Elgg Documentation

Release master

Various

Jan 23, 2020

Contents

Elgg (pronunciation) is an open source rapid development framework for socially aware web applications. It is a great fit for building any app where users log in and share information.

CHAPTER 1

Features

- **Well-documented core API** that allows developers to kick start their new project with a simple learning curve
- **Composer** is the package manager of choice that greatly simplifes installation and maintenance of Elgg core and plugins
- **Flexible system of hooks and events** that allows plugins to extend and modify most aspects of application's functionality and behavior
- **Extendable system of views** that allows plugins to collaborate on application's presentation layer and built out complex custom themes
- **Cacheable system of static assets** that allows themes and plugins to serve images, stylesheets, fonts and scripts bypassing the engine
- **User authentication** is powered by pluggable auth modules, which allow applications to implement custom authentication protocols
- **Security** is ensured by built-in anti CSRF validation, strict XSS filters, HMAC signatures, latest cryptographic approaches to password hashing
- **Client-side API** powered by asynchronous JavaScript modules via RequireJS and a build-in Ajax service for easy communication with the server
- **Flexible entity system** that allows applications to prototype new types of content and user interactions
- **Opinionated data model** with a consolidated API layer that allows the developers to easily interface with the database
- **Access control system** that allows applications to build granular content access policies, as well as create private networks and intranets
- **Groups** - out of the box support for user groups
- **File storage** powered by flexible API that allows plugins to store user-generated files and serve/stream them without booting the engine
- **Notifications service** that allows applications to subscribe users to on-site and email notifications and implement integrations with other their-party services

- **RPC web services** that can be used for complex integrations with external applications and mobile clients
- **Internationalization** and localization of Elgg applications is simple and can be integrated with third-party services such as Transifex
- **Elgg community** that can help with any arising issues and hosts a repository of **1000+ open source plugins**

Under the hood:

- Elgg is a modular OOP framework that is driven by DI services
- NGINX or Apache compatible
- Symfony2 HTTP Foundation handles requests and responses
- RequireJS handles AMD
- Zend Mail handles outgoing email
- htmLawed XSS filters
- DBAL
- Phinx database migrations
- CSS-Crush for CSS preprocessing
- Imagine for image manipulation
- Persistent caching with Memcached and/or Redis
- Error handling with Monolog

CHAPTER 2

Examples

It has been used to build all kinds of social apps:

- open networks (similar to Facebook)
- topical (like the Elgg Community)
- private/corporate intranets
- dating
- educational
- company blog

This is the canonical documentation for the Elgg project.

CHAPTER 3

Continue Reading

3.1 Getting Started

Discover if Elgg is right for your community.

3.1.1 Bundled plugins

Elgg comes with a set of plugins. These provide the basic functionality for your social network.

Blog

A weblog, or blog, is arguably one of the fundamental DNA pieces of most types of social networking site. The simplest form of personal publishing, it allows for text-based notes to be published in reverse-chronological order. Commenting is also an important part of blogging, turning an individual act of publishing into a conversation.

Elgg's blog expands this model by providing per-entry access controls and cross-blog tagging. You can control exactly who can see each individual entry, as well as find other entries that people have written on similar topics. You can also see entries written by your friends (that you have access to).

See also:

Blogging on Wikipedia

Dashboard

The dashboard is bundled with both the full and core-only Elgg packages. This is a users portal to activity that is important to them both from within the site and from external sources. Using Elgg's powerful widget API, it is possible to build widgets that pull out relevant content from within an Elgg powered site as well as grab information from third party sources such as Twitter or Flickr (providing those widgets exist). A users dashboard is not the same as their profile, whereas

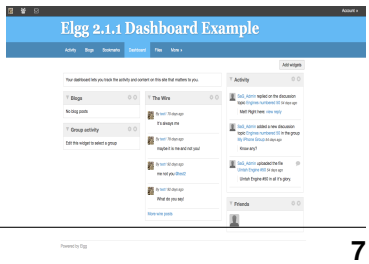

Fig. 1: A typical Elgg dashboard

the profile is for consumption by others, the dashboard is a space for users to use for their own needs.

Diagnostics

For the technically savvy user, system diagnostics enables you to quickly evaluate the server environment, Elgg code, and plugins of an Elgg install. Diagnostics is a core system plugin that comes turned on by default with Elgg. To download the diagnostics file, follow the steps below. The file is a dump of all sorts of useful information.

