes

The storage engine for the database tables has been changed from MyISAM to InnoDB. You maybe need to optimize your database settings for this change. The `datalists` table has been removed. All settings from datalists table have been merged into the `config` table.

Metastrings in the database have been denormalized for performance purposes. We removed the metastrings table and put all the string values directly in the metadata and annotation tables. You need to update your custom queries to reflect these changes. Also the `msv` and `msn` table aliases are no longer available. It is best practice not to rely on the table aliases used in core queries. If you need to use custom clauses you should do your own joins.

From the « users_entity » table, the `password` and `hash` columns have been removed.

The `geocode_cache` table has been removed as it was no longer used.

`subtype` column in `entities` table no longer holds a subtype ID, but a subtype string `entity_subtypes` table has been dropped.

`type`, `subtype` and `access_id` columns in `river` table have been dropped. For queries without `elgg_get_river()` join the `entities` table on `object_guid` to check the type and the subtype of the entity. Access column hasn't been in use for some time : queries are built to ensure access to all three entities (subject, object and target).

Changes in `elgg_get_entities`, `elgg_get_metadata` and `elgg_get_annotations` getter functions

`elgg_get_entities` now accepts all options that were previously distributed between `elgg_get_entities_from_metadata`, `elgg_get_entities_from_annotations`, `elgg_get_entities_from_relationship`, `elgg_get_entities_from_private_settings` and `elgg_get_entities_from_access_id`. The latter have been been deprecated.

Passing raw MySQL statements to options is deprecated. Plugins are advised to use closures that receive an instance of `\Elgg\Database\QueryBuilder` and prepare the statement using database abstraction layer. On one hand this will ensure that all statements are properly sanitized using the database driver, on the other hand it will allow us to transition to testable object-oriented query building.

`wheres` statements should not use raw SQL strings, instead pass an instance of `\Elgg\Database\Clauses\WhereClause` or a closure that returns an instance of `\Doctrine\DBAL\Query\Expression\CompositeExpression`:

```
elgg_get_entities([
   'wheres' => [
        function(\Elgg\Database\QueryBuilder $qb, $alias) {
           $joined_alias = $qb->joinMetadataTable($alias, 'guid', 'status');
           return $qb->compare("$joined_alias.name", 'in', ['draft', 'unsaved_draft'],
↪ ELGG_VALUE_STRING);
        }
   ]
]);
```

`joins`, `order_by`, `group_by`, `selects` clauses should not use raw SQL strings. Use closures that receive an instance of `\Elgg\Database\QueryBuilder` and return a prepared statement.

The `reverse_order_by` option has been removed.

Plugins should not rely on joined and selected table aliases. Closures passed to the options array will receive a second argument that corresponds to the selected table alias. Plugins must perform their own joins and use joined aliases accordingly.

Note that all of the private API around building raw SQL strings has also been removed. If you were relying on them in your plugins, be advised that anything marked as `@access private` or `@internal` in core can be modified and removed at any time, and we do not guarantee any backward compatibility for those functions. DO NOT USE THEM. If you find yourself needing to use them, open an issue on Github and we will consider adding a public equivalent.

Boolean entity properties

Storage of metadata, annotation and private setting values has been aligned.

Boolean values are cast to integers when saved : `false` is stored as `0` and `true` is stored as `1`. This has breaking implications for private settings, which were previously stored as empty strings for `false` values. Plugins should write their own migration scripts to alter DB values from empty strings to `0` (for private settings that are expected to store boolean values) to ensure that `elgg_get_entities()` can retrieve these values with `private_setting_name_value_pairs` containing `false` values. This applies to plugin settings, as well as any private settings added to entities.

Metadata Changes

Metadata is no longer access controlled. If your plugin created metadata with restricted access, those restrictions will not be honored. You should use annotations or entities instead, which do provide access control.

Do not read or write to the `access_id` property on ElggMetadata objects.

Metadata is no longer enabled or disabled. You can no longer perform the `enable` and `disable` API calls on metadata.

Metadata no longer has an `owner_guid`. It is no longer possible to query metadata based on `owner_guids`. Subsequently, `ElggMetadata::canEdit()` will always return `true` regardless of the logged in user, unless explicitly overriden by a plugin hook.

Permissions and Access

User capabilities service will no longer trigger permission check hooks when :
- permissions are checked for an admin user
- permissions are checked when access is ignored with `elgg_set_ignore_access()`

This means that plugins can no longer alter permissions in aforementioned cases.

`elgg_check_access_overrides()` has been removed, as plugins will no longer need to validate access overrides.

The translations for the default Elgg access levels have new translation language keys.

Multi Site Changes

Pre 3.0 Elgg has some (partial) support for having multiple sites in the same database. This Multi Site concept has been completely removed in 3.0. Entities no longer have the site_guid attribute. This means there is no longer the ability to have entities on different sites. If you currently have multiple sites in your database, upgrading Elgg to 3.0 will fail. You need to separate the different sites into separate databases/tables.

Related to the removal of the Multi Site concept in Elgg, there is no longer a need for entities having a "member_of_site" relationship with the Site Entity. All functions related to adding/removing this relationship has been removed. All existing relationships will be removed as part of this upgrade.

Setting `ElggSite::$url` has no effect. Reading the site URL always pulls from the `$CONFIG->wwwroot` set in settings.php, or computed by Symphony Request.

`ElggSite::save()` will fail if it isn't the main site.

Entity Subtable Changes

The subtable `sites_entity` for `ElggSite` no longer exists. All attributes have been moved to metadata. The subtable `groups_entity` for `ElggGroup` no longer exists. All attributes have been moved to metadata. The subtable `objects_entity` for `ElggObject` no longer exists. All attributes have been moved to metadata. The subtable `users_entity` for `ElggUser` no longer exists. All attributes have been moved to metadata.

If you have custom queries referencing this table you need to update them. If you have function that rely on `Entity->getOriginalAttributes()` be advised that this will only return the base attributes of an `ElggEntity` and no longer contain the secondary attributes.

Friends and Group Access Collection

The access collections table now has a subtype column. This extra data helps identifying the purpose of the ACL. The user owned access collections are assumed to be used as Friends Collections and now have the "friends_collection" subtype. The groups access collection information was previously stored in the group_acl metadata. With the introduction of the ACL subtype this information has been moved to the ACL subtype attribute.

The `ACCESS_FRIENDS` access_id has been migrated to an actual access collection (with the subtype `friends`). All entities and annotations have been updated to use the new access collection id. The access collection is created when a user is created. When a relationship of the type `friends` is created, the related guid will also be added to the access collection. You can no longer save or update entities with the access id `ACCESS_FRIENDS`.

Subtypes no longer have an ID

Entity subtypes have been denormalized. `entity_subtypes` table has been removed and `subtype` column in entities table simply holds the string representation of the subtype.

Consequently, all API around adding/updating entity subtypes and classes have been removed.

Plugins can now use `elgg_set_entity_class()` and `elgg_get_entity_class()` to register a custom entity class at runtime (e.g. in system init handler).

All entities now **MUST** have a subtype. By default, the following subtypes are added and reserved :
- `user` for users
- `group` for groups
- `site` for sites

Custom class loading

Elgg no longer provides API functions to register custom classes. If you need custom classes you can use `PSR-0` classes in the `/classes` folder of your plugin or use composer for autoloading of additional classes.

The following class registration related functions have been removed :
- `elgg_get_class_loader`
- `elgg_register_class`
- `elgg_register_classes`

Dependency Injection Container

Plugins can now define their services and attach them to Elgg's public DI container by providing definitions in `elgg-services.php` in the root of the plugin directory.

`elgg()` no longer returns an instance of Elgg application, but a DI container instance.

Search changes

We have added a search service into core, consequently the `search` plugin now only provides a user interface for displaying forms and listing search results. Many of the views in the search plugin have been affected by this change.

The FULLTEXT indices have been removed on various tables. The search plugin will now always use a like query when performing a search.

See *Search Service* and *Search hooks* documentation for detailed information about new search capabilities.

Form and field related changes

- `input/password` : by default this field will no longer show a value passed to it, this can be overridden by passing the view var `always_empty` and set it to false
- `input/submit`, `input/reset` and `input/button` are now rendered with a `<button>` instead of the `<input>` tag. These input view also accept `text` and `icon` parameters.
- `output/url` now sets `.elgg-anchor` class on anchor elements and accepts `icon` parameter. If no `text` is set, the `href` parameter used as a label will be restricted to 100 characters.
- `output/url` now supports a `badge` parameter, which can be used where a counter, a badge, or similar is required as a postfix (mainly in menu items that have counters).
- `output/tags` no longer uses `<ul>` tags with floats and instead it relies on inherently inline elements such as `<span>` and `<a>`

Entity and River Menu Changes

The Entity and River menu now shows all the items in a dropdown. Social actions like liking or commenting are moved to an alternate menu called the social menu, which is meant for social actions.

Removed libraries

`elgg_register_library` and `elgg_load_library` have been removed. These functions had little impact on performance (especially with OPCache enabled), and made it difficult for other plugins to work with APIs contained in libraries. Additionally it was difficult for developers to know that APIs were contained in a library while there being autocompleted by IDE.

If you are concerned with performance, move the logic to classes and let PHP autoload them as necessary, otherwise use `require_once` and require your libraries.

Removed pagehandling

- `file/download`
- `file/search`
- `groupicon`
- `twitterservice`
- `collections/pickercallback`
- `discussion/reply` : See *Discussion replies moved to comments*
- `expages`
- `invitefriends` : Use `friends/{username}/invite`
- `messages/compose` : Use `messages/add`
- `reportedcontent`

Removed actions

- `file/download` : Use `elgg_get_inline_url` or `elgg_get_download_url`
- `file/delete` : Use `entity/delete` action
- `import/opendd`
- `discussion/reply/save` : See *Discussion replies moved to comments*
- `discussion/reply/delete` : See *Discussion replies moved to comments*
- `comment/delete` : Use `entity/delete` action
- `uservalidationbyemail/bulk_action` : use `admin/user/bulk/validate` or `admin/user/bulk/delete`
- `uservalidationbyemail/delete` : use `admin/user/bulk/delete`
- `uservalidationbyemail/validate` : use `admin/user/bulk/validate`
- `invitefriends/invite` : use `friends/invite`

Inheritance changes

- `ElggData` (and hence most Elgg domain objects) no longer implements `Exportable`
- `ElggEntity` no longer implements `Importable`
- `ElggGroup` no longer implements `Friendable`
- `ElggRelationship` no longer implements `Importable`
- `ElggSession` no longer implements `ArrayAccess`
- `Elgg\Application\Database` no longer extends `Elgg\Database`

Removed JavaScript APIs

- `admin.js`
- `elgg.widgets` : Use the `elgg/widgets` module. The « widgets » layouts do this module automatically
- `lightbox.js` : Use the `elgg/lightbox` module as needed
- `lightbox/settings.js` : Use the `getOptions, ui.lightbox` JS hook or the `data-colorbox-opts` attribute
- `elgg.ui.popupClose` : Use the `elgg/popup` module
- `elgg.ui.popupOpen` : Use the `elgg/popup` module
- `elgg.ui.initAccessInputs`
- `elgg.ui.river`
- `elgg.ui.initDatePicker` : Use the `input/date` module
- `elgg.ui.likesPopupHandler`
- `elgg.embed` : Use the `elgg/embed` module
- `elgg.discussion` : Use the `elgg/discussion` module
- `embed/custom_insert_js` : Use the `embed, editor` JS hook
- `elgg/ckeditor.js` : replaced by `elgg-ckeditor.js`
- `elgg/ckeditor/set-basepath.js`
- `elgg/ckeditor/insert.js`
- `jQuery.cookie` : Use `elgg.session.cookie`
- `jquery.jeditable`
- `likes.js` : The `elgg/likes` module is loaded automatically
- `messageboard.js`
- `elgg.autocomplete` is no longer defined.
- `elgg.messageboard` is no longer defined.
- `jQuery.fn.friendsPicker`
- `elgg.ui.toggleMenu` is no longer defined
- `elgg.ui.toggleMenuItems` : Use `data-toggle` attribute when registering toggleable menu items
- `uservalidationbyemail/js.php` : Use the `elgg/uservalidationbyemail` module

— `discussion.js` : See *Discussion replies moved to comments*

Removed hooks/events

— Event **login, user** : Use **login :before** or **login :after**. Note the user is not logged in during the **login :before** event
— Event **delete, annotations** : Use **delete, annotation**
— Event **pagesetup, system** : Use the menu or page shell hooks instead
— Event **upgrade, upgrade** : Use **upgrade, system** instead
— Hook **index, system** : Override the `resources/index` view
— Hook **object :notifications, <type>** : Use the hook **send :before, notifications**
— Hook **output :before, layout** : Use **view_vars, page/layout/<layout_name>**
— Hook **output :after, layout** : Use **view, page/layout/<layout_name>**
— Hook **email, system** : Use more granular **<hook>, system :email** hooks
— Hook **email :message, system** : Use **zend :message, system :email** hook
— Hook **members :list, <page>** : Use your own pagehandler or route hook
— Hook **members :config, <page>** : Use **register, menu :filter :members**
— Hook **profile_buttons, group** : Use **register, menu :title**

Removed forms/actions

— `notificationsettings/save` form and action
— `notificationsettings/groupsave` form and action
— `discussion/reply/save` form and action

APIs that now accept only an `$options` array

— `ElggEntity::getAnnotations`
— `ElggEntity::getEntitiesFromRelationship`
— `ElggGroup::getMembers`
— `ElggUser::getGroups`
— `ElggUser::getFriends` (as part of `Friendable`)
— `ElggUser::getFriendsOf` (as part of `Friendable`)
— `ElggUser::getFriendsObjects` (as part of `Friendable`)
— `ElggUser::getObjects` (as part of `Friendable`)
— `find_active_users`
— `elgg_get_admin_notices`

Plugin functions that now require an explicit `$plugin_id`

— `elgg_get_all_plugin_user_settings`
— `elgg_set_plugin_user_setting`
— `elgg_unset_plugin_user_setting`
— `elgg_get_plugin_user_setting`
— `elgg_set_plugin_setting`
— `elgg_get_plugin_setting`
— `elgg_unset_plugin_setting`
— `elgg_unset_all_plugin_settings`

Class constructors that now accept only a `stdClass` object or `null`

- `ElggAnnotation` : No longer accepts an annotation ID
- `ElggGroup` : No longer accepts a GUID
- `ElggMetadata` : No longer accepts a metadata ID
- `ElggObject` : No longer accepts a GUID
- `ElggRelationship` : No longer accepts a relationship ID or `null`
- `ElggSite` : No longer accepts a GUID or URL
- `ElggUser` : No longer accepts a GUID or username
- `ElggPlugin` : No longer accepts a GUID or path. Use `ElggPlugin::fromId` to construct a plugin from its path

Miscellaneous API changes

- `ElggBatch` : You may only access public properties
- `ElggEntity` : The `tables_split` and `tables_loaded` properties were removed
- `ElggEntity` : Empty URLs will no longer be normalized. This means entities without URLs will no longer result in the site URL
- `ElggGroup::removeObjectFromGroup` requires passing in an `ElggObject` (no longer accepts a GUID)
- `ElggUser::$salt` no longer exists as an attribute, nor is it used for authentication
- `ElggUser::$password` no longer exists as an attribute, nor is it used for authentication
- `elgg_get_widget_types` no longer supports `$exact` as the 2nd argument
- `elgg_instanceof` no longer supports the fourth `class` argument
- `elgg_view` : The 3rd and 4th (unused) arguments have been removed. If you use the `$viewtype` argument, you must update your usage.
- `elgg_view_icon` no longer supports `true` as the 2nd argument
- `elgg_list_entities` no longer supports the option `view_type_toggle`
- `elgg_list_registered_entities` no longer supports the option `view_type_toggle`
- `elgg_log` no longer accepts the level `"DEBUG"`
- `elgg_dump` no longer accepts a `$to_screen` argument.
- `elgg_gatekeeper` and `elgg_admin_gatekeeper` no longer report `login` or `admin` as forward reason, but `403`
- `Application::getDb()` no longer returns an instance of `Elgg\Database`, but rather a `Elgg\Application\Database`
- `$CONFIG` is no longer available as a local variable inside plugin `start.php` files.
- `elgg_get_config('siteemail')` is no longer available. Use `elgg_get_site_entity()->email`.
- `ElggEntity::saveIconFromUploadedFile` only saves *master* size, the other sizes are created when requested by `ElggEntity::getIcon()` based on the *master* size
- `ElggEntity::saveIconFromLocalFile` only saves *master* size, the other sizes are created when requested by `ElggEntity::getIcon()` based on the *master* size
- `ElggEntity::saveIconFromElggFile` only saves *master* size, the other sizes are created when requested by `ElggEntity::getIcon()` based on the *master* size
- Group entities do no longer have the magic `username` attribute.
- Pagehandling will no longer detect `group:<guid>` in the URL
- The CRON interval `reboot` is removed.
- The URL endpoints `js/` and `css/` are no longer supported. Use `elgg_get_simplecache_url()`.
- The generic comment save action no longer sends the notification directly, this has been offloaded to the notification system.
- The script `engine/start.php` is removed.
- The functions `set_config`, `unset_config` and `get_config` have been deprecated and replaced by `elgg_set_config`, `elgg_remove_config` and `elgg_get_config`.