To use:

- Log in as Administrator
- Go to Administration -> Administer -> Utilities ->System Diagnostics
- Click 'Download'

System diagnostics dump file contents:

- List of all Elgg files along with a hash for each file
- List of all the plugins
- PHP superglobals
- PHP settings
- Apache settings
- **Elgg CONFIG values**
  - language strings
  - site settings
  - database settings
  - plugin hooks
  - actions
  - views
  - page handlers
  - much more

File repository

The file repository allows users to upload any kind of file. As with everything in an Elgg system, you can filter uploaded files by tag and restrict access so that they're only visible by the people you want them to be. Each file may also have comments attached to it.

There are a number of different uses for this functionality

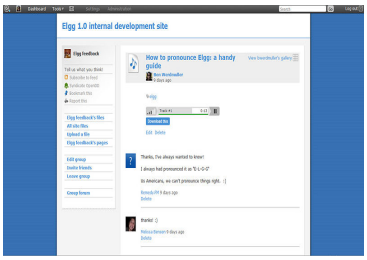

Fig. 2: A file in an Elgg file repository

Photo gallery

When a user uploads photographs or other pictures, they are automatically collated into an Elgg photo gallery that can be browsed through. Users can also see pictures that their friends have uploaded, or see pictures attached to a group. Clicking into an individual file shows a larger version of the photo.

Podcasting

An Elgg file repository RSS feed automatically doubles as an RSS feed, so you can subscribe to new audio content using programs like iTunes.

Special content

It is possible for other plugins to add to the players available for different content types. It's possible for a plugin author to embed a viewer for Word documents, for example.

Note for developers

To add a special content type player, create a plugin with views of the form `file/specialcontent/mime/type`. For example, to create a special viewer for Word documents, you would create a view called `file/specialcontent/application/msword`, because `application/msword` is the MIME-type for Word documents. Within this view, the `ElggEntity` version of the file will be referenced as `$vars['entity']`. Therefore, the URL of the downloadable file is:

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

Using this, it should be possible to develop most types of embeddable viewers.

Friends

Being a social network framework Elgg supports relationships between users.

By default any user can befriend any other user, it's like following the activity of the other user.

After enabling friendship requests as a feature of the Friends plugin, when user A wants to be friends with user B, user B has to approve the request. Upon approval user A will be friends with user B and user B will be friends with user A.

Groups

Once you have found others with similar interests - or perhaps you are part of a research groups or a course/class - you may want to have a more structured setting to share content and discuss ideas. This is where Elgg's powerful group building can be used. You can create and moderate as many groups as you like

- You can keep all group activity private to the group or you can use the 'make public' option to disseminate work to the wider public.
- Each group produces granular RSS feeds, so it is easy to follow group developments
- Each group has its own URL and profile

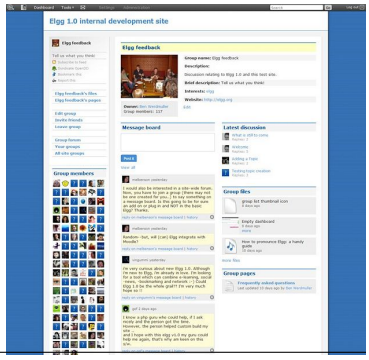

Fig. 3: A typical group profile

- Each group comes with a *File repository*, forum, pages and messageboard

Messageboard

The messageboard - similar to 'The Wall' in Facebook or a comment wall in other networks is a plugin that lets users put a messageboard widget on their profile. Other users can then post messages that will appear on the messageboard. You can then reply directly to any message and view the history between yourself and the person posting the message.

Messages

Private messaging can be sent to users by clicking on their avatar or profile link, providing you have permission. Then, using the built in *WYSIWYG editor*, it is possible to format the message. Each user has their own inbox and sentbox. It is possible to be notified via email of new messages.

When users first login, they will be notified about any new message by the messages notification mechanism in their top toolbar.

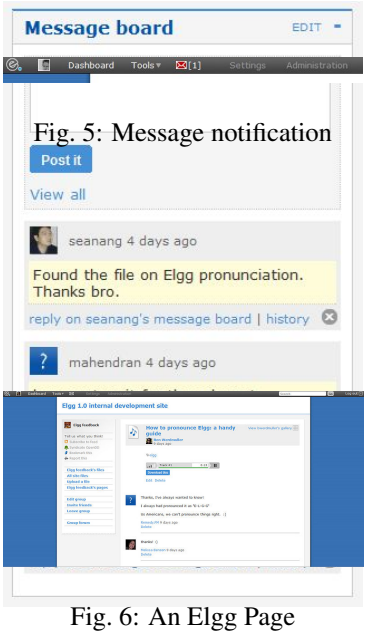

Fig. 5: Message notification

Fig. 6: An Elgg Page

Fig. 4: A sample messageboard placed on the profile

Pages

The pages plugin allows you to save and store hierarchically-organized pages of text, and restrict both reading and writing privileges to them. This means that you can collaboratively create a set of documents with a loose collection of people, participate in a writing process with a formal group, or simply use the functionality to write a document that only you can see, and only choose to share it once it's done. The easy navigation menu allows you to see the whole document structure from any page. You can create as many of these structures as you like; each individual page has its own access controls, so you can reveal portions of the structure while keeping others hidden. In keeping with all other elements in Elgg, you can add comments on a page, or search for pages by tag.

Usage

Pages really come into their own in two areas, firstly as a way for users to build up things such as a resume, reflective documentation and so on. The second thing is in the area of collaboration, especially when in the context of groups. With the powerful access controls on both read and write, this plugin is ideal for collaborative document creation.

Note: Developers should note that there are actually 2 types of pages:

1. Top-level pages (with subtype `page_top`)
2. Normal pages (with subtype `page`)

Profile

The profile plugin is bundled with both the full and core-only Elgg packages. The intention is that it can be disabled and replaced with another profile plugin if you wish. It provides a number of pieces of functionality which many consider fundamental to the concept of a social networking site, and is unique within the plugins because the profile icon it defines is referenced as standard from all over the system.

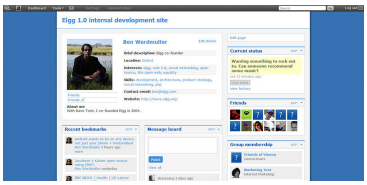

Fig. 7: An Elgg profile

User details

This provides information about a user, which is configurable from within the plugin's start.php file. You can change the available profile fields form the admin panel. Each profile field has its own access restriction, so users can choose exactly who can see each individual element. Some of the fields contain tags (for example *skills*) limiting access to a field will also limit who can find you by that tag.

User avatar

The user avatar represents a user (or a group) throughout the site. By default, this includes a context-sensitive menu that allows you to perform actions on the user it belongs to wherever you see their avatar. For example, you can add them as a friend, send an internal message, and more. Each plugin can add to this context menu, so its full contents will vary depending on the functionality active in the current Elgg site.

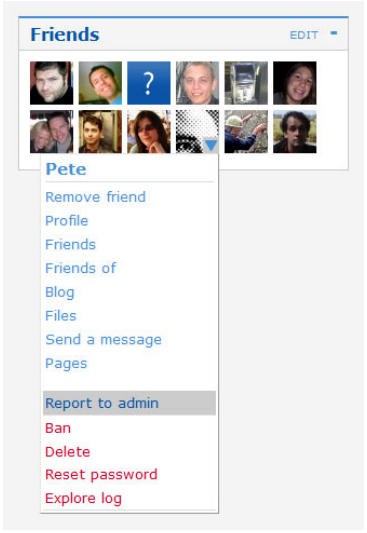

Fig. 8: The Elgg context menu

Notes for developers

Using a different profile icon To replace the profile icon, or provide more content, extend the `icon/user/default` view.

Adding to the context menu The context menu can be expanded by registering a *plugin hook* for 'register' 'menu:user_hover', the following sections have special meaning:

- **default** for non-active links (eg to read a blog)
- **admin** for links accessible by administrators only

In each case, the user in question will be passed as `$params['entity']`.

The Wire

Elgg wire plugin "The Wire" is Twitter-style microblogging plugin that allows users to post notes to the wire.

User validation by e-mail

The uservalidationbyemail plugin adds a step to the user registration process. After the user registered on the site, an e-mail is sent to their e-mail address in order to validate that the e-mail address belongs to the user. In the e-mail is an verification link, only after the user clicked on the link will the account of the user be able to login to the site.

The process for the user

1. The user creates an account by going to the registration page of your site
2. After the account is created the user lands on a page with instructions to check their e-mail account for the validation e-mail
3. In the validation e-mail is a link to confirm their e-mail address
4. After clicking on the link, the account is validated
5. If possible the user gets logged in

If the user tries to login before validating their account an error is shown to indicate that the user needs to check their e-mail account. Also the validation e-mail is sent again.

Options for site administrators

A site administrator can take some actions on unvalidated accounts. Under Administration -> Users -> Unvalidated is a list of unvalidated users. The administrator can manualy validate or delete the user. Also the option to resend the validation e-mail is present.

The following plugins are also bundled with Elgg, but are not (yet) documented

- activity
- bookmarks
- ckeditor
- custom_index
- developers
- discussions
- embed
- externalpages
- friends_collections
- garbagecollector
- invitefriends
- likes
- members
- notifications
- reportedcontent
- search
- site_notifications
- system_log
- tagcloud
- web_services

3.1.2 License

MIT or GPLv2

A full Elgg package that includes the framework and a core set of plugins is available under version 2 of the GNU General Public License (GPLv2). We also make the framework (without the plugins) available under the MIT license.