— Config values `path`, `wwwroot`, and `dataroot` are not read from the database. The settings.php file values are always used.
— Config functions like `elgg_get_config` no longer trim keys.
— If you override the view `navigation/menu/user_hover/placeholder`, you must change the config key `lazy_hover:menus` to `elgg_lazy_hover_menus`.
— The config value `entity_types` is no longer present or used.
— Uploaded images are autorotated based on their orientation metadata.
— The view `object/widget/edit/num_display` now uses an `input/number` field instead of `input/select`; you might need to update your widget edit views accordingly.
— Annotation names are no longer trimmed during save

View extension behaviour changed

An extended view now will receive all the regular hooks (like the *view_vars* hook). It now is also possible to extend view extensions. With this change in behaviour all view rendering will behave the same. It no longer matters if it was used as an extension or not.

JavaScript hook calling order may change

When registering for hooks, the `all` keyword for wildcard matching no longer has any effect on the order that handlers are called. To ensure your handler is called last, you must give it the highest priority of all matching handlers, or to ensure your handler is called first, you must give it the lowest priority of all matching handlers.

If handlers were registered with the same priority, these are called in the order they were registered.

To emulate prior behavior, Elgg core handlers registered with the `all` keyword have been raised in priority. Some of these handlers will most likely be called in a different order.

Widget layout related changes

The widget layout usage has been changed. Content is no longer drawn as part of the layout. You need to wrap you content in another layout and use the widgets layout as part of your content. If you want some special content to show if there are no widgets in the layout, you can now pass a special `no_widgets` parameter (as String or as a Closure).

When registering widgets you can no longer omit passing a context as the `all` context is no longer supported. You need to explicitely pass the contexts for which the widget is intended.

Routing

Page handling using `elgg_register_page_handler()` has been deprecated.

We have added new routing API using `elgg_register_route()`, which allows plugins to define named routes, subsequently using route names to generate URLs using `elgg_generate_url()`.

See *routing* docs for details.

As a result of this change all core page handlers have been removed, and any logic contained within these page handlers has been moved to respective resource views.

`elgg_generate_entity_url()` has been added as shortcut way to generate URLs from named routes that depend on entity type and subtype.

Use of `handler` parameter in entity menus has been deprecated in favour of named entity routes.

Gatekeeper function have been refactored to serve as middleware in the routing process, and as such they no longer return values. These functions throw HTTP exceptions that are then routed to error pages and can be redirected to other pages via hooks.

Labelling

Entity and collection labelling conventions have changed to comply with the new routing patterns :

```
return [
   'item:object:blog' => 'Blog',
   'collection:object:blog' => 'Blogs',
   'collection:object:blog:all' => 'All site blogs',
   'collection:object:blog:owner' => '%s\'s blogs',
   'collection:object:blog:group' => 'Group blogs',
   'collection:object:blog:friends' => 'Friends\' blogs',
   'add:object:blog' => 'Add blog post',
   'edit:object:blog' => 'Edit blog post',
];
```

These conventions are used across the routing and navigation systems, so plugins are advised to follow them.

Request value filtering

`set_input()` and `get_input()` no longer trim values.

Action responses

All core and core plugin actions now all use the new Http Response functions like *elgg_ok_response* and *elgg_error_response* instead of *forward()*. The effect of this change is that is most cases the *"forward", "system"* hook is no longer triggered. If you like to influence the responses you now can use the *"response", "action :<name/of/action>"* hook. This gives you more control over the response and allows to target a specific action very easily.

HtmLawed is no longer a plugin

— Do not call `elgg_load_library('htmlawed')`.
— In the hook params for `'config', 'htmlawed'`, the `hook_tag` function name changed.

New approach to page layouts

`one_column`, `one_sidebar`, `two_sidebar` and `content` layouts have been removed - instead layout rendering has been centralized in the `default`. Updated `default` layout provides full control over the layout elements via `$vars`. For maximum backwards compatibility, calls to `elgg_view_layout()` with these layout names will still yield expected output, but the plugins should start using the `default` layout with an updated set of parameters.

Page layouts have been decomposed into smaller elements, which should make it easier for themes to target specific layout elements without having to override layouts at large.

As a result of these changes :
— all layouts are consistent in how they handle title and filter menus, breadcrumbs and layout subviews

- all layouts can now be easily extended to have multiple tabs. Plugins can pass `filter_id` parameter that will allow other plugins to hook into `register, menu:filter:<filter_id>` hook and add new tabs. If no `filter_id` is provided, default `register, menu:filter` hook can be used.
- layout views and subviews now receive `identifier` and `segments` of the page being rendered
- layout parameters are available to title and filter menu hooks, which allows resources to provide additional context information, for example, an `$entity` in case of a profile resource

Plugins and themes should :

- Update calls to `elgg_view_layout()` to use `default` layout
- Update replace `nav` parameter in layout views with `breadcrumbs` parameter
- Update their use of `filter` parameter in layout views by either providing a default set of filter tabs, or setting a `filter_id` parameter and using hooks
- Remove `page/layouts/one_column` view
- Remove `page/layouts/one_sidebar` view
- Remove `page/layouts/two_sidebar` view
- Remove `page/layouts/content` view
- Update their use of `page/layouts/default`
- Update their use of `page/layouts/error`
- Update their use of `page/layouts/elements/filter`
- Update their use of `page/layouts/elements/header`
- Update their use of `page/layouts/elements/footer`
- Update their use of `page/elements/title`
- Update their use of `navigation/breadcrumbs` to pass `$vars['breadcrumbs']` to `elgg_get_breadcrumbs()`
- Update hook registrations for `output:before, layout` to `view_vars, page/layout/<layout_name>`
- Update hook registrations for `output:after, layout` to `view, page/layout/<layout_name>`

Likes plugin

Likes no longer uses Elgg's toggle API, so only a single `likes` menu item is used. The add/remove actions no longer return Ajax values directly, as likes status data is now returned with *every* Ajax request that sends a « guid ». When the number of likes is zero, the `likes_count` menu item is now hidden by adding *.hidden* to the LI element, instead of the anchor. Also the `likes_count` menu item is a regular link, and is no longer created by the `likes/count` view.

Notifications plugin

Notifications plugin has been rewritten dropping many views and actions. The purpose of this rewrite was to implement a more efficient, extendable and scalable interface for managing notifications preferences. We have implemented a much simpler markup and removed excessive styling and javascript that was required to make the old interface work.

If your plugin is extending any of the views or relies on any actions in the notifications plugin, it has to be updated.

Pages plugin

The suptype `page_top` has been migrated into the subtype `page`. The subtype `page` has it's own class namely `ElggPage`. In order to check if an `ElggPage` is a top page the class function `ElggPage->isTopPage()` was added.

All pages have a metadata value for `parent_guid`, for top pages this contains `0`.

Profile plugin

All profile related functionality has been moved out of core into this plugin. Most noteable are the profile field admin utility and the hook to set up the profile fields config data.

Data Views plugin

The Data Views plugin no longer comes bundled.

Twitter API plugin

The `twitter_api` plugin has been removed from the Elgg core. The plugin is still available as the Composer package elgg/twitter_api, in order to install it add the following to you `composer.json` `require` section :

```
"elgg/twitter_api": "~1.9"
```

Legacy URLs plugin

The `legacy_urls` plugin has been removed from the Elgg core. The plugin is still available as the Composer package elgg/legacy_urls, in order to install it add the following to you `composer.json` `require` section :

```
"elgg/legacy_urls": "~2.3"
```

User validation by email plugin

The listing view of unvalidated users has been moved from the plugin to Elgg core. Some generic action (eg. validate and delete) have also been moved to Elgg core.

Email delivery

To provide for more granularity in email handling and delivery, **email, system** hook has been removed. New email service provides for several other replacement hooks that allow plugins to control email content, format, and transport used for delivery.

`elgg_set_email_transport()` can now be used to replace the default Sendmail transport with another instance of `\Zend\Mail\Transport\TransportInterface`, e.g. SMTP, in-memory, or file transport. Note that this function must be called early in the boot process. Note that if you call this function on each request, using plugin settings to determine transport config may not be very efficient - store these settings in as datalist or site config values, so they are loaded from boot cache.

Theme and styling changes

Aalborg theme is no longer bundled with Elgg. Default core theme is now based on Aalboard, but it has undergone major changes.

Notable changes in plugins :

- Topbar, navbar and header have been combined into a single responsive topbar component
- Default inner width is now 1280px (80rem * 16px/1rem)
- Preferred unit of measurement is now *rem* and not *px*
- The theme uses *8-point grid system <https ://builttoadapt.io/intro-to-the-8-point-grid-system-d2573cde8632>*
- Menus, layout elements and other components now use flexbox
- Reset is done using *8-point grid system <https ://necolas.github.io/normalize.css/>*
- Media queries have been rewritten for mobile-first experience
- Form elements (text inputs, buttons and selects) now have an equal height of 2.5rem
- Layout header is now positioned outside of the layout columns, which have been wrapped into `elgg-layout-columns`
- z-index properties have been reviewed and stacked with simple iteration instead of *9999999 <https ://hackernoon.com/my-approach-to-using-z-index-eca67feb079c>*.
- Color scheme has been changed to highlight actionable elements and reduce abundance of gray shades
- search plugin no longer extends `page/elements/header` and instead `page/elements/topbar` renders `search/search_box` view
- `.elgg-icon` no longer has a global `font-size`, `line-height` or `color` : these values will be inherited from parent items
- Support for `.elgg-icon-hover` has been dropped
- User « hover » icons are no longer covered with a « caret » icon

Read more about *Theming Principles*

Also note, CSS views served via `/cache` URLs are pre-processed using *CSS Crush <http ://the-echoplex.net/csscrush/>*. If you make references to CSS variables or other elements, the definition must be located within the same view output. E.g. A variable defined in `elgg.css` cannot be referenced in a separate CSS file like `colorbox.css`.

Comments

Submitting comments is now AJAXed. After a succesful submission the comment list will be updated automatically.

The following changes have been made to the comment notifications.

- The language keys related to comment notifications have changed. Check the `generic_comment:notification:owner:` language keys
- The action for creating a comment (`action/comment/save`) was changed. If your plugin overruled this action you should have a look at it in order to prevent double notifications

Object listing views

- `object/elements/full/body` now wraps the full listing body in a `.elgg-listing-full-body` wrapper
- `object/elements/full` now supports `attachments` and `responses` which are rendered after listing body
- In core plugins, resource views no longer render comments/replies - instead they pass a `show_responses` flag to the entity view, which renders the responses and passes them to the full listing view. Third party plugins will need to update their uses of `object/<subtype>` and `resources/<handler>/view` views.
- Full discussion view is now rendered using `object/elements/full` view
- `object/file` now passes image (specialcontent) view as an `attachment` to the full listing view

Menu changes

Default sorting of menu items has been changed from `text` to `priority`.

Note that `register` and `prepare` hooks now use collections API. For the most part, all hooks should continue working, as long as they are not performing complex operations with arrays.

Support for `icon` and `badge` parameters was added. Plugins should start using these parameters and prefer them to a single `text` parameter. CSS should be used to control visibility of the label, icon and badge, instead of conditionals in preparing menu items.

All menus are now wrapped with `nav.elgg-menu-container` to ensure that multiple menu sections have a single parent element, and can be styled using flexbox or floats.

All menu items are now identified with with `data-menu-item` attribute, sections - with `data-menu-section`, containers with - `data-menu-name` attributes.

`topbar` menu :

- `account` menu item with priority `800` added to `alt` section
- `site_notifications` menu item is now a child of `account` with priority `100`
- `usersettings` menu item is now a child of `account` with priority `300`
- `administration` menu item is now a child of `account` with priority `800`
- `logout` menu item is now a child of `account` with priority `900`
- `dashboard` menu item now is now a child of `acount` has priority of `100`
- In `default` section (`profile`, `friends`, `messages`), core menu items now use `icon` parameter and use CSS to hide the label. Plugins that register items to this section and expect a visible label need to update their CSS.
- `profile` menu item is now a child of `account`
- `friends` menu item is now a child of `account`

`entity` menu :

- `access` menu item has been removed. Access information is now rendered in the entity byline.

`user_hover` menu :

- All items use the `icon` parameter.
- The layout of the dropdown has been changed. If you have modified the look and feel of this dropdown, you might need to update your HTML/CSS.

`widget` menu :

- `collapse` menu item has been removed and CSS updated accordingly

`title` menu :

The `profile` plugin no longer uses the actions section of the `user_hover` menu, but registers regulare `title` menu items.

`extras` menu :

This menu has been removed from the page layout. Menu items that registered for this menu have been moved to other menus.

`groups:my_status` menu :

This menu has been removed from the group profile page.

`site_notifications` menu :

This menu has been removed. Site Notification objects now use the entity menu for actions.

`site` menu :

Registration of custom menu item defined in admin interface has been moved to `register, menu:site` hook. `navigation/menu/site` view has been removed. Site menu now adds a `more` menu item directly to the ``default` section.

Entity icons

Default icon image files have been moved and re-mapped as follows :

- Default icons : `views/default/icon/default/$size.png`
- User icons : `views/default/icon/user/default/$size.gif`
- Group icons : `views/default/icon/group/default/$size.gif` in the groups plugin

Groups icon files have been moved from `groups/<guid><size>.jpg` relative to group owner's directory on filestore to a location prescribed by the entity icon service. Plugins should stop accessing files on the filestore directly and use the entity icon API. Upgrade script is available via admin interface.

The generation of entity icons has ben changed. No longer will all the configured sizes be generated when calling one of the entity icon functions (`ElggEntity::saveIconFromUploadedFile`, `ElggEntity::saveIconFromLocalFile` or `ElggEntity::saveIconFromElggFile`), but only the *master* size. The other configured sizes will be generated when requesting that size based of the *master* icon.

Icon glyphs

FontAwesome has been upgraded to version 5.0+. There were certain changes to how FontAwesome glyphs are rendered. The core will take care of most changes (e.g. mapping old icon names to new ones, and using the correct prefix for brand and solid icons).

Autocomplete (user and friends pickers)

Friends Picker input is now rendered using `input/userpicker`.

Plugins should :

- Update overriden `input/userpicker` to support new `only_friends` parameter
- Remove friends picker CSS from their stylesheets

Friends collections

Friends collections UI has been moved to its own plugins - `friends_collections`.

Layout of `.elgg-body` elements

In 3.0, these elements by default no longer stretch to fill available space in a block context. They still clear floats and allow breaking words to wrap text.

Core modules and layouts that relied on space-filling have been reworked for Flexbox and we encourage devs to do the same, rather than use the problematic `overflow: hidden`.

Delete river items

The function `elgg_delete_river()` which was deprecated in 2.3, has been reinstated. Notable changes between the internals of this function are;

- It accepts all `$options` from `elgg_get_river()` but requires at least one of the following params to be set id(s), annotation_id(s), subject_guid(s), object_guid(s), target_guid(s) or view(s)
- Since `elgg_get_river` by default has a limit on the number of river items it fetches, if you wish to remove all river items you need to set `limit` to `false`
- Access is ignored when deleting river items
- Events are fired just before and after a river item has been deleted

Discussion replies moved to comments

Since discussion replies where mostly a carbon copy of comments, all discussion replies have been migrated to comments. All related action, hooks, event, language keys etc. have been removed.

Note : Discussion comments will now show up in the Comments section of Search, no longer under the Discussion section.

Translations cleanup

All plugins have been scanned for unused translation keys. The unused keys have been removed. If there was a generic translation available for the custom translation key, these have also been updated.

System Log

System log API has been moved out of core into a `system_log` plugin. `logbrowser` and `logrotate` plugins have been merged into the `system_log` plugin.

Error logging

Sending `elgg_log()` and PHP error messages to page output is now only possible via the developers plugin « Log to the screen » setting. See the `settings.example.php` file for more information on using `$CONFIG->debug` in your `settings.php` file. Debugging should generally be done via the `xdebug` extension or `tail -f /path/to/error.log` on your server.

Composer asset plugin no longer required

Assets are now loaded from https://asset-packagist.org. FXP composer asset plugin is no longer required when installing Elgg or updating composer dependencies.

Cron logs

The cron logs are no longer stored in the database, but on the filesystem (in dataroot). This will allow longer output to be stored. A migration script was added to migrate the old database settings to the new location and remove the old values from the database.

Removed / changed language keys

— The language keys related to comment notifications have changed. Check the `generic_comment:notification:owner:` language keys

New MySQL schema features are not applied

New 3.0 installations require MySQL 5.5.3 (or higher) and use the utf8mb4 character set and LONGTEXT content columns (notably allowing storing longer content and extended characters like emoji).

Miscellaneous changes

The settings « Allow visitors to register » and « Restrict pages to logged-in users » now appear on the Basic Settings admin page.

Twitter API plugin

The `twitter_api` plugin no longer comes bundled with Elgg.