FAQ

The following answers are provided as a convenience to you; they are not legal counsel. Consult with a lawyer to be sure about the answers to these questions. The Elgg Foundation cannot be held responsible for decisions you make based on what you read on this page.

For questions not answered here, please refer to the official FAQ for the GPLv2.

How much does Elgg cost?

Elgg is free to download, install, and use. If you'd like to donate, we do appreciate our financial supporters!

Can I remove the Elgg branding/links?

Yes.

Can I modify the source code?

Yes, but in general we recommend you make your modifications as plugins so that when a new version of Elgg is released, the upgrade process is as painless as possible.

Can I charge my users membership fees?

Yes.

If I modify Elgg, do I have to make the changes available?

No, if you are using Elgg to provide a service, you do not have to make the source available. If you distribute a modified version of Elgg, then you must include the source code for the changes.

If I use Elgg to host a network, does The Elgg Foundation have any rights over my network?

No.

What's the difference between the MIT and GPL versions?

Plugins are not included with the MIT version.

You can distribute a commercial product based on Elgg using the MIT version without making your modifications available.

With the GPL licensed version, you have to include make your modifications of the framework public if you redistribute the framework.

Why are plugins missing from the MIT version?

The plugins were developed under the GPL license, so they cannot be released under an MIT license. Also, some plugins include external dependencies that are not compatible with the MIT license.

May I distribute a plugin for Elgg under a commercial license?

We believe you can, since plugins typically depend only the core framework and the framework is available under the MIT license. That said, we really recommend you consult with a lawyer on this particular issue to be absolutely sure.

Note that plugins released via the community site repository must be licensed under a GPLv2-compatible license. They do not necessarily have to be GPLv2, just compatible (like MIT).

Can we build our own tool that uses Elgg and sell that tool to our clients?

Yes, but then your clients will be free to redistribute that tool under the terms of the GPLv2.

3.1.3 Installation

Get your own instance of Elgg running in no time.

Contents

Requirements

- MySQL 5.5.3+ (5.0.0+ if upgrading an existing installation)
- PHP 7.1+ with the following extensions:
  - GD (for graphics processing)
  - PDO (for database connection)
  - JSON (for AJAX responses, etc.)

  - XML (for reading plugin manifest files, etc.)
  - Multibyte String support (for i18n)
  - Proper configuration and ability to send email through an MTA
- Web server with support for URL rewriting

Official support is provided for the following configurations:

- **Apache server**
  - Apache with the rewrite module enabled
  - PHP running as an Apache module
- **Nginx server**
  - Nginx with PHP-FPM using FastCGI

By "official support", we mean that:

- Most development and testing is performed with these configurations
- Much of the installation documentation is written assuming Apache or Nginx is used
- Priority on bug reports is given to Apache and Nginx users if the bug is web server specific (but those are rare).

Browser support policy

Feature branches support the latest 2 versions of all major browsers as were available at the time of the first stable release on that branch.

Bugfix release will not alter browser support, even if a new version of the browser has since been released.

Major browsers here means all of the following, plus their mobile counterparts:

- Android Browser
- Chrome
- Firefox
- IE
- Safari

"Support" may mean that we take advantage of newer, unimplemented technologies but provide a JavaScript polyfill for the browsers that need it.

You may find that Elgg happens to work on unsupported browsers, but compatibility may break at any time, even during a bugfix release.

Overview

Upload Elgg

With Composer (recommended if comfortable with CLI):

```
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
```

From pre-packaged zip (recommended if not comfortable with CLI):

- Download the latest version of Elgg
- Upload the ZIP file with an FTP client to your server
- Unzip the files in your domain's document root.

Create a data folder

Elgg needs a special folder to store uploaded files including profile icons and photos. You will need to create this directory.

Attention: For security reasons, this folder **MUST** be stored outside of your document root. If you created it under `/www/` or `/public_html/`, you're doing it wrong.

Once this folder has been created, you'll need to make sure the web server Elgg is running on has permission to write to and create directories in it. This shouldn't be a problem on Windows-based servers, but if your server runs Linux, Mac OS X or a UNIX variant, you'll need to set the permissions on the directory.

If you are using a graphical FTP client to upload files, you can usually set permissions by right clicking on the folder and selecting 'properties' or 'Get Info'.

Note: Directories must be executable to be read and written to. The suggested permissions depend upon the exact server and user configuration. If the data directory is owned by the web server user, the recommended permissions are `750`.

Warning: Setting your data directory to `777` will work, but it is insecure and is not recommended. If you are unsure how to correctly set permissions, contact your host for more information.

Create a MySQL database

Using your database administration tool of choice (if you're unsure about this, ask your system administrator), create a new MySQL database for Elgg. You can create a MySQL database with any of the following tools:

Make sure you add a user to the database with all privileges and record the database name, username and password. You will need this information when installing Elgg.

Set up Cron

Elgg uses timed requests to your site to perform background tasks like sending notifications or performing database cleanup jobs. You need to configure the *cron* to be able to use those kind of features.

Visit your Elgg site

Once you've performed these steps, visit your Elgg site in your web browser. Elgg will take you through the rest of the installation process from there. The first account that you create at the end of the installation process will be an administrator account.

A note on settings.php and .htaccess

The Elgg installer will try to create two files for you:

- `elgg-config/settings.php`, which contains local environment configuration for your installation
- `.htaccess`, which allows Elgg to generate dynamic URLs

If these files can't be automatically generated, for example because the web server doesn't have write permissions in the directories, Elgg will tell you how to create them. You could also temporarily change the permissions on the root directory and the engine directory. Set the permissions on those two directories so that the web server can write those two files, complete the install process, and them change the permissions back to their original settings. If, for some reason, this won't work, you will need to:

- In `elgg-config/`, copy `settings.example.php` to `settings.php`, open it up in a text editor and fill in your database details
- On Apache server, copy `install/config/htaccess.dist` to `.htaccess`
- On Nginx server copy `install/config/nginx.dist` to `/etc/nginx/sites-enabled` and adjust it's contents

Other Configurations

- Cloud9
- Homestead
- EasyPHP
- IIS
- MAMP
- MariaDB
- Nginx
- Ubuntu
- Virtual hosts
- XAMPP

Troubleshooting

Help! I'm having trouble installing Elgg

First:

- Recheck that your server meets the technical requirements for Elgg.
- Follow the environment-specific instructions if need be

- Have you verified that `mod_rewrite` is being loaded?
- Is the mysql apache being loaded?

Keep notes on steps that you take to fix the install. Sometimes changing some setting or file to try to fix a problem may cause some other problem later on. If you need to start over, just delete all the files, drop your database, and begin again.

I can't save my settings on installation (I get a 404 error when saving settings)

Elgg relies on the `mod_rewrite` Apache extension in order to simulate certain URLs. For example, whenever you perform an action in Elgg, or when you visit a user's profile, the URL is translated by the server into something Elgg understands internally. This is done using rules defined in an `.htaccess` file, which is Apache's standard way of defining extra configuration for a site.

This error suggests that the `mod_rewrite` rules aren't being picked up correctly. This may be for several reasons. If you're not comfortable implementing the solutions provided below, we strongly recommend that you contact your system administrator or technical support and forward this page to them.

The `.htaccess`, if not generated automatically (that happens when you have problem with `mod_rewrite`), you can create it by renaming `install/config/htaccess.dist` file you find with elgg package to `.htaccess`. Also if you find a `.htaccess` file inside the installation path, but you are still getting 404 error, make sure the contents of `.htaccess` are same as that of `install/config/htaccess.dist`.

''mod_rewrite'' isn't installed.

Check your `httpd.conf` to make sure that this module is being loaded by Apache. You may have to restart Apache to get it to pick up any changes in configuration. You can also use PHP info to check to see if the module is being loaded.

The rules in ''.htaccess'' aren't being obeyed.

In your virtual host configuration settings (which may be contained within `httpd.conf`), change the AllowOverride setting so that it reads:

```
AllowOverride all
```

This will tell Apache to pick up the `mod_rewrite` rules from `.htaccess`.

Elgg is not installed in the root of your web directory (ex: http://example.org/elgg/ instead of http://example.org/)

The install script redirects me to "action" when it should be "actions"

This is a problem with your `mod_rewrite` setup.

> **Attention:** DO NOT, REPEAT, DO NOT change any directory names!

I installed in a subdirectory and my install action isn't working!

If you installed Elgg so that it is reached with an address like http://example.org/mysite/ rather than http://example.org/, there is a small chance that the rewrite rules in .htaccess will not be processed correctly. This is usually due to using an alias with Apache. You may need to give mod_rewrite a pointer to where your Elgg installation is.

- Open up .htaccess in a text editor

- Where prompted, add a line like `RewriteBase /path/to/your/elgg/installation/` (Don't forget the trailing slash)
- Save the file and refresh your browser.

Please note that the path you are using is the **web** path, minus the host.