Unit and Integration Testing

Elgg's PHPUnit bootstrap can now handle both unit and integration tests. Please note that **you shouldn't run tests on a production site**, as it may damage data integrity. To prevent data loss, you need to specify database settings via environment variables. You can do so via the phpunit.xml bootstrap.

Plugins can now implement their own PHPUnit tests by extending `\Elgg\UnitTestCase` and `\Elgg\IntegrationTestCase` classes. `plugins` test suite will automatically autoload PHPUnit tests from `mod/<plugin_id>/tests/phpunit/unit` and `mod/<plugin_id>/tests/phpunit/integration`.

Prior to running integration tests, you need to enable the plugins that you wish to test alongside core API.

`\Elgg\IntegrationTestCase` uses `\Elgg\Seeding` trait, which can be used to conveniently build new entities and write them to the database.

`\Elgg\UnitTestCase` does not use the database, but provides a database mocking interface, which allows tests to define query specs with predefined returns.

By default, both unit and integration tests will be run whenever `phpunit` is called. You can use `--testsuite` flag to only run a specific suite : `phpunit --testsuite unit` or `phpunit --testsuite integration` or `phpunit --testsuite plugins`.

For integration testing to run properly, plugins are advised to not put any logic into the root of `start.php`, and instead return a Closure. This allows the testsuite to build a new Application instance without loosing plugin initialization logic.

Plugins with simpletests will continue working as perviously. However, method signatures in the `ElggCoreUnitTest` abstract class have changed and you will need to update your tests accordingly. Namely, it's discouraged to use `__construct` and `__desctruct` methods. `setUp` and `tearDown` have been marked as private and are used for consistent test boostrapping and asserting pre and post conditions, your test case should use `up` and `down` methods instead.

Simpletests can no longer be executed from the admin interface of the developers plugin. Use Elgg cli command : `elgg-cli simpletest`

From 2.2 to 2.3

Contents

PHP Version

PHP 5.5 has reached end of life in July 2016. To ensure that Elgg sites are secure, we now require PHP 5.6 for new installations.

Existing installations can continue using PHP 5.5 until Elgg 3.0.

In order to upgrade Elgg to 2.3 using composer while using PHP 5.5, you may need to use `--ignore-platform-reqs` flag.

Deprecated APIs

- Registering for `to:object` hook by the extender name : Use `to:object, annotation` and `to:object, metadata` hooks instead.
- `ajax_forward_hook()` : No longer used as handler for *"forward","all"* hook. Ajax response is now wrapped by the `ResponseFactory`
- `ajax_action_hook()` : No longer used as handler for *"action","all"* hook. Output buffering now starts before the hook is triggered in `ActionsService`
- `elgg_error_page_handler()` : No longer used as a handler for *"forward",<error_code>* hooks
- `get_uploaded_file()` : Use new file uploads API instead

- `get_user_notification_settings()` : Use `ElggUser::getNotificationSettings()`
- `set_user_notification_setting()` : Use `ElggUser::setNotificationSetting()`
- `pagesetup, system` event : Use the menu or page shell hooks instead.
- `elgg.walled_garden` JavaScript is deprecated : Use `elgg/walled_garden` AMD module instead.
- `elgg()->getDb()->getTableprefix()` : Use `elgg_get_config('dbprefix')`.
- Private `update_entity_last_action()` : Refrain from manually updating last action timestamp.
- Setting non-public `access_id` on metadata is deprecated. See below.
- `get_resized_image_from_existing_file()` : Use `elgg_save_resized_image()`.
- `get_resized_image_from_uploaded_file()` : Use `elgg_save_resized_image()` in combination with upload API.
- `get_image_resize_parameters()` will be removed.
- `elgg_view_input()` : Use `elgg_view_field()`. Apologies for the API churn.

Deprecated Views

- `resources/file/world` : Use the `resources/file/all` view instead.
- `resources/pages/world` : Use the `resources/pages/all` view instead.
- `walled_garden.js` : Use the `elgg/walled_garden` module instead.

New API for page and action handling

Page handlers and action script files should now return an instance of `\Elgg\Http\ResponseBuilder`. Plugins should use the following convenience functions to build responses :

- `elgg_ok_response()` sends a 2xx response with HTML (page handler) or JSON data (actions)
- `elgg_error_response()` sends a 4xx or 5xx response without content/data
- `elgg_redirect_response()` silently redirects the request

New API for working with file uploads

- `elgg_get_uploaded_files()` - returns an array of Symfony uploaded file objects
- `ElggFile::acceptUploadedFile()` - moves an uploaded file to Elgg's filestore

New API for manipulating images

New image manipulation service implements a more efficient approach to cropping and resizing images.

- `elgg_save_resized_image()` - crops and resizes an image to preferred dimensions

New API for events

- `elgg_clear_event_handlers()` - similar to `elgg_clear_plugin_hook_handlers` this functions removes all registered event handlers

New API for signing URLs

URLs can now be signed with a SHA-256 HMAC key and validated at any time before URL expiry. This feature can be used to tokenize action URLs in email notifications, as well as other uses outside of the Elgg installation.

- `elgg_http_get_signed_url()` - signs the URL with HMAC key
- `elgg_http_validate_signed_url()` - validates the signed URL
- `elgg_signed_request_gatekeeper()` - gatekeeper that validates the signature of the current request

Extendable form views

Form footer rendering can now be deferred until the form view and its extensions have finished rendering. This allows plugins to collaborate on form views without breaking the markup logic.

- `elgg_set_form_footer()` - sets form footer for deferred rendering
- `elgg_get_form_footer()` - returns currently set form footer

Metadata `access_id`

It's now deprecated to create metadata with an explicit `access_id` value other than `ACCESS_PUBLIC`.

In Elgg 3.0, metadata will not be access controlled, and will be available in all contexts. If your plugin relies on access control of metadata, it would be wise to migrate storage to annotations or entities instead.

New API for extracting class names from arrays

Similar to `elgg_extract()`, `elgg_extract_class()` extracts the « class » key (if present), merges into existing class names, and always returns an array.

Notifications

- A high level `'prepare','notification'` hook is now triggered for instant and subscription notifications and can be used to alter notification objects irrespective of their type.
- `'format','notification:<method>'` hook is now triggered for instant and subscription notifications and can be used to format the notification (e.g. strip HTML tags, wrap the notification body in a template etc).
- Instant notifications are now handled by the notifications service, hence almost all hooks applicable to subscription notifications also apply to instant notifications.
- `elgg_get_notification_methods()` can be used to obtain registered notification methods
- Added `ElggUser::getNotificationSettings()` and `ElggUser::setNotificationSetting()`

Entity list functions can output tables

In functions like `elgg_list_entities($options)`, table output is possible by setting `$options['list_type'] = 'table'` and providing an array of table columns as `$options['columns']`. Each column is an `Elgg\Views\TableColumn` object, usually created via methods on the service `elgg()->table_columns`.

Plugins can provide or alter these factory methods (see `Elgg\Views\TableColumn\ColumnFactory`). See the view `admin/users/newest` for a usage example.

Inline tabs components

Inline tabs component can now be rendered with `page/components/tabs` view. The components allows to switch between pre-poluated and ajax-loaded. See `page/components/tabs` in core views and `theme_sandbox/components/tabs` in developers plugin for usage instructions and examples.

API to alter registration and login URL

- `elgg_get_registration_url()` should be used to obtain site's registration URL
- `elgg_get_login_url()` should be used to obtain site's login URL
- `registration_url, site` hook can be used to alter the default registration URL
- `login_url, site` hook can be used to alter the default login URL

Support for fieldsets in forms

- `elgg_view_field()` replaces `elgg_view_input()`. It has a similar API, but accepts a single array.
- `elgg_view_field()` supports `#type`, `#label`, `#help` and `#class`, allowing unprefixed versions to be sent to the input view `$vars`.
- The new view `input/fieldset` can be used to render a set of fields, each rendered with `elgg_view_field()`.

Lightbox

- Lightbox css is no longer loaded as an external CSS file. Lightbox theme now extends `elgg.css` and `admin.css`
- Default lightbox config is now defined via `'elgg.data','site'` server-side hook

From 2.1 to 2.2

Contents

Deprecated APIs

- `elgg.ui.river` JavaScript library : Remove calls to `elgg_load_js('elgg.ui.river')` from plugin code. Update `core/river/filter` and `forms/comment/save`, if overwritten, to require component AMD modules
- `elgg.ui.popupOpen()` and `elgg.ui.popupClose()` methods in `elgg.ui` JS library : Use `elgg/popup` module instead.
- `lightbox.js` library : Do not use `elgg_load_js('lightbox.js');` unless your code references deprecated `elgg.ui.lightbox` namespace. Use `elgg/lightbox` AMD module instead.
- `elgg.embed` library and `elgg.embed` object : Do not use `elgg_load_js('elgg.embed')`. Use `elgg/embed` AMD module instead
- Accessing `icons_sizes` config value directly : Use `elgg_get_icon_sizes()`
- `can_write_to_container()` : Use `ElggEntity::canWriteToContainer()`

Deprecated Views

- `elgg/ui.river.js` is deprecated : Do not rely on simplecache URLs to work.
- `groups/js` is deprecated : Use `groups/navigation` AMD module as a menu item dependency for « feature » and « unfeature » menu items instead.
- `lightbox/settings.js` is deprecated : Use `getOptions, ui.lightbox` JS plugin hook or `data-colorbox-opts` attribute.
- `elgg/ckeditor/insert.js` is deprecated : You no longer need to include it, hook registration takes place in `elgg/ckeditor` module
- `embed/embed.js` is deprecated : Use `elgg/embed` AMD module.

Added `elgg/popup` module

New *elgg/popup module* can be used to build out more complex trigger-popup interactions, including binding custom anchor types and opening/closing popups programmatically.

Added `elgg/lightbox` module

New *elgg/lightbox module* can be used to open and close the lightbox programmatically.

Added `elgg/embed` module

Even though rarely necessary, `elgg/embed` AMD module can be used to access the embed methods programmatically. The module bootstraps itself when necessary and is unlikely to require further decoration.

New API for handling entity icons

- `ElggEntity` now implements `\Elgg\EntityIcon` interface
- `elgg_get_icon_sizes()` - return entity type/subtype specific icon sizes
- `ElggEntity::saveIconFromUploadedFile()` - creates icons from an uploaded file
- `ElggEntity::saveIconFromLocalFile()` - creates icons from a local file
- `ElggEntity::saveIconFromElggFile()` - creates icons from an instance of `ElggFile`
- `ElggEntity::getIcon()` - returns an instanceof `ElggIcon` that points to entity icon location on filestore (this may be just a placeholder, use `ElggEntity::hasIcon()` to validate if file has been written)
- `ElggEntity::deleteIcon()` - deletes entity icons

- `ElggEntity::getIconLastChange()` - return modified time of the icon file
- `ElggEntity::hasIcon()` - checks if an icon with given size has been created
- `elgg_get_embed_url()` - can be used to return an embed URL for an entity's icon (served via */serve-icon* handler)

User avatars are now served via `serve-file` handler. Plugins should start using `elgg_get_inline_url()` and note that :

- `/avatar/view` page handler and resource view have been deprecated
- `/mod/profile/icondirect.php` file has been deprecated
- `profile_set_icon_url()` is no longer registered as a callback for `"entity:icon:url","user"` plugin hook

Group avatars are now served via `serve-file` handler. Plugins should start using `elgg_get_inline_url()` and note that :

- `groupicon` page handler (`groups_icon_handler()`) has been deprecated
- `/mod/groups/icon.php` file has been deprecated

File entity thumbs and downloads are now served via `serve-file` handler. Plugins should start using `elgg_get_inline_url()` and `elgg_get_download_url()` and note that :

- `file/download` page handler and resource view have been deprecated
- `mod/file/thumbnail.php` file has been deprecated
- Several views have been updated to use new download URLs, including :
  - `mod/file/views/default/file/specialcontent/audio/default.php`
  - `mod/file/views/default/file/specialcontent/image/default.php`
  - `mod/file/views/default/resources/file/view.php`
  - `mod/file/views/rss/file/enclosure.php`

Removed APIs

Just a warning that the private entity cache functions (e.g. `_elgg_retrieve_cached_entity`) have been removed. Some plugins may have been using them. Plugins should not use private APIs as they will more often be removed without notice.

Improved `elgg/ckeditor` module

elgg/ckeditor module can now be used to add WYSIWYG to a textarea programmatically with `elgg/ckeditor#bind`.

From 2.0 to 2.1

Contents

Deprecated APIs

- `ElggFile::setFilestore`
- `get_default_filestore`
- `set_default_filestore`
- `elgg_get_config('siteemail')` : Use `elgg_get_site_entity()->email`
- URLs starting with `/css/` and `/js/` : Use `elgg_get_simplecache_url()`
- `elgg.ui.widgets` JavaScript object is deprecated by `elgg/widgets` AMD module

`Application::getDb()` changes

If you're using this low-level API, do not expect it to return an `Elgg\Database` instance in 3.0. It now returns an `Elgg\Application\Database` with many deprecated. These methods were never meant to be made public API, but we will do our best to support them in 2.x.

Added `elgg/widgets` module

If your plugin code calls `elgg.ui.widgets.init()`, instead use the *elgg/widgets module*.

From 1.x to 2.0

Contents

Elgg can be now installed as a composer dependency instead of at document root

That means an Elgg site can look something like this :

```
settings.php
vendor/
  elgg/
    elgg/
      engine/
        start.php
      _graphics/
        elgg_sprites.png
mod/
  blog
  bookmarks
  ...
```

`elgg_get_root_path` and `$CONFIG->path` will return the path to the application root directory, which is not necessarily the same as Elgg core's root directory (which in this case is `vendor/elgg/elgg/`).

Do not attempt to access the core Elgg from your plugin directly, since you cannot rely on its location on the filesystem.

In particular, don't try load `engine/start.php`.

```
// Don't do this!
dirname(__DIR__) . "/engine/start.php";
```

To boot Elgg manually, you can use the class `Elgg\Application`.

```
// boot Elgg in mod/myplugin/foo.php
require_once dirname(dirname(__DIR__)) . '/vendor/autoload.php';
\Elgg\Application::start();
```

However, use this approach sparingly. Prefer *Routing* instead whenever possible as that keeps your public URLs and your filesystem layout decoupled.

Also, don't try to access the `_graphics` files directly.

```
readfile(elgg_get_root_path() . "_graphics/elgg_sprites.png");
```

Use *Views* instead :

```
echo elgg_view('elgg_sprites.png');
```

Cacheable views must have a file extension in their names

This requirement makes it possibile for us to serve assets directly from disk for performance, instead of serving them through PHP.

It also makes it much easier to explore the available cached resources by navigating to dataroot/views_simplecache and browsing around.

— Bad : `my/cool/template`
— Good : `my/cool/template.html`

We now cache assets by `"$viewtype/$view"`, not `md5("$viewtype|$view")`, which can result in conflicts between cacheable views that don't have file extensions to disambiguate files from directories.

Dropped `jquery-migrate` and upgraded `jquery` to ^2.1.4

jQuery 2.x is API-compatible with 1.x, but drops support for IE8-, which Elgg hasn't supported for some time anyways.

See http://jquery.com/upgrade-guide/1.9/ for how to move off jquery-migrate.

If you'd prefer to just add it back, you can use this code in your plugin's init :

```
elgg_register_js('jquery-migrate', elgg_get_simplecache_url('jquery-migrate.js'),
↪'head');
elgg_load_js('jquery-migrate');
```

Also, define a `jquery-migrate.js` view containing the contents of the script.

JS and CSS views have been moved out of the js/ and css/ directories

They also have been given .js and .css extensions respectively if they didn't already have them :

Old view	New view
`js/view`	`view.js`
`js/other.js`	`other.js`
`css/view`	`view.css`
`css/other.css`	`other.css`
`js/img.png`	`img.png`

The main benefit this brings is being able to co-locate related assets. So a template (`view.php`) can have its CSS/JS dependencies right next to it (`view.css`, `view.js`).

Care has been taken to make this change as backwards-compatible as possible, so you should not need to update any view references right away. However, you are certainly encouraged to move your JS and CSS views to their new, canonical locations.

Practically speaking, this carries a few gotchas :

The `view_vars, $view_name` and `view, $view_name` hooks will operate on the *canonical* view name :

```
elgg_register_plugin_hook_handler('view', 'css/elgg', function($hook, $view_name) {
  assert($view_name == 'elgg.css') // not "css/elgg"
});
```

Using the `view, all` hook and checking for individual views may not work as intended :

```
elgg_register_plugin_hook_handler('view', 'all', function($hook, $view_name) {
  // Won't work because "css/elgg" was aliased to "elgg.css"
  if ($view_name == 'css/elgg') {
    // Never executed...
  }

  // Won't work because no canonical views start with css/* anymore
  if (strpos($view_name, 'css/') === 0) {
    // Never executed...
  }
});
```

Please let us know about any other BC issues this change causes. We'd like to fix as many as possible to make the transition smooth.

`fxp/composer-asset-plugin` is now required to install Elgg from source

We use `fxp/composer-asset-plugin` to manage our browser assets (js, css, html) with Composer, but it must be installed globally *before installing Elgg* in order for the `bower-asset/*` packages to be recognized. To install it, run :

```
composer global require fxp/composer-asset-plugin
```

If you don't do this before running `composer install` or `composer create-project`, you will get an error message :

```
[InvalidArgumentException]
Package fxp/composer-asset-plugin not found
```

List of deprecated views and view arguments that have been removed

We dropped support for and/or removed the following views :

- canvas/layouts/*
- categories
- categories/view
- core/settings/tools
- embed/addcontentjs
- footer/analytics (Use page/elements/foot instead)
- groups/left_column
- groups/right_column
- groups/search/finishblurb
- groups/search/startblurb
- input/calendar (Use input/date instead)
- input/datepicker (Use input/date instead)
- input/pulldown (Use input/select instead)
- invitefriends/formitems
- js/admin (Use AMD and `elgg_require_js` instead of extending JS views)

— js/initialise_elgg (Use AMD and `elgg_require_js` instead of extending JS views)
— members/nav
— metatags (Use the "head", "page" plugin hook instead)
— navigation/topbar_tools
— navigation/viewtype
— notifications/subscriptions/groupsform
— object/groupforumtopic
— output/calendar (Use output/date instead)
— output/confirmlink (Use output/url instead)
— page_elements/contentwrapper
— page/elements/shortcut_icon (Use the "head", "page" plugin hook instead)
— page/elements/wrapper
— profile/icon (Use `elgg_get_entity_icon`)
— river/object/groupforumtopic/create
— settings/{plugin}/edit (Use plugins/{plugin}/settings instead)
— user/search/finishblurb
— user/search/startblurb
— usersettings/{plugin}/edit (Use plugins/{plugin}/usersettings instead)
— widgets/{handler}/view (Use widgets/{handler}/content instead)

We also dropped the following arguments to views :

— « value » in output/iframe (Use « src » instead)
— « area2 » and « area3 » in page/elements/sidebar (Use « sidebar » or view extension instead)
— « js » in icon views (e.g. icon/user/default)
— « options » to input/radio and input/checkboxes which aren't key-value pairs will no longer be acceptable.

All scripts moved to bottom of page

You should test your plugin **with the JavaScript error console visible**. For performance reasons, Elgg no longer supports `script` elements in the `head` element or in HTML views. `elgg_register_js` will now load *all* scripts at the end of the `body` element.

You must convert inline scripts to *AMD* or to external scripts loaded with `elgg_load_js`.

Early in the page, Elgg provides a shim of the RequireJS `require()` function that simply queues code until the AMD `elgg` and `jQuery` modules are defined. This provides a straightforward way to convert many inline scripts to use `require()`.

Inline code which will fail because the stack is not yet loaded :

```
<script>
$(function () {
    // code using $ and elgg
});
</script>
```

This should work in Elgg 2.0 :

```
<script>
require(['elgg', 'jquery'], function (elgg, $) {
    $(function () {
        // code using $ and elgg
    });
});
</script>
```

Attribute formatter removes keys with underscores

`elgg_format_attributes()` (and all APIs that use it) now filter out attributes whose name contains an underscore. If the attribute begins with `data-`, however, it will not be removed.

Breadcrumbs

Breadcrumb display now removes the last item if it does not contain a link. To restore the previous behavior, replace the plugin hook handler `elgg_prepare_breadcrumbs` with your own :

```
elgg_unregister_plugin_hook_handler('prepare', 'breadcrumbs', 'elgg_prepare_
↪breadcrumbs');
elgg_register_plugin_hook_handler('prepare', 'breadcrumbs', 'myplugin_prepare_
↪breadcrumbs');

function myplugin_prepare_breadcrumbs($hook, $type, $breadcrumbs, $params) {
    // just apply excerpt to titles
    foreach (array_keys($breadcrumbs) as $i) {
        $breadcrumbs[$i]['title'] = elgg_get_excerpt($breadcrumbs[$i]['title'], 100);
    }
    return $breadcrumbs;
}
```

Callbacks in Queries

Make sure to use only valid *callable* values for « callback » argument/options in the API.

Querying functions will now will throw a `RuntimeException` if `is_callable()` returns `false` for the given callback value. This includes functions such as `elgg_get_entities()`, `get_data()`, and many more.

Comments plugin hook

Plugins can now return an empty string from `'comments',$entity_type` hook in order to override the default comments component view. To force the default comments component, your plugin must return `false`. If you were using empty strings to force the default comments view, you need to update your hook handlers to return `false`.

Container permissions hook

The behavior of the `container_permissions_check` hook has changed when an entity is being created : Before 2.0, the hook would be called twice if the entity's container was not the owner. On the first call, the entity's owner would be passed in as `$params['container']`, which could confuse handlers.

In 2.0, when an entity is created in a container like a group, if the owner is the same as the logged in user (almost always the case), this first check is bypassed. So the `container_permissions_check` hook will almost always be called once with `$params['container']` being the correct container of the entity.

Creating or deleting a relationship triggers only one event

The « create » and « delete » relationship events are now only fired once, with `"relationship"` as the object type.

E.g. Listening for the `"create", "member"` or `"delete", "member"` event(s) will no longer capture group membership additions/removals. Use the `"create", "relationship"` or `"delete", "relationship"` events.

Discussion feature has been pulled from groups into its own plugin

The `object, groupforumtopic` subtype has been replaced with the `object, discussion` subtype. If your plugin is using or altering the old discussion feature, you should upgrade it to use the new subtype.

Nothing changes from the group owners" point of view. The discussion feature is still available as a group tool and all old discussions are intact.

Dropped login-over-https feature

For the best security and performance, serve all pages over HTTPS by switching the scheme in your site's wwwroot to `https` at http://yoursite.tld/admin/settings/advanced

Elgg has migrated from ext/mysql to PDO MySQL

Elgg now uses a `PDO_MYSQL` connection and no longer uses any ext/mysql functions. If you use `mysql_*` functions, implicitly relying on an open connection, these will fail.

If your code uses one of the following functions, read below.

- `execute_delayed_write_query()`
- `execute_delayed_read_query()`

If you provide a callable `$handler` to be called with the results, your handler will now receive a `\Doctrine\DBAL\Driver\Statement` object. Formerly this was an ext/mysql `result` resource.

Event/Hook calling order may change

When registering for events/hooks, the `all` keyword for wildcard matching no longer has any effect on the order that handlers are called. To ensure your handler is called last, you must give it the highest priority of all matching handlers, or to ensure your handler is called first, you must give it the lowest priority of all matching handlers.

If handlers were registered with the same priority, these are called in the order they were registered.

To emulate prior behavior, Elgg core handlers registered with the `all` keyword have been raised in priority. Some of these handlers will most likely be called in a different order.

`export/` URLs are no longer available

Elgg no longer provides this endpoint for exposing resource data.

Icons migrated to Font Awesome

Elgg's sprites and most of the CSS classes beginning with `elgg-icon-` have been removed.

Usage of `elgg_view_icon()` is backward compatible, but static HTML using the `elgg-icon` classes will have to be updated to the new markup.

Increase of z-index value in elgg-menu-site class

The value of z-index in the elgg-menu-site class has been increased from 1 to 50 to allow for page elements in the content area to use the z-index property without the « More » site menu's dropdown being displayed behind these elements. If your plugin/theme overrides the elgg-menu-site class or views/default/elements/navigation.css please adjust the z-index value in your modified CSS file accordingly.

input/autocomplete view

Plugins that override the `input/autocomplete` view will need to include the source URL in the `data-source` attribute of the input element, require the new `elgg/autocomplete` AMD module, and call its `init` method. The 1.x javascript library `elgg.autocomplete` is no longer used.

Introduced third-party library for sending email

We are using the excellent `Zend\Mail` library to send emails in Elgg 2.0. There are likely edge cases that the library handles differently than Elgg 1.x. Take care to test your email notifications carefully when upgrading to 2.0.

Label elements

The following views received `label` elements around some of the input fields. If your plugin/theme overrides these views please check for the new content.

- views/default/core/river/filter.php
- views/default/forms/admin/plugins/filter.php
- views/default/forms/admin/plugins/sort.php
- views/default/forms/login.php

Plugin Aalborg Theme

The view `page/elements/navbar` now uses a Font Awesome icon for the mobile menu selector instead of an image. The `bars.png` image and supporting CSS for the 1.12 rendering has been removed, so update your theme accordingly.

Plugin Likes

Objects are no longer likable by default. To support liking, you can register a handler to permit the annotation, or more simply register for the hook `["likes:is_likable", "<type>:<subtype>"]` and return true. E.g.

```
elgg_register_plugin_hook_handler('likes:is_likable', 'object:mysubtype', 'Elgg\
↪Values::getTrue');
```

Just as before, the `permissions_check:annotate` hook is still called and may be used to override default behavior.

Plugin Messages

If you've removed or replaced the handler function `messages_notifier` to hide/alter the inbox icon, you'll instead need to do the same for the topbar menu handler `messages_register_topbar`. `messages_notifier` is no longer used to add the menu link.

Messages will no longer get the metadata "msg" for newly created messages. This means you can not rely on that metadata to exist.

Plugin Blog

The blog pages showing "Mine" or "Friends" listings of blogs have been changed to list all the blogs owned by the users (including those created in groups).

Plugin Bookmarks

The bookmark pages showing "Mine" or "Friends" listings of bookmarks have been changed to list all the bookmarks owned by the users (including those created in groups).

Plugin File

The file pages showing "Mine" or "Friends" listings of files have been changed to list all the files owned by the users (including those created in groups).

Removed Classes

- `ElggInspector`
- `Notable`
- `FilePluginFile` : replace with `ElggFile` (or load with `get_entity()`)

Removed keys available via `elgg_get_config()`

- `allowed_ajax_views`
- `dataroot_in_settings`
- `externals`
- `externals_map`
- `i18n_loaded_from_cache`
- `language_paths`
- `pagesetupdone`
- `registered_tag_metadata_names`
- `simplecache_enabled_in_settings`
- `translations`
- `viewpath`
- `views`
- `view_path`
- `viewtype`
- `wordblacklist`

Also note that plugins should not be accessing the global `$CONFIG` variable except for in `settings.php`.

Removed Functions

- `blog_get_page_content_friends`
- `blog_get_page_content_read`
- `count_unread_messages()`
- `delete_entities()`
- `delete_object_entity()`
- `delete_user_entity()`
- `elgg_get_view_location()`
- `elgg_validate_action_url()`
- `execute_delayed_query()`
- `extend_view()`
- `get_db_error()`
- `get_db_link()`
- `get_entities()`
- `get_entities_from_access_id()`
- `get_entities_from_access_collection()`
- `get_entities_from_annotations()`
- `get_entities_from_metadata()`
- `get_entities_from_metadata_multi()`
- `get_entities_from_relationship()`
- `get_filetype_cloud()`
- `get_library_files()`
- `get_views()`
- `is_ip_in_array()`
- `list_entities()`
- `list_entities_from_annotations()`
- `list_group_search()`
- `list_registered_entities()`
- `list_user_search()`
- `load_plugins()`
- `menu_item()`
- `make_register_object()`

— `mysql_*()` : Elgg *no longer uses ext/mysql*
— `remove_blacklist()`
— `search_for_group()`
— `search_for_object()`
— `search_for_site()`
— `search_for_user()`
— `search_list_objects_by_name()`
— `search_list_groups_by_name()`
— `search_list_users_by_name()`
— `set_template_handler()`
— `test_ip()`

Removed methods

— `ElggCache::set_variable()`
— `ElggCache::get_variable()`
— `ElggData::initialise_attributes()`
— `ElggData::getObjectOwnerGUID()`
— `ElggDiskFilestore::make_directory_root()`
— `ElggDiskFilestore::make_file_matrix()`
— `ElggDiskFilestore::user_file_matrix()`
— `ElggDiskFilestore::mb_str_split()`
— `ElggEntity::clearMetadata()`
— `ElggEntity::clearRelationships()`
— `ElggEntity::clearAnnotations()`
— `ElggEntity::getOwner()`
— `ElggEntity::setContainer()`
— `ElggEntity::getContainer()`
— `ElggEntity::getIcon()`
— `ElggEntity::setIcon()`
— `ElggExtender::getOwner()`
— `ElggFileCache::create_file()`
— `ElggObject::addToSite()` : parent function in ElggEntity still available
— `ElggObject::getSites()` : parent function in ElggEntity still available
— `ElggSite::getCollections()`
— `ElggUser::addToSite()` : parent function in ElggEntity still available
— `ElggUser::getCollections()`
— `ElggUser::getOwner()`
— `ElggUser::getSites()` : parent function in ElggEntity still available
— `ElggUser::listFriends()`
— `ElggUser::listGroups()`
— `ElggUser::removeFromSite()` : parent function in ElggEntity still available

The following arguments have also been dropped :
— `ElggSite::getMembers()` - 2 : `$offset`
— `elgg_view_entity_list()` - 3 : `$offset` - 4 : `$limit` - 5 : `$full_view` - 6 : `$list_type_toggle` - 7 : `$pagination`

Removed Plugin Hooks

— `[display, view]` : See the *new plugin hook*.

Removed Actions

— `widgets/upgrade`

Removed Views

— `forms/admin/plugins/change_state`

Removed View Variables

During rendering, the view system no longer injects these into the scope :
— `$vars['url']` : replace with `elgg_get_site_url()`
— `$vars['user']` : replace with `elgg_get_logged_in_user_entity()`
— `$vars['config']` : use `elgg_get_config()` and `elgg_set_config()`
— `$CONFIG` : use `elgg_get_config()` and `elgg_set_config()`
Also several workarounds for very old views are no longer performed. Make these changes :
— Set `$vars['full_view']` instead of `$vars['full']`.
— Set `$vars['name']` instead of `$vars['internalname']`.
— Set `$vars['id']` instead of `$vars['internalid']`.

Removed libraries

— `elgg:markdown` : Elgg no longer provides a markdown implementation. You must provide your own.

Specifying View via Properties

The metadata `$entity->view` no longer specifies the view used to render in `elgg_view_entity()`.

Similarly the property `$annotation->view` no longer has an effect within `elgg_view_annotation()`.

Viewtype is static after the initial `elgg_get_viewtype()` call

`elgg_set_viewtype()` must be used to set the viewtype at runtime. Although Elgg still checks the `view` input and `$CONFIG->view` initially, this is only done once per request.

Deprecations

It's deprecated to read or write to metadata keys starting with `filestore::` on `ElggFile` objects. In Elgg 3.0 this metadata will be deleted if it points to the current data root path, so few file objects will have it. Plugins should only use `ElggFile::setFilestore` if files need to be stored in a custom location.

Note : This is not the only deprecation in Elgg 2.0. Plugin developers should watch their site error logs.

From 1.10 to 1.11

Contents

Comment highlighting

If your theme is using the file `views/default/css/elements/components.php`, you must add the following style definitions in it to enable highlighting for comments and discussion replies :

```
.elgg-comments .elgg-state-highlight {
        -webkit-animation: comment-highlight 5s;
        animation: comment-highlight 5s;
}
@-webkit-keyframes comment-highlight {
        from {background: #dff2ff;}
        to {background: white;}
}
@keyframes comment-highlight {
        from {background: #dff2ff;}
        to {background: white;}
}
```

From 1.9 to 1.10

Contents

File uploads

If your plugin is using a snippet copied from the `file/upload` action to fix detected mime types for Microsoft zipped formats, it can now be safely removed.

If your upload action performs other manipulations on detected mime and simple types, it is recommended to make use of available plugin hooks :

- `'mime_type','file'` for filtering detected mime types
- `'simple_type','file'` for filtering parsed simple types

From 1.8 to 1.9

Contents

In the examples we are upgrading an imaginary « Photos » plugin.

Only the key changes are included. For example some of the deprecated functions are not mentioned here separately.

Each section will include information whether the change is backwards compatible with Elgg 1.8.

The manifest file

No changes are needed if your plugin is compatible with 1.8.

It's however recommended to add the `<id>` tag. It's value should be the name of the directory where the plugin is located inside the `mod/` directory.

If you make changes that break BC, you must update the plugin version and the required Elgg release.

Example of (shortened) old version :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
    <name>Photos</name>
    <author>John Doe</author>
    <version>1.0</version>
    <description>Adds possibility to upload photos and arrange them into albums.</
↪description>
    <requires>
        <type>elgg_release</type>
        <version>1.8</version>
    </requires>
</plugin_manifest>
```

Example of (shortened) new version :

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
    <name>Photos</name>
    <id>photos</id>
    <author>John Doe</author>
    <version>2.0</version>
    <description>Adds possibility to upload photos and arrange them into albums.</
↪description>
    <requires>
        <type>elgg_release</type>
        <version>1.9</version>
    </requires>
</plugin_manifest>
```

$CONFIG and $vars[“config”]

Both the global `$CONFIG` variable and the `$vars['config']` parameter have been deprecated. They should be replaced with the `elgg_get_config()` function.