For example, if you reach your elgg install at http://example.org/elgg/, you would set the base like this:

```
RewriteBase /elgg/
```

Please note that installing in a subdirectory does not require using RewriteBase. There are only some rare circumstances when it is needed due to the set up of the server.

I did everything! mod_rewrite is working fine, but still the 404 error

Maybe there is a problem with the file .htaccess. Sometimes the elgg install routine is unable to create one and unable to tell you that. If you are on this point and tried everything that is written above:

- check if it is really the elgg-created .htaccess (not only a dummy provided from the server provider)
- if it is not the elgg provided htaccess file, use the htaccess_dist (rename it to .htaccess)

I get an error message that the rewrite test failed after the requirements check page

I get the following messages after the requirements check step (step 2) of the install:

> We think your server is running the Apache web server.
>
> The rewrite test failed and the most likely cause is that AllowOverride is not set to All for Elgg's directory. This prevents Apache from processing the .htaccess file which contains the rewrite rules.
>
> A less likely cause is Apache is configured with an alias for your Elgg directory and you need to set the RewriteBase in your .htaccess. There are further instructions in the .htaccess file in your Elgg directory.

After this error, everinteraction with the web interface results in a error 500 (Internal Server Error)

This is likely caused by not loading the "filter module by un-commenting the

> #LoadModule filter_module modules/mod_filter.so

line in the "httpd.conf" file.

the Apache "error.log" file will contain an entry similar to:

> ... .htaccess: Invalid command 'AddOutputFilterByType', perhaps misspelled or defined by a module not included in the server configuration

There is a white page after I submit my database settings

Check that the Apache mysql module is installed and is being loaded.

I'm getting a 404 error with a really long url

If you see a 404 error during the install or on the creation of the first user with a url like: `http://example.com/homepages/26/d147515119/htdocs/elgg/action/register` that means your site url is incorrect in your sites_entity table in your database. This was set by you on the second page of the install. Elgg tries to guess the correct value but has difficulty with shared hosting sites. Use phpMyAdmin to edit this value to the correct base url.

I am having trouble setting my data path

This is highly server specific so it is difficult to give specific advice. If you have created a directory for uploading data, make sure your http server can access it. The easiest (but least secure) way to do this is give it permissions `777`. It is better to give the web server ownership of the directory and limit the permissions.

> **Warning:** Setting directory permissions to `777` allows the **ENTIRE** internet to place files in your directory structure an possibly infect you webserver with malware. Setting permissions to `750` should be more than enough.

The top cause of this issue is PHP configured to prevent access to most directories using open_basedir. You may want to check with your hosting provider on this.

Make sure the path is correct and ends with a /. You can check the path in your database in the config table.

If you only have ftp access to your server and created a directory but do not know the path of it, you might be able to figure it out from the www file path set in your config database table. Asking for help from your hosting help team is recommended at this stage.

I can't validate my admin account because I don't have an email server!

While it's true that normal accounts (aside from those created from the admin panel) require their email address to be authenticated before they can log in, the admin account does not.

Once you have registered your first account you will be able to log in using the credentials you have provided!

I have tried all of these suggestions and I still cannot install Elgg

It is possible that during the process of debugging your install you have broken something else. Try doing a clean install:

- drop your elgg database
- delete your data directory
- delete the Elgg source files
- start over

If that fails, seek the help of the Elgg community. Be sure to mention what version of Elgg you are installing, details of your server platform, and any error messages that you may have received including ones in the error log of your server.

3.1.4 Developer Overview

This is a quick developer introduction to Elgg. It covers the basic approach to working with Elgg as a framework, and mentions some of the terms and technologies used.

See the *Developer Guides* for tutorials or the *Design Docs* for in-depth discussion on design.

Database and Persistence

Elgg uses MySQL 5.5 or higher for data persistence, and maps database values into Entities (a representation of an atomic unit of information) and Extenders (additional information and descriptions about Entities). Elgg supports additional information such as relationships between Entities, activity streams, and various types of settings.

Plugins

Plugins change the behavior or appearance of Elgg by overriding views, or by handling events and plugin hooks. All changes to an Elgg site should be implemented through plugins to ensure upgrading core is easy.

Actions

Actions are the primary way users interact with an Elgg site. Actions are registered by plugins.

Events and Plugin Hooks

Events and Plugin Hooks are used in Elgg Plugins to interact with the Elgg engine under certain circumstances. Events and hooks are triggered at strategic times throughout Elgg's boot and execution process, and allows plugins to modify or cancel the default behavior.

Views

Views are the primary presentation layer for Elgg. Views can be overridden or extended by Plugins. Views are categories into a Viewtype, which hints at what sort of output should be expected by the view.

JavaScript

Elgg uses an AMD-compatible JavaScript system provided by RequireJs. Bundled with Elgg are jQuery, jQuery UI, jQuery Form, and jQuery UI Autocomplete.

Plugins can load their own JS libs.

Internationalization

Elgg's interface supports multiple languages, and uses Transifex for translation.

Caching

Elgg uses two caches to improve performance: a system cache and SimpleCache.

3rd party libraries

The use of 3rd party libraries in Elgg is managed by using Composer dependencies. Examples of 3rd party libraries are jQuery, RequireJs or Zend mail.

To get a list of all the Elgg dependencies check out the Packagist page for Elgg.

Database Seeding

Elgg provides some base database seeds to populate the database with entities for testing purposes.

You can run the following commands to seed and unseed the database.

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

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

Plugins can register their own seeds via `'seeds', 'database'` hook. The handler must return the class name of the seed, which must extend `\Elgg\Database\Seeder\Seed` class.

3.1.5 Elgg CLI

Contents

elgg-cli command line tools

Depending on how you installed Elgg and your server configuration you can access‘‘elgg-cli‘‘ binaries as one of the following from the root of your Elgg installation:

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

Available commands

```
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]

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

# Remove seeded faked entities
```

(continues on next page)

(continued from previous page)

```
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]
```

Adding custom commands

Plugins can add their commands to the CLI application, by adding command class name via `'commands','cli'` hook. Command class must extend `\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;

});
```

Custom commands are based on Symfony Console Commands. Please refer to their documentation for more details.

3.2 Administrator Guides

Best practices for effectively managing an Elgg-based site.

3.2.1 Getting Started

You have installed Elgg and worked through any potential initial issues. What now? Here are some suggestions on how to to familiarize yourself with Elgg.

Focus first on core functionality

When you're new to Elgg, it's best to explore the stock features in core and its bundled plugins before installing any third party plugins. It's tempting install every interesting plugin from the community site, but exploring the core features builds a familiarity with Elgg's expected behavior, and prevents introducing any confusing bugs from third party plugin into your new Elgg network.

Elgg installs with a basic set of social network plugins activated: blogs, social bookmarking, files, groups, likes, message boards, wiki-like pages, user profiles, and microblogging. To change the plugins that are activated, log in as an admin user, then use the topbar to browse to Administration, then to Plugins on the right sidebar.

Note: The user you create during installation is an admin user.

Create test users

Users can be created two ways in stock Elgg:

1. Complete the signup process using a different email address and username. (Logout first or use a different browser!)
2. Add a user through the Admin section by browsing to Administration -> Users -> Add New User.

Note: Users that self-register must validate their account through email before they can log in. Users that an admin creates are already validated.

Explore user functionality

Use your test users to create blogs, add widgets to your profile or dashboard, post to the Wire (microblogging), and create pages (wiki-like page creation). Investigate the Settings on the topbar. This is where a user sets notification settings and configures tools (which will be blank because none of the default plugins add controls here).

Explore admin functionality

All of the admin controls are found by clicking Administration in the topbar. The has a dashboard with a widget that explains the various sections. Change options in the Configure menu to change how Elgg looks and acts.

Extending Elgg

After exploring what Elgg can do out of the box, install some themes and plugins. You can find many plugins and themes at the community site that have been developed by third parties. These plugins do everything from changing language strings, to adding chat, to completely redesigning Elgg's interface. Because these plugins are not official, be certain to check the comments to make sure you only install well-written plugins by high quality developers.

3.2.2 Composer installation

The easiest way to keep your Elgg site up-to-date is by using Composer. Composer will take care of installing all the required dependencies of all plugins and Elgg, while also keeping those depencies up-to-date without having conflicts.

Contents

Install Composer

https://getcomposer.org/download/

Install Elgg as a Composer Project

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

This will create a composer.json file based of the Elgg starter project which has the basics of installing Elgg.

Open your browser

Go to your browser and install Elgg via the installation interface

Setup version controls

This step is optional but highly recommended. It'll allow you to easily manage the installation of the same plugin versions between environments (development/testing/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
```

Install plugins

Install plugins as Composer depencies. This assumes that a plugin has been registered on Packagist

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

Commit

Make sure `composer.lock` is not ignored in `.gitignore`

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

Deploy to production

Initial Deploy

```
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
```

Subsequent Deploys

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

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

3.2.3 Upgrading Elgg

This document will guide you through steps necessary to upgrade your Elgg installation to the latest version.

If you've written custom plugins, you should also read the developer guides for *information on upgrading plugin code* for the latest version of Elgg.