Example of old code :

```
// Using the global $CONFIG variable:
global $CONFIG;
$plugins_path = $CONFIG->plugins_path

// Using the $vars view parameter:
$plugins_path = $vars['plugins_path'];
```

Example of new code :

```
$plugins_path = elgg_get_config('plugins_path');
```

Note : Compatible with 1.8

Note : See how the community_plugins plugin was updated : https://github.com/Elgg/community_plugins/commit/f233999bbd1478a200ee783679c2e2897c9a0483

Language files

In Elgg 1.8 the language files needed to use the `add_translation()` function. In 1.9 it is enough to just return the array that was previously passed to the function as a parameter. Elgg core will use the file name (e.g. en.php) to tell which language the file contains.

Example of the old way in `languages/en.php` :

```
$english = array(
    'photos:all' => 'All photos',
);
add_translation('en', $english);
```

Example of new way :

```
return array(
    'photos:all' => 'All photos',
);
```

Avertissement : Not compatible with 1.8

Notifications

One of the biggest changes in Elgg 1.9 is the notifications system. The new system allows more flexible and scalable way of sending notifications.

Example of the old way :

```
function photos_init() {
    // Tell core that we want to send notifications about new photos
    register_notification_object('object', 'photo', elgg_echo('photo:new'));

    // Register a handler that creates the notification message
    elgg_register_plugin_hook_handler('notify:entity:message', 'object', 'photos_
↪notify_message');
}

/**
 * Set the notification message body
 *
 * @param string $hook    Hook name
 * @param string $type    Hook type
 * @param string $message The current message body
 * @param array  $params  Parameters about the photo
 * @return string
 */
function photos_notify_message($hook, $type, $message, $params) {
    $entity = $params['entity'];
    $to_entity = $params['to_entity'];
    $method = $params['method'];
    if (elgg_instanceof($entity, 'object', 'photo')) {
        $descr = $entity->excerpt;
        $title = $entity->title;
        $owner = $entity->getOwnerEntity();
        return elgg_echo('photos:notification', array(
            $owner->name,
            $title,
            $descr,
            $entity->getURL()
        ));
    }
    return null;
}
```

Example of the new way :

```
function photos_init() {
    elgg_register_notification_event('object', 'photo', array('create'));
    elgg_register_plugin_hook_handler('prepare', 'notification:publish:object:photo',
↪'photos_prepare_notification');
```

(suite sur la page suivante)

(suite de la page précédente)

```
}

/**
 * Prepare a notification message about a new photo
 *
 * @param string                           $hook         Hook name
 * @param string                           $type         Hook type
 * @param Elgg_Notifications_Notification $notification The notification to prepare
 * @param array                            $params       Hook parameters
 * @return Elgg_Notifications_Notification
 */
function photos_prepare_notification($hook, $type, $notification, $params) {
    $entity = $params['event']->getObject();
    $owner = $params['event']->getActor();
    $recipient = $params['recipient'];
    $language = $params['language'];
    $method = $params['method'];

    // Title for the notification
    $notification->subject = elgg_echo('photos:notify:subject', array($entity->title),
↪ $language);

    // Message body for the notification
    $notification->body = elgg_echo('photos:notify:body', array(
        $owner->name,
        $entity->title,
        $entity->getExcerpt(),
        $entity->getURL()
    ), $language);

    // The summary text is used e.g. by the site_notifications plugin
    $notification->summary = elgg_echo('photos:notify:summary', array($entity->title),
↪ $language);

    return $notification;
}
```

Avertissement : Not compatible with 1.8

Note : See how the community_plugins plugin was updated to use the new system : https://github.com/Elgg/community_plugins/commit/bfa356cfe8fb99ebbca4109a1b8a1383b70ff123

Notifications can also be sent with the `notify_user()` function.

It has however been updated to support three new optional parameters passed inside an array as the fifth parameter.

The parameters give notification plugins more control over the notifications, so they should be included whenever possible. For example the bundled site_notifications plugin won't work properly if the parameters are missing.

Parameters :

— **object** The object that we are notifying about (e.g. ElggEntity or ElggAnnotation). This is needed so that notification plugins can provide a link to the object.

— **action** String that describes the action that triggered the notification (e.g. « create », « update », etc).

— **summary** String that contains a summary of the notification. (It should be more informative than the notification subject but less informative than the notification body.)

Example of the old way :

```
// Notify $owner that $user has added a $rating to an $entity created by him

$subject = elgg_echo('rating:notify:subject');
$body = elgg_echo('rating:notify:body', array(
        $owner->name,
        $user->name,
        $entity->title,
        $entity->getURL(),
));

notify_user($owner->guid,
                        $user->guid,
                        $subject,
                        $body
                );
```

Example of the new way :

```
// Notify $owner that $user has added a $rating to an $entity created by him

$subject = elgg_echo('rating:notify:subject');
$summary = elgg_echo('rating:notify:summary', array($entity->title));
$body = elgg_echo('rating:notify:body', array(
        $owner->name,
        $user->name,
        $entity->title,
        $entity->getURL(),
));

$params = array(
        'object' => $rating,
        'action' => 'create',
        'summary' => $summary,
);

notify_user($owner->guid,
                        $user->guid,
                        $subject,
                        $body,
                        $params
                );
```

Note : Compatible with 1.8

Adding items to the Activity listing

```
add_to_river('river/object/photo/create', 'create', $user_guid, $photo_guid);
```

```
elgg_create_river_item(array(
    'view' => 'river/object/photo/create',
    'action_type' => 'create',
    'subject_guid' => $user_guid,
    'object_guid' => $photo_guid,
));
```

You can also add the optional `target_guid` parameter which tells the target of the create action.

If the photo would had been added for example into a photo album, we could add it by passing in also :

```
'target_guid' => $album_guid,
```

Avertissement : Not compatible with 1.8

Entity URL handlers

The `elgg_register_entity_url_handler()` function has been deprecated. In 1.9 you should use the `'entity:url', 'object'` plugin hook instead.

Example of the old way :

```
/**
 * Initialize the photo plugin
 */
my_plugin_init() {
    elgg_register_entity_url_handler('object', 'photo', 'photo_url_handler');
}

/**
 * Returns the URL from a photo entity
 *
 * @param ElggEntity $entity
 * @return string
 */
function photo_url_handler($entity) {
    return "photo/view/{$entity->guid}";
}
```

Example of the new way :

```
/**
 * Initialize the photo plugin
 */
my_plugin_init() {
    elgg_register_plugin_hook_handler('entity:url', 'object', 'photo_url_handler');
}

/**
 * Returns the URL from a photo entity
```

(suite sur la page suivante)

(suite de la page précédente)

```
 *
 * @param string $hook   'entity:url'
 * @param string $type   'object'
 * @param string $url    The current URL
 * @param array  $params Hook parameters
 * @return string
 */
function photo_url_handler($hook, $type, $url, $params) {
    $entity = $params['entity'];

    // Check that the entity is a photo object
    if ($entity->getSubtype() !== 'photo') {
        // This is not a photo object, so there's no need to go further
        return;
    }

    return "photo/view/{$entity->guid}";
}
```

Avertissement : Not compatible with 1.8

Web services

In Elgg 1.8 the web services API was included in core and methods were exposed using `expose_function()`. To enable the same functionality for Elgg 1.9, enable the « Web services 1.9 » plugin and replace all calls to `expose_function()` with `elgg_ws_expose_function()`.

From 1.7 to 1.8

Contents

Elgg 1.8 is the biggest leap forward in the development of Elgg since version 1.0. As such, there is more work to update core and plugins than with previous upgrades. There were a small number of API changes and following our standard practice, the methods we deprecated have been updated to work with the new API. The biggest changes are in the standardization of plugins and in the views system.

Updating core

Delete the following core directories (same level as _graphics and engine) :

- _css
- account
- admin
- dashboard
- entities
- friends
- search
- settings
- simplecache
- views

Avertissement : If you do not delete these directories before an upgrade, you will have problems !

Updating plugins

Use standardized routing with page handlers

- All : /page_handler/all
- User's content : /page_handler/owner/ :username
- User's friends" content : /page_handler/friends/ :username
- Single entity : /page_handler/view/ :guid/ :title
- Added : /page_handler/add/ :container_guid
- Editing : /page_handler/edit/ :guid
- Group list : /page_handler/group/ :guid/all

Include page handler scripts from the page handler

Almost every page handler should have a page handler script. (Example : `bookmarks/all => mod/bookmarks/pages/bookmarks/all.php`)

- Call `set_input()` for entity guids in the page handler and use `get_input()` in the page handler scripts.
- Call `gatekeeper()` and `admin_gatekeeper()` in the page handler function if required.
- The group URL should use the `pages/:handler/owner.php` script.
- Page handlers should not contain HTML.
- Update the URLs throughout the plugin. (Don't forget to remove `/pg/` !)

Use standardized page handlers and scripts

- Store page handler scripts in `mod/:plugin/pages/:page_handler/:page_name.php`
- Use the content page layout in page handler scripts :

```
$content = elgg_view_layout('content', $options);
```

- Page handler scripts should not contain HTML.
- Call `elgg_push_breadcrumb()` in the page handler scripts.
- No need to set page owner if the URLs are in the standardized format.
- For group content, check the container_guid by using elgg_get_page_owner_entity().

The `object/:subtype` view

— Make sure there are views for `$vars['full_view'] == true` and `$vars['full_view'] == false`. `$vars['full_view']` replaced `$vars['full]`.
— Check for the object in `$vars['entity']`. Use `elgg_instance_of()` to make sure it's the type of entity you want.
— Return `true` to short circuit the view if the entity is missing or wrong.
— Use `elgg_view('object/elements/summary', array('entity' => $entity));` and `elgg_view_menu('entity', array('entity' => $entity));` to help format. You should use very little markup in these views.

Update action structure

— Namespace action files and action names (example : `mod/blog/actions/blog/save.php => action/blog/save`)
— Use the following action URLs :
  — Add : `action/:plugin/save`
  — Edit : `action/:plugin/save`
  — Delete : `action/:plugin/delete`
— Make the delete action accept `action/:handler/delete?guid=:guid` so the metadata entity menu has the correct URL by default.

Update deprecated functions

— Functions deprecated in 1.7 will produce visible errors in 1.8.
— You can also update functions deprecated in 1.8.
  — Many registration functions simply added an `elgg_` prefix for consistency, and should be easy to update.
  — See `/engine/lib/deprecated-1.8.php` for the full list.
  — You can set the debug level to "warning" to get visual reminders of deprecated functions.

Update the widget views

See the blog or file widgets for examples.

Update the group profile module

Use the blog or file plugins for examples. This will help with making your plugin themeable by the new CSS framework.

Update forms

— Move form bodies to the `forms/:action` view to use Evan's new `elgg_view_form`.
— Use input views in form bodies rather than html. This helps with theming and future-proofing.
— Add a function that prepares the form (see `mod/file/lib/file.php` for an example)
— Make your forms sticky (see the file plugin's upload action and form prepare function).

The forms API is discussed in more detail in *Forms + Actions*.

Clean up CSS/HTML

We have added many CSS patterns to the base CSS file (modules, image block, spacing primitives). We encourage you to use these patterns and classes wherever possible. Doing so should :

1. Reduce maintenance costs, since you can delete most custom CSS.
2. Make your plugin more compatible with community themes.

Look for patterns that can be moved into core if you need significant CSS.

We use hyphens rather than underscores in classes/ids and encourage you do the same for consistency.

If you do need your own CSS, you should use your own namespace, rather than `elgg-`.

Update manifest.xml

— Use http://el.gg/manifest17to18 to automate this.
— Don't use the « bundled » category with your plugins. That is only for plugins distributed with Elgg.

Update settings and user settings views

— The view for settings is now `plugins/:plugin/settings` (previously `settings/:plugin/edit`).
— The view for user settings is now `plugins/:plugin/usersettings` (previously `usersettings/:plugin/edit`).

3.7.2 FAQs and Other Troubleshooting

Below are some commonly asked questions about Elgg.

Contents

General

Voir aussi :

Trouver de l'aide

« Plugin cannot start and has been deactivated » or « This plugin is invalid »

This error is usually accompanied by more details explaining why the plugin is invalid. This is usually caused by an incorrectly installed plugin.

If you are installing a plugin called « test », there will be a test directory under mod. In that test directory there needs to be a manifest.xml file `/mod/test/manifest.xml`.

If this file does not exist, it could be caused by :

- installing a plugin to the wrong directory
- creating a directory under /mod that does not contain a plugin
- a bad ftp transfer
- unzipping a plugin into an extra directory (myplugin.zip unzips to `myplugin/myplugin`)

If you are on a Unix-based host and the files exist in the correct directory, check the permissions. Elgg must have read access to the files and read + execute access on the directories.

White Page (WSOD)

A blank, white page (often called a « white screen of death ») means there is a PHP syntax error. There are a few possible ca

- corrupted file - try transfering the code again to your server
- a call to a php module that was not loaded - this can happen after you install a plugin that requires a specific module.
- bad plugin - not all plugins have been written to the same quality so you should be careful which ones you install.

To find where the error is occurring, change the .htaccess file to display errors to the browser. Set display_errors to 1 and load the same page again. You should see a PHP error in your browser. Change the setting back once you've resolved the problem.

Note : If you are using the Developer's Tools plugin, go to its settings page and make sure you have « Display fatal PHP errors » enabled.

If the white screen is due to a bad plugin, remove the latest plugins that you have installed by deleting their directories and then reload the page.

Note : You can temporarily disable all plugins by creating an empty file at `mod/disabled`. You can then disable the offending module via the administrator tools panel.

If you are getting a WSOD when performing an action, like logging in or posting a blog, but there are no error messages, it's most likely caused by non-printable characters in plugin code. Check the plugin for white spaces/new lines characters after finishing php tag (`?>`) and remove them.

Page not found

If you have recently installed your Elgg site, the most likely cause for a page not found error is that `mod_rewrite` is not setup correctly on your server. There is information in the *Install Troubleshooting* page on fixing this. The second most likely cause is that your site url in your database is incorrect.

If you've been running your site for a while and suddenly start getting page not found errors, you need to ask yourself what has changed. Have you added any plugins ? Did you change your server configuration ?

To debug a page not found error :

— Confirm that the link leading to the missing page is correct. If not, how is that link being generated ?
— Confirm that the `.htaccess` rewrite rules are being picked up.

Login token mismatch

If you have to log in twice to your site and the error message after the first attempt says there was a token mismatch error, the URL in Elgg's settings does not match the URL used to access it. The most common cause for this is adding or removing the « www » when accessing the site. For example, www.elgg.org vs elgg.org. This causes a problem with session handling because of the way that web browsers save cookies.

To fix this, you can add rewrite rules. To redirect from www.elgg.org to elgg.org in Apache, the rules might look like :

```
RewriteCond %{HTTP_HOST} .
RewriteCond %{HTTP_HOST} !^elgg\.org
RewriteRule (.*) http://elgg.org/$1 [R=301,L]
```

Redirecting from non-www to www could look like this :

```
RewriteCond %{HTTP_HOST} ^elgg\.org
RewriteRule ^(.*)$ http://www.elgg.org/$1 [R=301,L]
```

If you don't know how to configure rewrite rules, ask your host for more information.

Form is missing __token or __ts fields

All Elgg actions require a security token, and this error occurs when that token is missing. This is either a problem with your server configuration or with a 3rd party plugin.

If you experience this on a new installation, make sure that your server is properly configured and your rewrite rules are correct. If you experience this on an upgrade, make sure you have updated your rewrite rules either in .htaccess (Apache) or in the server configuration.

If you are experiencing this, disable all 3rd party plugins and try again. Very old plugins for Elgg don't use security tokens. If the problem goes away when plugins are disabled, it's due to a plugin that should be updated by its author.

Maintenance mode

To take your site temporarily offline, go to Administration -> Utilities -> Maintenance Mode. Complete the form and hit save to disable your site for everyone except admin users.

Missing email

If your users are reporting that validation emails are not showing up, have them check their spam folder. It is possible that the emails coming from your server are being marked as spam. This depends on many factors such as whether your hosting provider has a problem with spammers, how your PHP mail configuration is set up, what mail transport agent your server is using, or your host limiting the number of email that you can send in an hour.

If no one gets email at all, it is quite likely your server is not configured properly for email. Your server needs a program to send email (called a Mail Transfer Agent - MTA) and PHP must be configured to use the MTA.

To quickly check if PHP and an MTA are correctly configured, create a file on your server with the following content :

```
<?php
$address = "your_email@your_host.com";

$subject = 'Test email.';

$body = 'If you can read this, your email is working.';

echo "Attempting to email $address...<br />";

if (mail($address, $subject, $body)) {
        echo 'SUCCESS!  PHP successfully delivered email to your MTA.  If you don\'t␣
↪see the email in your inbox in a few minutes, there is a problem with your MTA.';
} else {
        echo 'ERROR!  PHP could not deliver email to your MTA.  Check that your PHP␣
↪settings are correct for your MTA and your MTA will deliver email.';
}
```

Be sure to replace « your_email@your_host.com » with your actual email address. Take care to keep quotes around it ! When you access this page through your web browser, it will attempt to send a test email. This test will let you know that PHP and your MTA are correctly configured. If it fails–either you get an error or you never receive the email–you will need to do more investigating and possibly contact your service provider.

Fully configuring an MTA and PHP's email functionality is beyond the scope of this FAQ and you should search the Internet for more resources on this. Some basic information on php parameters can be found on PHP's site

Server logs

Most likely you are using Apache as your web server. Warnings and errors are written to a log by the web server and can be useful for debugging problems. You will commonly see two types of log files : access logs and error logs. Information from PHP and Elgg is written to the server error log.

— Linux – The error log is probably in /var/log/httpd or /var/log/apache2.
— Windows - It is probably inside your Apache directory.
— Mac OS - The error log is probably in /var/log/apache2/error_log

If you are using shared hosting without ssh access, your hosting provider may provide a mechanism for obtaining access to your server logs. You will need to ask them about this.

How does registration work ?

With a default setup, this is how registration works :

1. User fills out registration form and submits it
2. User account is created and disabled until validated
3. Email is sent to user with a link to validate the account
4. When a user clicks on the link, the account is validated
5. The user can now log in

Failures during this process include the user entering an incorrect email address, the validation email being marked as spam, or a user never bothering to validate the account.

User validation

By default, all users who self-register must validate their accounts through email. If a user has problems validating an account, you can validate users manually by going to Administration -> Users -> Unvalidated.

You can remove this requirement by deactivating the User Validation by Email plugin.

Note : Removing validation has some consequences : There is no way to know that a user registered with a working email address, and it may leave you system open to spammers.

Manually add user

To manually add a user, under the Administer controls go to Users. There you will see a link title « Add new User ». After you fill out the information and submit the form, the new user will receive an email with username and password and a reminder to change the password.

Note : Elgg does not force the user to change the password.

I'm making or just installed a new theme, but graphics or other elements aren't working

Make sure the theme is at the bottom of the plugin list.

Clear your browser cache and reload the page. To lighten the load on the server, Elgg instructs the browser to rarely load the CSS file. A new theme will completely change the CSS file and a refresh should cause the browser to request the CSS file again.

If you're building or modifying a theme, make sure you have disabled the simple and system caches. This can be done by enabling the Developer Tools plugin, then browsing to Administration -> Develop -> Settings. Once you're satisfied with the changes, enable the caches or performance will suffer.

Changing profile fields

Within the Administration settings of Elgg is a page for replacing the default profile fields. Elgg by default gives the administrator two choices :

— Use the default profile fields
— Replace the default with a set of custom profile fields

You cannot add new profile fields to the default ones. Adding a new profile field through the replace profile fields option clears the default ones. Before letting in users, it is best to determine what profile fields you want, what field types they should be, and the order they should appear. You cannot change the field type or order or delete fields after they have been created without wiping the entire profile blank.

More flexibility can be gained through plugins. There is at least two plugins on the community site that enable you to have more control over profile fields. The Profile Manager plugin has become quite popular in the Elgg community. It lets you add new profile fields whenever you want, change the order, group profile fields, and add them to registration.

Changing registration

The registration process can be changed through a plugin. Everything about registration can be changed : the look and feel, different registration fields, additional validation of the fields, additional steps and so on. These types of changes require some basic knowledge of HTML, CSS, PHP.

Another option is to use the Profile Manager plugin that lets you add fields to both user profiles and the registration form.

Create the plugin skeleton *Plugin skeleton*

Changing registration display Override the `account/forms/register` view

Changing the registration action handler You can write your own action to create the user's account

How do I change PHP settings using .htaccess ?

You may want to change php settings in your `.htaccess` file. This is especially true if your hosting provider does not give you access to the server's `php.ini` file. The variables could be related to file upload size limits, security, session length, or any number of other php attributes. For examples of how to do this, see the PHP documentation on this.

HTTPS login turned on accidently

If you have turned on HTTPS login but do not have SSL configured, you are now locked out of your Elgg install. To turn off this configuration parameter, you will need to edit your database. Use a tool like phpMyAdmin to view your database. Select the `config` table and delete the row that has the name `https_login`.

Using a test site

It is recommended to always try out new releases or new plugins on a test site before running them on a production site (a site with actual users). The easiest way to do this is to maintain a separate install of Elgg with dummy accounts. When testing changes it is important to use dummy accounts that are not admins to test what your users will see.

A more realistic test is to mirror the content from your production site to your test site. Following the instructions for *duplicating a site*. Then make sure you prevent emails from being sent to your users. You could write a small plugin that redirects all email to your own account (be aware of plugins that include their own custom email sending code so you'll have to modify those plugins). After this is done you can view all of the content to make sure the upgrade or new plugin is functioning as desired and is not breaking anything. If this process sounds overwhelming, please stick with running a simple test site.

500 - Internal Server Error

What is it ?

A **500 - Internal Server Error** means the web server experienced a problem serving a request.

Voir aussi :

The Wikipedia page on HTTP status codes

Possible causes

Web server configuration The most common cause for this is an incorrectly configured server. If you edited the `.htaccess` file and added something incorrect, Apache will send a 500 error.

Permissions on files It could also be a permissions problem on a file. Apache needs to be able to read Elgg's files. Using permissions 755 on directories and 644 on files will allow Apache to read the files.

When I upload a photo or change my profile picture I get a white screen

Most likely you don't have the PHP GD library installed or configured properly. You may need assistance from the administrator of your server.

CSS is missing

Wrong URL

Sometimes people install Elgg so that the base URL is `localhost` and then try to view the site using a hostname. In this case, the browser won't be able to load the CSS file. Try viewing the source of the web page and copying the link for the CSS file. Paste that into your browser. If you get a 404 error, it is likely this is your problem. You will need to change the base URL of your site.

Syntax error

Elgg stores its CSS as PHP code to provide flexibility and power. If there is a syntax error, the CSS file served to the browser may be blank. Disabling non-bundled plugins is the recommended first step.

Rewrite rules errors

A bad `.htaccess` file could also result in a 404 error when requesting the CSS file. This could happen when doing an upgrade and forgetting to also upgrade `.htaccess`.

Should I edit the database manually ?

Avertissement : No, you should never manually edit the database !

Will editing the database manually break my site ?

Yes.

Can I add extra fields to tables in the database ?

(AKA : I don't understand the Elgg *data model* so I'm going to add columns. Will you help ?)

No, this is a bad idea. Learn the *data model* and you will see that unless it's a very specific and highly customized installation, you can do everything you need within Elgg's current data model.

I want to remove users. Can't I just delete them from the elgg_entities table ?

No, it will corrupt your database. Delete them through the site.

I want to remove spam. Can't I just search and delete it from the elgg_entities table?

No, it will corrupt your database. Delete it through the site.

Someone on the community site told me to edit the database manually. Should I?

Who was it? Is it someone experienced with Elgg, like one of the core developers or a well-known plugin author? Did he or she give you clear and specific instructions on what to edit? If you don't know who it is, or if you can't understand or aren't comfortable following the instructions, do not edit the database manually.

I know PHP and MySQL and have a legitimate reason to edit the database. Is it okay to manually edit the database?

Make sure you understand Elgg's *data model* and schema first. Make a backup, edit carefully, then test copiously.

Internet Explorer (IE) login problem

Canonical URL

IE does not like working with sites that use both http://example.org and http://www.example.org. It stores multiple cookies and this causes problems. Best to only use one base URL. For details on how to do this see Login token mismatch error.

Chrome Frame

Using the chrome frame within IE can break the login process.

Emails don't support non-Latin characters

In order to support non-Latin characters, (such as Cyrillic or Chinese) Elgg requires multibyte string support to be compiled into PHP.

On many installs (e.g. Debian & Ubuntu) this is turned on by default. If it is not, you need to turn it on (or recompile PHP to include it). To check whether your server supports multibyte strings, check phpinfo.

Session length

Session length is controlled by your php configuration. You will first need to locate your `php.ini` file. In that file will be several session variables. A complete list and what they do can be found in the php manual.

File is missing an owner

There are three causes for this error. You could have an entity in your database that has an `owner_guid` of `0`. This should be extremely rare and may only occur if your database/server crashes during a write operation.

The second cause would be an entity where the owner no longer exists. This could occur if a plugin is turned off that was involved in the creation of the entity and then the owner is deleted but the delete operation failed (because the plugin is turned off). If you can figure out entity is causing this, look in your `entities` table and change the `owner_guid` to your own and then you can delete the entity through Elgg.

Avertissement : Reed the section « Should I edit the database manually ? ». Be very carefull when editing the database directly. It can break your site. **Always** make a backup before doing this.

Fixes

Database Validator plugin will check your database for these causes and provide an option to fix them. Be sure to backup the database before you try the fix option.

No images

If profile images, group images, or other files have stopped working on your site it is likely due to a misconfiguration, especially if you have migrated to a new server.

These are the most common misconfigurations that cause images and other files to stop working.

Wrong path for data directory

Make sure the data directory's path is correct in the Site Administration admin area. It should have a trailing slash.

Wrong permissions on the data directory

Check the permissions for the data directory. The data directory should be readable and writeable by the web server user.

Migrated installation with new data directory location

If you migrated an installation and need to change your data directory path, be sure to update the SQL for the filestore location as documented in the *Dupliquer une installation* instructions.

Deprecation warnings

If you are seeing many deprecation warnings that say things like

```
Deprecated in 1.7: extend_view() was deprecated by elgg_extend_view()!
```

then you are using a plugin that was written for an older version of Elgg. This means the plugin is using functions that are scheduled to be removed in a future version of Elgg. You can ask the plugin developer if the plugin will be updated or you can update the plugin yourself. If neither of those are likely to happen, you should not use that plugin.

Javascript not working

If the user hover menu stops working or you cannot dismiss system messages, that means JavaScript is broken on your site. This usually due to a plugin having bad JavaScript code. You should find the plugin causing the problem and disable it. You can do this be disabling non-bundled plugins one at a time until the problem goes away. Another approach is disabling all non-bundled plugins and then enabling them one by one until the problem occurs again.

Most web browsers will give you a hint as to what is breaking the JavaScript code. They often have a console for JavaScript errors or an advanced mode for displaying errors. Once you see the error message, you may have an easier time locating the problem.

Security

Is upgrade.php a security concern ?

Upgrade.php is a file used to run code and database upgrades. It is in the root of the directory and doesn't require a logged in account to access. On a fully upgraded site, running the file will only reset the caches and exit, so this is not a security concern.

If you are still concerned, you can either delete, move, or change permissions on the file until you need to upgrade.

Should I delete install.php ?

This file is used to install Elgg and doesn't need to be deleted. The file checks if Elgg is already installed and forwards the user to the front page if it is.

Filtering

Filtering is used in Elgg to make XSS attacks more difficult. The purpose of the filtering is to remove Javascript and other dangerous input from users.

Filtering is performed through the function `filter_tags()`. This function takes in a string and returns a filtered string. It triggers a *validate*, *input* *plugin hook*. By default Elgg comes with the htmLawed filtering code as a plugin. Developers can drop in any additional or replacement filtering code as a plugin.

The `filter_tags()` function is called on any user input as long as the input is obtained through a call to `get_input()`. If for some reason a developer did not want to perform the default filtering on some user input, the `get_input()` function has a parameter for turning off filtering.

Development

What should I use to edit php code ?

There are two main options : text editor or integrated development environment (IDE).

Text Editor

If you are new to software development or do not have much experience with IDEs, using a text editor will get you up and running the quickest. At a minimum, you will want one that does syntax highlighting to make the code easier to read. If you think you might submit patches to the bug tracker, you will want to make sure that your text editor does not change line endings. If you are using Windows, Notepad++ is a good choice. If you are on a Mac, TextWrangler is a popular choice. You could also give TextMate a try.

Integrated Development Environment

An IDE does just what its name implies : it includes a set of tools that you would normally use separately. Most IDEs will include source code control which will allow you to directly commit and update your code from your cvs repository. It may have an FTP client built into it to make the transfer of files to a remote server easier. It will have syntax checking to catch errors before you try to execute the code on a server.

The two most popular free IDEs for PHP developers are Eclipse and NetBeans. Eclipse has two different plugins for working with PHP code : PDT and PHPEclipse.

I don't like the wording of something in Elgg. How do I change it ?

The best way to do this is with a plugin.

Create the plugin skeleton

Plugin skeleton

Locate the string that you want to change

All the strings that a user sees should be in the `/languages` directory or in a plugin's languages directory (`/mod/<plugin name>/languages`). This is done so that it is easy to change what language Elgg uses. For more information on this see the developer documentation on *Internationalization* .

To find the string use `grep` or a text editor that provides searching through files to locate the string. (A good text editor for Windows is Notepad++) Let's say we want to change the string « Add friend » to « Make a new friend ». The `grep` command to find this string would be `grep -r "Add friend" *`. Using Notepad++ , you would use the « Find in files » command. You would search for the string, set the filter to `*.php`, set the directory to the base directory of Elgg, and make sure it searches all subdirectories. You might want to set it to be case sensitive also.

You should locate the string « Add friend » in `/languages/en.php`. You should see something like this in the file :

```
'friend:add' => "Add friend",
```

This means every time Elgg sees `friend:add` it replaces it with « Add friend ». We want to change the definition of `friend:add`.

Override the string

To override this definition, we will add a languages file to the plugin that we built in the first step.

1. Create a new directory : `/mod/<your plugin name>/languages`
2. Create a file in that directory called `en.php`
3. Add these lines to that file

```
<?php

return array(
   'friend:add' => 'Make a new friend',
);
```

Make sure that you do not have any spaces or newlines before the `<?php`.

You're done now and should be able to enable the plugin and see the change. If you are override the language of a plugin, make sure your plugin is loaded after the one you are trying to modify. The loading order is determined in the Tools Administration page of the admin section. As you find more things that you'd like to change, you can keep adding them to this plugin.

How do I find the code that does x ?

The best way to find the code that does something that you would like to change is to use `grep` or a similar search tool. If you do not have `grep` as a part of your operating system, you will want to install a grep tool or use a text-editor/IDE that has good searching in files. Notepad++ is a good choice for Windows users. Eclipse with PHP and NetBeans are good choices for any platform.

String Example

Let's say that you want to find where the *Log In* box code is located. A string from the *Log In* box that should be fairly unique is `Remember me`. `Grep` for that string. You will find that it is only used in the `en.php` file in the `/languages` directory. There it is used to define the *Internationalization* string `user:persistent`. `Grep` for that string now. You will find it in two places : the same `en.php` language file and in `/views/default/forms/login.php`. The latter defines the html code that makes up the *Log In* box.

Action Example

Let's say that you want to find the code that is run when a user clicks on the *Save* button when arranging widgets on a profile page. View the Profile page for a test user. Use Firebug to drill down through the html of the page until you come to the action of the edit widgets form. You'll see the url from the base is `action/widgets/move`.

`Grep` on `widgets/move` and two files are returned. One is the JavaScript code for the widgets : `/js/lib/ui.widgets.js`. The other one, `/engine/lib/widgets.php`, is where the action is registered using `elgg_register_action('widgets/reorder')`. You may not be familiar with that function in which case, you should look it up at the API reference. Do a search on the function and it returns the documentation on the function. This tells you that the action is in the default location since a file location was not specified. The default location for actions is `/actions` so you will find the file at `/actions/widgets/move.php`.

Debug mode

During the installation process you might have noticed a checkbox that controlled whether debug mode was turned on or off. This setting can also be changed on the Site Administration page. Debug mode writes a lot of extra data to your php log. For example, when running in this mode every query to the database is written to your logs. It may be useful for debugging a problem though it can produce an overwhelming amount of data that may not be related to the problem at all. You may want to experiment with this mode to understand what it does, but make sure you run Elgg in normal mode on a production server.

Avertissement : Because of the amount of data being logged, don't enable this on a production server as it can fill up the log files really quick.

What goes into the log in debug mode ?

- All database queries
- Database query profiling
- Page generation time
- Number of queries per page
- List of plugin language files
- Additional errors/warnings compared to normal mode (it's very rare for these types of errors to be related to any problem that you might be having)

What does the data look like ?

```
[07-Mar-2009 14:27:20] Query cache invalidated
[07-Mar-2009 14:27:20] ** GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggentities where guid=1 and ( (1 = 1)  and␣
↪enabled='yes') results cached
[07-Mar-2009 14:27:20] SELECT guid from elggsites_entity where guid = 1 results cached
[07-Mar-2009 14:27:20] Query cache invalidated
[07-Mar-2009 14:27:20] ** GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggentities where guid=1 and ( (1 = 1)  and␣
↪enabled='yes') results cached
[07-Mar-2009 14:27:20] ** GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggentities where guid=1 and ( (1 = 1)  and␣
↪enabled='yes') results returned from cache
[07-Mar-2009 14:27:20] ** Sub part of GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggsites_entity where guid=1 results cached
[07-Mar-2009 14:27:20] Query cache invalidated
[07-Mar-2009 14:27:20] DEBUG: 2009-03-07 14:27:20 (MST): "Undefined index:  user" in␣
↪file /var/www/elgg/engine/lib/elgglib.php (line 62)
[07-Mar-2009 14:27:20] DEBUG: 2009-03-07 14:27:20 (MST): "Undefined index:  pass" in␣
↪file /var/www/elgg/engine/lib/elgglib.php (line 62)
[07-Mar-2009 14:27:20] ****************** DB PROFILING ********************
[07-Mar-2009 14:27:20] 1 times: 'SELECT * from elggentities where guid=1 and ( ␣
↪(access_id in (2) or (owner_guid = -1) or (access_id = 0 and owner_guid = -1)) and␣
↪enabled='yes')'
...
[07-Mar-2009 14:27:20] 2 times: 'update elggmetadata set access_id = 2 where entity_
↪guid = 1'
[07-Mar-2009 14:27:20] 1 times: 'UPDATE elggentities set owner_guid='0', access_id='2
↪', container_guid='0', time_updated='1236461868' WHERE guid=1'
```

(suite sur la page suivante)

(suite de la page précédente)

```
[07-Mar-2009 14:27:20] 1 times: 'SELECT guid from elggsites_entity where guid = 1'
[07-Mar-2009 14:27:20] 1 times: 'UPDATE elggsites_entity set name='3124/944',␣
↪description='', url='http://example.org/' where guid=1'
[07-Mar-2009 14:27:20] 1 times: 'UPDATE elggusers_entity set prev_last_action = last_
↪action, last_action = 1236461868 where guid = 2'
[07-Mar-2009 14:27:20] DB Queries for this page: 56
[07-Mar-2009 14:27:20] ************************************************
[07-Mar-2009 14:27:20] Page /action/admin/site/update_basic generated in 0.
↪36997294426 seconds
```

What events are triggered on every page load ?

There are 4 *Elgg events* that are triggered on every page load :

1. plugins_boot, system
2. init, system
3. ready, system
4. shutdown, system

The first three are triggered in `Elgg\Application::bootCore`. **shutdown, system** is triggered in `\Elgg\Application\ShutdownHandler` after the response has been sent to the client. They are all *documented*.

There are other events triggered by Elgg occasionally (such as when a user logs in).

Copy a plugin

There are many questions asked about how to copy a plugin. Let's say you want to copy the `blog` plugin in order to run one plugin called `blog` and another called `poetry`. This is not difficult but it does require a lot of work. You would need to

— change the directory name
— change the names of every function (having two functions causes PHP to crash)
— change the name of every view (so as not to override the views on the original plugin)
— change any data model subtypes
— change the language file
— change anything else that was specific to the original plugin

---

Note : If you are trying to clone the `groups` plugin, you will have the additional difficulty that the group plugin does not set a subtype.

General

Voir aussi :

Trouver de l'aide

« Plugin cannot start and has been deactivated » or « This plugin is invalid »

This error is usually accompanied by more details explaining why the plugin is invalid. This is usually caused by an incorrectly installed plugin.

If you are installing a plugin called « test », there will be a test directory under mod. In that test directory there needs to be a manifest.xml file `/mod/test/manifest.xml`.

If this file does not exist, it could be caused by :

- installing a plugin to the wrong directory
- creating a directory under /mod that does not contain a plugin
- a bad ftp transfer
- unzipping a plugin into an extra directory (myplugin.zip unzips to `myplugin/myplugin`)

If you are on a Unix-based host and the files exist in the correct directory, check the permissions. Elgg must have read access to the files and read + execute access on the directories.

White Page (WSOD)

A blank, white page (often called a « white screen of death ») means there is a PHP syntax error. There are a few possible ca

- corrupted file - try transfering the code again to your server
- a call to a php module that was not loaded - this can happen after you install a plugin that requires a specific module.
- bad plugin - not all plugins have been written to the same quality so you should be careful which ones you install.

To find where the error is occurring, change the .htaccess file to display errors to the browser. Set display_errors to 1 and load the same page again. You should see a PHP error in your browser. Change the setting back once you've resolved the problem.

Note : If you are using the Developer's Tools plugin, go to its settings page and make sure you have « Display fatal PHP errors » enabled.

If the white screen is due to a bad plugin, remove the latest plugins that you have installed by deleting their directories and then reload the page.

Note : You can temporarily disable all plugins by creating an empty file at `mod/disabled`. You can then disable the offending module via the administrator tools panel.

If you are getting a WSOD when performing an action, like logging in or posting a blog, but there are no error messages, it's most likely caused by non-printable characters in plugin code. Check the plugin for white spaces/new lines characters after finishing php tag (`?>`) and remove them.

Page not found

If you have recently installed your Elgg site, the most likely cause for a page not found error is that `mod_rewrite` is not setup correctly on your server. There is information in the *Install Troubleshooting* page on fixing this. The second most likely cause is that your site url in your database is incorrect.

If you've been running your site for a while and suddenly start getting page not found errors, you need to ask yourself what has changed. Have you added any plugins ? Did you change your server configuration ?

To debug a page not found error :

— Confirm that the link leading to the missing page is correct. If not, how is that link being generated ?
— Confirm that the `.htaccess` rewrite rules are being picked up.

Login token mismatch

If you have to log in twice to your site and the error message after the first attempt says there was a token mismatch error, the URL in Elgg's settings does not match the URL used to access it. The most common cause for this is adding or removing the « www » when accessing the site. For example, www.elgg.org vs elgg.org. This causes a problem with session handling because of the way that web browsers save cookies.

To fix this, you can add rewrite rules. To redirect from www.elgg.org to elgg.org in Apache, the rules might look like :

```
RewriteCond %{HTTP_HOST} .
RewriteCond %{HTTP_HOST} !^elgg\.org
RewriteRule (.*) http://elgg.org/$1 [R=301,L]
```

Redirecting from non-www to www could look like this :

```
RewriteCond %{HTTP_HOST} ^elgg\.org
RewriteRule ^(.*)$ http://www.elgg.org/$1 [R=301,L]
```

If you don't know how to configure rewrite rules, ask your host for more information.

Form is missing __token or __ts fields

All Elgg actions require a security token, and this error occurs when that token is missing. This is either a problem with your server configuration or with a 3rd party plugin.

If you experience this on a new installation, make sure that your server is properly configured and your rewrite rules are correct. If you experience this on an upgrade, make sure you have updated your rewrite rules either in .htaccess (Apache) or in the server configuration.

If you are experiencing this, disable all 3rd party plugins and try again. Very old plugins for Elgg don't use security tokens. If the problem goes away when plugins are disabled, it's due to a plugin that should be updated by its author.

Maintenance mode

To take your site temporarily offline, go to Administration -> Utilities -> Maintenance Mode. Complete the form and hit save to disable your site for everyone except admin users.

Missing email

If your users are reporting that validation emails are not showing up, have them check their spam folder. It is possible that the emails coming from your server are being marked as spam. This depends on many factors such as whether your hosting provider has a problem with spammers, how your PHP mail configuration is set up, what mail transport agent your server is using, or your host limiting the number of email that you can send in an hour.

If no one gets email at all, it is quite likely your server is not configured properly for email. Your server needs a program to send email (called a Mail Transfer Agent - MTA) and PHP must be configured to use the MTA.

To quickly check if PHP and an MTA are correctly configured, create a file on your server with the following content :

```
<?php
$address = "your_email@your_host.com";

$subject = 'Test email.';

$body = 'If you can read this, your email is working.';

echo "Attempting to email $address...<br />";

if (mail($address, $subject, $body)) {
        echo 'SUCCESS!  PHP successfully delivered email to your MTA.  If you don\'t␣
↪see the email in your inbox in a few minutes, there is a problem with your MTA.';
} else {
        echo 'ERROR!  PHP could not deliver email to your MTA.  Check that your PHP␣
↪settings are correct for your MTA and your MTA will deliver email.';
}
```

Be sure to replace « your_email@your_host.com » with your actual email address. Take care to keep quotes around it ! When you access this page through your web browser, it will attempt to send a test email. This test will let you know that PHP and your MTA are correctly configured. If it fails–either you get an error or you never receive the email–you will need to do more investigating and possibly contact your service provider.

Fully configuring an MTA and PHP's email functionality is beyond the scope of this FAQ and you should search the Internet for more resources on this. Some basic information on php parameters can be found on PHP's site

Server logs

Most likely you are using Apache as your web server. Warnings and errors are written to a log by the web server and can be useful for debugging problems. You will commonly see two types of log files : access logs and error logs. Information from PHP and Elgg is written to the server error log.

- Linux – The error log is probably in /var/log/httpd or /var/log/apache2.
- Windows - It is probably inside your Apache directory.
- Mac OS - The error log is probably in /var/log/apache2/error_log

If you are using shared hosting without ssh access, your hosting provider may provide a mechanism for obtaining access to your server logs. You will need to ask them about this.

How does registration work ?

With a default setup, this is how registration works :

1. User fills out registration form and submits it
2. User account is created and disabled until validated
3. Email is sent to user with a link to validate the account
4. When a user clicks on the link, the account is validated
5. The user can now log in

Failures during this process include the user entering an incorrect email address, the validation email being marked as spam, or a user never bothering to validate the account.

User validation

By default, all users who self-register must validate their accounts through email. If a user has problems validating an account, you can validate users manually by going to Administration -> Users -> Unvalidated.

You can remove this requirement by deactivating the User Validation by Email plugin.

Note : Removing validation has some consequences : There is no way to know that a user registered with a working email address, and it may leave you system open to spammers.

Manually add user

To manually add a user, under the Administer controls go to Users. There you will see a link title « Add new User ». After you fill out the information and submit the form, the new user will receive an email with username and password and a reminder to change the password.

Note : Elgg does not force the user to change the password.

I'm making or just installed a new theme, but graphics or other elements aren't working

Make sure the theme is at the bottom of the plugin list.

Clear your browser cache and reload the page. To lighten the load on the server, Elgg instructs the browser to rarely load the CSS file. A new theme will completely change the CSS file and a refresh should cause the browser to request the CSS file again.

If you're building or modifying a theme, make sure you have disabled the simple and system caches. This can be done by enabling the Developer Tools plugin, then browsing to Administration -> Develop -> Settings. Once you're satisfied with the changes, enable the caches or performance will suffer.

Changing profile fields

Within the Administration settings of Elgg is a page for replacing the default profile fields. Elgg by default gives the administrator two choices :

— Use the default profile fields
— Replace the default with a set of custom profile fields

You cannot add new profile fields to the default ones. Adding a new profile field through the replace profile fields option clears the default ones. Before letting in users, it is best to determine what profile fields you want, what field types they should be, and the order they should appear. You cannot change the field type or order or delete fields after they have been created without wiping the entire profile blank.

More flexibility can be gained through plugins. There is at least two plugins on the community site that enable you to have more control over profile fields. The Profile Manager plugin has become quite popular in the Elgg community. It lets you add new profile fields whenever you want, change the order, group profile fields, and add them to registration.

Changing registration

The registration process can be changed through a plugin. Everything about registration can be changed : the look and feel, different registration fields, additional validation of the fields, additional steps and so on. These types of changes require some basic knowledge of HTML, CSS, PHP.

Another option is to use the Profile Manager plugin that lets you add fields to both user profiles and the registration form.

Create the plugin skeleton *Plugin skeleton*

Changing registration display Override the `account/forms/register` view

Changing the registration action handler You can write your own action to create the user's account

How do I change PHP settings using .htaccess ?

You may want to change php settings in your `.htaccess` file. This is especially true if your hosting provider does not give you access to the server's `php.ini` file. The variables could be related to file upload size limits, security, session length, or any number of other php attributes. For examples of how to do this, see the PHP documentation on this.

HTTPS login turned on accidently

If you have turned on HTTPS login but do not have SSL configured, you are now locked out of your Elgg install. To turn off this configuration parameter, you will need to edit your database. Use a tool like phpMyAdmin to view your database. Select the `config` table and delete the row that has the name `https_login`.

Using a test site

It is recommended to always try out new releases or new plugins on a test site before running them on a production site (a site with actual users). The easiest way to do this is to maintain a separate install of Elgg with dummy accounts. When testing changes it is important to use dummy accounts that are not admins to test what your users will see.

A more realistic test is to mirror the content from your production site to your test site. Following the instructions for *duplicating a site*. Then make sure you prevent emails from being sent to your users. You could write a small plugin that redirects all email to your own account (be aware of plugins that include their own custom email sending code so you'll have to modify those plugins). After this is done you can view all of the content to make sure the upgrade or

new plugin is functioning as desired and is not breaking anything. If this process sounds overwhelming, please stick with running a simple test site.

500 - Internal Server Error

What is it ?

A **500 - Internal Server Error** means the web server experienced a problem serving a request.

Voir aussi :

The Wikipedia page on HTTP status codes

Possible causes

Web server configuration The most common cause for this is an incorrectly configured server. If you edited the `.htaccess` file and added something incorrect, Apache will send a 500 error.

Permissions on files It could also be a permissions problem on a file. Apache needs to be able to read Elgg's files. Using permissions 755 on directories and 644 on files will allow Apache to read the files.

When I upload a photo or change my profile picture I get a white screen

Most likely you don't have the PHP GD library installed or configured properly. You may need assistance from the administrator of your server.

CSS is missing

Wrong URL

Sometimes people install Elgg so that the base URL is `localhost` and then try to view the site using a hostname. In this case, the browser won't be able to load the CSS file. Try viewing the source of the web page and copying the link for the CSS file. Paste that into your browser. If you get a 404 error, it is likely this is your problem. You will need to change the base URL of your site.

Syntax error

Elgg stores its CSS as PHP code to provide flexibility and power. If there is a syntax error, the CSS file served to the browser may be blank. Disabling non-bundled plugins is the recommended first step.

Rewrite rules errors

A bad `.htaccess` file could also result in a 404 error when requesting the CSS file. This could happen when doing an upgrade and forgetting to also upgrade `.htaccess`.

Should I edit the database manually ?

Avertissement : No, you should never manually edit the database !

Will editing the database manually break my site ?

Yes.

Can I add extra fields to tables in the database ?

(AKA : I don't understand the Elgg *data model* so I'm going to add columns. Will you help ?)

No, this is a bad idea. Learn the *data model* and you will see that unless it's a very specific and highly customized installation, you can do everything you need within Elgg's current data model.

I want to remove users. Can't I just delete them from the elgg_entities table ?

No, it will corrupt your database. Delete them through the site.

I want to remove spam. Can't I just search and delete it from the elgg_entities table ?

No, it will corrupt your database. Delete it through the site.

Someone on the community site told me to edit the database manually. Should I ?

Who was it ? Is it someone experienced with Elgg, like one of the core developers or a well-known plugin author ? Did he or she give you clear and specific instructions on what to edit ? If you don't know who it is, or if you can't understand or aren't comfortable following the instructions, do not edit the database manually.

I know PHP and MySQL and have a legitimate reason to edit the database. Is it okay to manually edit the database ?

Make sure you understand Elgg's *data model* and schema first. Make a backup, edit carefully, then test copiously.

Internet Explorer (IE) login problem

Canonical URL

IE does not like working with sites that use both http://example.org and http://www.example.org. It stores multiple cookies and this causes problems. Best to only use one base URL. For details on how to do this see Login token mismatch error.

Chrome Frame

Using the chrome frame within IE can break the login process.

Emails don't support non-Latin characters

In order to support non-Latin characters, (such as Cyrillic or Chinese) Elgg requires multibyte string support to be compiled into PHP.

On many installs (e.g. Debian & Ubuntu) this is turned on by default. If it is not, you need to turn it on (or recompile PHP to include it). To check whether your server supports multibyte strings, check phpinfo.

Session length

Session length is controlled by your php configuration. You will first need to locate your `php.ini` file. In that file will be several session variables. A complete list and what they do can be found in the php manual.

File is missing an owner

There are three causes for this error. You could have an entity in your database that has an `owner_guid` of `0`. This should be extremely rare and may only occur if your database/server crashes during a write operation.

The second cause would be an entity where the owner no longer exists. This could occur if a plugin is turned off that was involved in the creation of the entity and then the owner is deleted but the delete operation failed (because the plugin is turned off). If you can figure out entity is causing this, look in your `entities` table and change the `owner_guid` to your own and then you can delete the entity through Elgg.

> **Avertissement :** Reed the section « Should I edit the database manually ? ». Be very carefull when editing the database directly. It can break your site. **Always** make a backup before doing this.

Fixes

Database Validator plugin will check your database for these causes and provide an option to fix them. Be sure to backup the database before you try the fix option.

No images

If profile images, group images, or other files have stopped working on your site it is likely due to a misconfiguration, especially if you have migrated to a new server.

These are the most common misconfigurations that cause images and other files to stop working.

Wrong path for data directory

Make sure the data directory's path is correct in the Site Administration admin area. It should have a trailing slash.

Wrong permissions on the data directory

Check the permissions for the data directory. The data directory should be readable and writeable by the web server user.

Migrated installation with new data directory location

If you migrated an installation and need to change your data directory path, be sure to update the SQL for the filestore location as documented in the *Dupliquer une installation* instructions.

Deprecation warnings

If you are seeing many deprecation warnings that say things like

```
Deprecated in 1.7: extend_view() was deprecated by elgg_extend_view()!
```

then you are using a plugin that was written for an older version of Elgg. This means the plugin is using functions that are scheduled to be removed in a future version of Elgg. You can ask the plugin developer if the plugin will be updated or you can update the plugin yourself. If neither of those are likely to happen, you should not use that plugin.

Javascript not working

If the user hover menu stops working or you cannot dismiss system messages, that means JavaScript is broken on your site. This usually due to a plugin having bad JavaScript code. You should find the plugin causing the problem and disable it. You can do this be disabling non-bundled plugins one at a time until the problem goes away. Another approach is disabling all non-bundled plugins and then enabling them one by one until the problem occurs again.

Most web browsers will give you a hint as to what is breaking the JavaScript code. They often have a console for JavaScript errors or an advanced mode for displaying errors. Once you see the error message, you may have an easier time locating the problem.

Security

Is upgrade.php a security concern ?

Upgrade.php is a file used to run code and database upgrades. It is in the root of the directory and doesn't require a logged in account to access. On a fully upgraded site, running the file will only reset the caches and exit, so this is not a security concern.

If you are still concerned, you can either delete, move, or change permissions on the file until you need to upgrade.

Should I delete install.php ?

This file is used to install Elgg and doesn't need to be deleted. The file checks if Elgg is already installed and forwards the user to the front page if it is.

Filtering

Filtering is used in Elgg to make XSS attacks more difficult. The purpose of the filtering is to remove Javascript and other dangerous input from users.

Filtering is performed through the function `filter_tags()`. This function takes in a string and returns a filtered string. It triggers a *validate*, *input* *plugin hook*. By default Elgg comes with the htmLawed filtering code as a plugin. Developers can drop in any additional or replacement filtering code as a plugin.

The `filter_tags()` function is called on any user input as long as the input is obtained through a call to `get_input()`. If for some reason a developer did not want to perform the default filtering on some user input, the `get_input()` function has a parameter for turning off filtering.

Development

What should I use to edit php code ?

There are two main options : text editor or integrated development environment (IDE).

Text Editor

If you are new to software development or do not have much experience with IDEs, using a text editor will get you up and running the quickest. At a minimum, you will want one that does syntax highlighting to make the code easier to read. If you think you might submit patches to the bug tracker, you will want to make sure that your text editor does not change line endings. If you are using Windows, Notepad++ is a good choice. If you are on a Mac, TextWrangler is a popular choice. You could also give TextMate a try.

Integrated Development Environment

An IDE does just what its name implies : it includes a set of tools that you would normally use separately. Most IDEs will include source code control which will allow you to directly commit and update your code from your cvs repository. It may have an FTP client built into it to make the transfer of files to a remote server easier. It will have syntax checking to catch errors before you try to execute the code on a server.

The two most popular free IDEs for PHP developers are Eclipse and NetBeans. Eclipse has two different plugins for working with PHP code : PDT and PHPEclipse.

I don't like the wording of something in Elgg. How do I change it ?

The best way to do this is with a plugin.

Create the plugin skeleton

Plugin skeleton

Locate the string that you want to change

All the strings that a user sees should be in the `/languages` directory or in a plugin's languages directory (`/mod/<plugin name>/languages`). This is done so that it is easy to change what language Elgg uses. For more information on this see the developer documentation on *Internationalization* .

To find the string use `grep` or a text editor that provides searching through files to locate the string. (A good text editor for Windows is Notepad++) Let's say we want to change the string « Add friend » to « Make a new friend ». The `grep` command to find this string would be `grep -r "Add friend" *`. Using Notepad++ , you would use the « Find in files » command. You would search for the string, set the filter to `*.php`, set the directory to the base directory of Elgg, and make sure it searches all subdirectories. You might want to set it to be case sensitive also.

You should locate the string « Add friend » in `/languages/en.php`. You should see something like this in the file :

```
'friend:add' => "Add friend",
```

This means every time Elgg sees `friend:add` it replaces it with « Add friend ». We want to change the definition of `friend:add`.

Override the string

To override this definition, we will add a languages file to the plugin that we built in the first step.

1. Create a new directory : `/mod/<your plugin name>/languages`
2. Create a file in that directory called `en.php`
3. Add these lines to that file

```
<?php

return array(
    'friend:add' => 'Make a new friend',
);
```

Make sure that you do not have any spaces or newlines before the `<?php`.

You're done now and should be able to enable the plugin and see the change. If you are override the language of a plugin, make sure your plugin is loaded after the one you are trying to modify. The loading order is determined in the Tools Administration page of the admin section. As you find more things that you'd like to change, you can keep adding them to this plugin.

How do I find the code that does x ?

The best way to find the code that does something that you would like to change is to use `grep` or a similar search tool. If you do not have `grep` as a part of your operating system, you will want to install a grep tool or use a text-editor/IDE that has good searching in files. Notepad++ is a good choice for Windows users. Eclipse with PHP and NetBeans are good choices for any platform.

String Example

Let's say that you want to find where the *Log In* box code is located. A string from the *Log In* box that should be fairly unique is `Remember me`. `Grep` for that string. You will find that it is only used in the `en.php` file in the `/languages` directory. There it is used to define the *Internationalization* string `user:persistent`. `Grep` for that string now. You will find it in two places : the same `en.php` language file and in `/views/default/forms/login.php`. The latter defines the html code that makes up the *Log In* box.

Action Example

Let's say that you want to find the code that is run when a user clicks on the *Save* button when arranging widgets on a profile page. View the Profile page for a test user. Use Firebug to drill down through the html of the page until you come to the action of the edit widgets form. You'll see the url from the base is `action/widgets/move`.

`Grep` on `widgets/move` and two files are returned. One is the JavaScript code for the widgets : `/js/lib/ui.widgets.js`. The other one, `/engine/lib/widgets.php`, is where the action is registered using `elgg_register_action('widgets/reorder')`. You may not be familiar with that function in which case, you should look it up at the API reference. Do a search on the function and it returns the documentation on the function. This tells you that the action is in the default location since a file location was not specified. The default location for actions is `/actions` so you will find the file at `/actions/widgets/move.php`.

Debug mode

During the installation process you might have noticed a checkbox that controlled whether debug mode was turned on or off. This setting can also be changed on the Site Administration page. Debug mode writes a lot of extra data to your php log. For example, when running in this mode every query to the database is written to your logs. It may be useful for debugging a problem though it can produce an overwhelming amount of data that may not be related to the problem at all. You may want to experiment with this mode to understand what it does, but make sure you run Elgg in normal mode on a production server.

> **Avertissement :** Because of the amount of data being logged, don't enable this on a production server as it can fill up the log files really quick.

What goes into the log in debug mode ?

— All database queries
— Database query profiling
— Page generation time
— Number of queries per page
— List of plugin language files
— Additional errors/warnings compared to normal mode (it's very rare for these types of errors to be related to any problem that you might be having)

What does the data look like ?

```
[07-Mar-2009 14:27:20] Query cache invalidated
[07-Mar-2009 14:27:20] ** GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggentities where guid=1 and ( (1 = 1)  and␣
↪enabled='yes') results cached
[07-Mar-2009 14:27:20] SELECT guid from elggsites_entity where guid = 1 results cached
[07-Mar-2009 14:27:20] Query cache invalidated
[07-Mar-2009 14:27:20] ** GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggentities where guid=1 and ( (1 = 1)  and␣
↪enabled='yes') results cached
[07-Mar-2009 14:27:20] ** GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggentities where guid=1 and ( (1 = 1)  and␣
↪enabled='yes') results returned from cache
[07-Mar-2009 14:27:20] ** Sub part of GUID:1 loaded from DB
[07-Mar-2009 14:27:20] SELECT * from elggsites_entity where guid=1 results cached
[07-Mar-2009 14:27:20] Query cache invalidated
[07-Mar-2009 14:27:20] DEBUG: 2009-03-07 14:27:20 (MST): "Undefined index:  user" in␣
↪file /var/www/elgg/engine/lib/elgglib.php (line 62)
[07-Mar-2009 14:27:20] DEBUG: 2009-03-07 14:27:20 (MST): "Undefined index:  pass" in␣
↪file /var/www/elgg/engine/lib/elgglib.php (line 62)
[07-Mar-2009 14:27:20] ***************** DB PROFILING ********************
[07-Mar-2009 14:27:20] 1 times: 'SELECT * from elggentities where guid=1 and ( ␣
↪(access_id in (2) or (owner_guid = -1) or (access_id = 0 and owner_guid = -1)) and␣
↪enabled='yes')'
...
[07-Mar-2009 14:27:20] 2 times: 'update elggmetadata set access_id = 2 where entity_
↪guid = 1'
[07-Mar-2009 14:27:20] 1 times: 'UPDATE elggentities set owner_guid='0', access_id='2
↪', container_guid='0', time_updated='1236461868' WHERE guid=1'
[07-Mar-2009 14:27:20] 1 times: 'SELECT guid from elggsites_entity where guid = 1'
[07-Mar-2009 14:27:20] 1 times: 'UPDATE elggsites_entity set name='3124/944',␣
↪description='', url='http://example.org/' where guid=1'
[07-Mar-2009 14:27:20] 1 times: 'UPDATE elggusers_entity set prev_last_action = last_
↪action, last_action = 1236461868 where guid = 2'
[07-Mar-2009 14:27:20] DB Queries for this page: 56
[07-Mar-2009 14:27:20] ***************************************************
[07-Mar-2009 14:27:20] Page /action/admin/site/update_basic generated in 0.
↪36997294426 seconds
```

What events are triggered on every page load ?

There are 4 *Elgg events* that are triggered on every page load :

1. plugins_boot, system
2. init, system
3. ready, system
4. shutdown, system

The first three are triggered in `Elgg\Application::bootCore`. **shutdown, system** is triggered in `\Elgg\Application\ShutdownHandler` after the response has been sent to the client. They are all *documented*.

There are other events triggered by Elgg occasionally (such as when a user logs in).

Copy a plugin

There are many questions asked about how to copy a plugin. Let's say you want to copy the `blog` plugin in order to run one plugin called `blog` and another called `poetry`. This is not difficult but it does require a lot of work. You would need to

— change the directory name
— change the names of every function (having two functions causes PHP to crash)
— change the name of every view (so as not to override the views on the original plugin)
— change any data model subtypes
— change the language file
— change anything else that was specific to the original plugin

---

Note : If you are trying to clone the `groups` plugin, you will have the additional difficulty that the group plugin does not set a subtype.

3.7.3 Roadmap

What direction is the project going ? What exciting new features are coming soon ?

We do not publish detailed roadmaps, but it's possible to get a sense for our general direction by utilizing the following resources :

— Our feedback and planning group is used to host early discussion about what will be worked on next.
— Our Github milestones represent a general direction for the future releases of Elgg. This is the closest thing to a traditional roadmap that we have.
— Github pull requests will give you a good idea of what's currently being developed, but nothing is sure until the PR is actually checked in.
— We use the developer blog to post announcements of features that have recently been checked in to our development branch, which gives the surest indication of what features will be available in the next release.

Values

We have several overarching goals/values that affect the direction of Elgg. Enhancements generally must promote these values in order to be accepted.

Accessibility

Elgg-based sites should be usable by anyone anywhere. That means we'll always strive to make Elgg :

— Device-agnostic – mobile, tablet, desktop, etc. friendly
— Language-agnostic – i18n, RTL, etc.
— Capability-agnostic – touch, keyboard, screen-reader friendly

Testability

We want to **make manual testing unnecessary** for core developers, plugin authors, and site administrators by promoting and enabling fast, automated testing at every level of the Elgg stack.

We think APIs are broken if they require plugin authors to write untestable code. We know there are a lot of violations of this principle in core currently and are working to fix it.

We look forward to a world where the core developers do not need to do any manual testing to verify the correctness of code contributed to Elgg. Similarly, we envision a world where site administrators can upgrade and install new plugins with confidence that everything works well together.

TODO : other goals/values ?

FAQ

When will feature X be implemented ?

We cannot promise when features will get implemented because new features are checked into Elgg only when someone is motivated enough to implement the feature and submit a pull request. The best we can do is tell you to look out for what features existing developers have expressed interest in working on.

The best way to ensure a feature gets implemented is to discuss it with the core team and implement it yourself. See our *Guides du contributeur* guide if you're interested. We love new contributors !

Do not rely on future enhancements if you're on the fence as to whether to use Elgg. Evaluate it given the current feature set. Upcoming features will almost certainly not materialize within your timeline.

When is version X.Y.Z going to be released ?

The next version will be released when the core team feels it's ready and has time to cut the release. http://github.com/Elgg/Elgg/issues/milestones will give you some rough ideas of timeline.

3.7.4 Release Policy

What to expect when upgrading Elgg.

We adhere to semantic versioning.

Follow the blog to stay up to date on the latest releases.

Contents

Patch/Bugfix Releases (2.1.x)

Every two weeks.

Bugfix releases are made regularly to make sure Elgg stays stable, secure, and bug-free. The higher the third digit, the more tested and stable the release is.

Since bugfix release focus on fixing bugs and not making major changes, themes and plugins should work from bugfix release to bugfix release.

Minor/Feature Releases (2.x.0)

Every three months.

Whenever we introduce new features, we'll bump the middle version number. These releases aren't as mature as bugfix release, but are considered stable and useable.

We make every effort to be backward compatible in these releases, so plugins should work from minor release to minor release.

However, plugins might need to be updated to make use of the new features.

Major/Breaking Releases (x.0.0)

Every year.

Inevitably, improving Elgg requires breaking changes and a new major release is made. These releases are opportunities for the core team to make strategic, breaking changes to the underlying platform. Themes and plugins from older versions are not expected to work without modification on different major releases.

We may remove deprecated APIs, but we will not remove APIs without first deprecating them.

Elgg's dependencies may be upgraded by their major version or removed entirely. We will not remove any dependences before a major release, but we do not « deprecate » dependencies or issue any warnings before removing them.

Your package, plugin, or app should declare its own dependencies directly so that this does not cause a problem.

Alphas, Betas, and Release Candidates

Before major releases (and sometimes before feature releases), the core team will offer a pre-release version of Elgg to get some real-world testing and feedback on the release. These are meant for testing only and should not be used on a live site.

SemVer 2.0 does not define a particular meaning for pre-releases, but we approach alpha, beta, and rc releases with these general guidelines :

An `-alpha.X` pre-release means that there are still breaking changes planned, but the feature set of the release is frozen. No new features or breaking changes can be proposed for that release.

A `-beta.X` pre-release means that there are no known breaking changes left to be included, but there are known regressions or critical bugs left to fix.

An `-rc.X` pre-release means that there are no known regressions or critical bugs left to be fixed. This version could become the final stable version of Elgg if no new blockers are reported.

Backwards compatibility

Some parts of the system need some additional clarification if we are talking about being backwards compatible. Everything that is considered public API needs to adhere to the backwards compatibility rules that are part of semantic versioning.

Classes and functions

Classes and functions marked with `@internal` are not considered part of the public API and can be changed / removed at any time. If a class is marked with `@internal` all properties and methods in that class are considered private API and therefor can be changed / removed at any time.

Event and plugin hook callbacks

All event and plugin hook callbacks should never be called directly but only be called by their respective event / plugin hook.

The name of the callback function is considered API as plugin developers need to be able to rely on the fact that they can (un)register a callback. This only applies if the callback still serves the same purpose. If a callback becomes obsolete its allowed to be removed from the system.

Avertissement : Exceptions to these rules are the callback functions related to the following `system` events, these callbacks can be renamed / removed at any time.

- `plugins_load`
- `plugins_boot`
- `init`
- `ready`
- `shutdown`
- `upgrade`

Views

— View names are API.
— View arguments ($vars array) are API.
— Removing views or renaming views follows API deprecation policies.
— Adding new views requires a minor version change.
— View output is not API and can be changed between patch releases.

3.7.5 Support policy

As of Elgg 2.0, each minor release receives bug and security fixes only until the next minor release.

Contents

Long Term Support Releases

Within each major version, the last minor release is designated for long term support (« LTS ») and will receive bug fixes until 1 year after the release of the next major version and security fixes until the 2nd following major version release.

E.g. 2.3 is the last minor release within 2.x. It will receive bug fixes until 1 year aftr 3.0 is released and security fixes until 4.0 is released.

Voir aussi :

— *Release Policy*
— *Signaler des problèmes*

Bugs

When bugs are found, a good faith effort will be made to patch the LTS release, but **not all fixes will be back-ported.** E.g. some fixes may depend on new APIs, break backwards compatibility, or require significant refactoring.

Important : If a fix risks stability of the LTS branch, it will not be included.

Security issues

When a security issue is found every effort will be made to patch the LTS release.

> **Attention :** Please report any security issue to **security @ elgg . org**

Timeline

Below is a table outlining the specifics for each release (future dates are tentative) :

Version	First stable release	Bug fixes through	Security fixes through
1.12	July 2015	April 2019	April 2019
2.0	December 2015	March 2016	
2.1	March 2016	June 2016	
2.2	June 2016	November 2016	
2.3 LTS	November 2016	April 2020	**Until 4.0**
3.0 LTS	April 2019		
4.0	TBD		

3.7.6 History

The name comes from a town in Switzerland. It also means « elk » or « moose » in Danish.

Elgg's initial funding was by a company called Curverider Ltd, which was started by David Tosh and Ben Werdmuller. In 2010, Curverider was acquired by Thematic Networks and control of the open-source project was turned over to The Elgg Foundation. Today, Elgg is a community-driven open source project and has a variety of contributors and supporters.