Contents

Advice

- **Back up your database, data directory and code**
- Mind any version-specific comments below
- Version below 2.0 are advised to only upgrade **one minor version at a time**
- You can upgrade from any minor version to any higher minor version in the same major (2.0 -> 2.1 or 2.0 -> 2.3)
- You can only upgrade the latest minor version in the previous major version to any minor version in the next version (2.3 -> 3.0 or 2.3 -> 3.2, but not 2.2 -> 3.x).
- From Elgg 2.3.* you can upgrade to any future version of Elgg without having to go through each minor version (e.g. you can upgrade directly from 2.3.8 to 3.2.5, without having to upgrade to 3.0 and 3.1)
- Try out the new version on a test site before doing an upgrade
- Report any problems in plugins to the plugin authors
- If you are a plugin author you can report any backwards-compatibility issues to GitHub

From 2.3 to 3.0

1. Update `composer.json`

If you have used Elgg's starter project to install Elgg 2.3, you may need to update your `composer.json`:

- change platform requirements to PHP >= 7.0
- optionally, set autoloader optimization parameters
- optionally, disable fxp-asset plugin in favor of asset-packagist

Your `composer.json` would look something like this (depending what changes you may have introduced yourself):

```
{
        "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. Update `.htaccess`

Find the line:

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

And replace it with:

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

3a. Composer Upgrade (recommended)

If you had your Elgg 2.3 project installed using composer, you can follow this sequence:

Back up your database, data directory, and 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
```

Note: In some cases the command line upgrade will fail because some database schema changes need to be applied first. In that case you need to execute the *Phinx migrations* manually

3b. Manual Upgrade (legacy approach)

Manual upgrades are a major undertaking for site admins. We discourage you from maintaining an Elgg installation using ZIP dist packages. Save yourself some time by learning how to use `composer` and version control systems, such as `git`. This task will also be complicated if you have third-party plugins and/or have made any modifications to core files!

1. **Back up your database, data directory, and code**
2. Log in as an admin to your site
3. Download the new version of Elgg from http://elgg.org
4. **Update the files**
    - If upgrading to a major version, you need to overwrite all core files and remove any files that were removed from Elgg core, as they may interfere with proper functioning of your site.
    - If upgrading to a minor version or patching, you need to overwrite all core files.
5. **Merge any new changes to the rewrite rules**
    - For Apache from `install/config/htaccess.dist` into `.htaccess`
    - For Nginx from `install/config/nginx.dist` into your server configuration (usually inside `/etc/nginx/sites-enabled`)
6. Visit http://your-elgg-site.com/upgrade.php
7. Execute asynchronous upgrades at http://your-elgg-site.com/admin/upgrades

---

Note: Any modifications should have been written within plugins, so that they are not lost on overwriting. If this is not the case, take care to maintain your modifications.

Note: If you are unable to access `upgrade.php` script and receive an error, add `$CONFIG->security_protect_upgrade = false;` to your `settings.php` and remove it after you have completed all of the upgrade steps.

Note: If you encounter issues with plugins during the upgrade, add an empty file called `disabled` in your `/mod/` directory. This will disable the plugins, so that you can finish the core upgrade. You can then deal with issues on per-plugin basis.

If you have installed Elgg using a dist package but would now like to switch to composer:

- Upgrade your current installation using Manual Upgrade method
- Move your codebase to a temporary location
- Create a new composer project using Elgg's starter project following *installation instructions* in the root directory of your current installation
- Copy third-party plugins from your old installation into `/mod` directory
- Run Elgg's installer using your browser or `elgg-cli` tool
- When you reach the database step, provide the same credentials you have used for manual installation, Elgg will understand that is's an existing installation and will not override any database values
- Optionally commit your new project to version control

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
```

Earlier versions

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

Plugins can modify the behavior of and add new features to Elgg.

Contents

Where to get plugins

Plugins can be obtained from:

- The Elgg Community
- Github
- Third-party sites (typically for a price)

If no existing plugins meet your needs, you can hire a developer or *create your own*.

The Elgg Community

Finding Plugins

Sort based on most popular

On the community plugin page, you can sort by date uploaded (Filter: Newest) or number of downloads (Filter: Most downloads). Sorting by the number of downloads is a good idea if you are new to Elgg and want to see which plugins are frequently used by other administrators. These will often (but not always) be higher quality plugins that provide significant capabilities.

Use the plugin tag search

Next to the filtering control on the plugin page is a search box. It enables you to search by tags. Plugins authors choose the tags.

Look for particular plugin authors

The quality of plugins varies substantially. If you find a plugin that works well on your site, you can check what else that plugin author has developed by clicking on their name when viewing a plugin.

Evaluating Plugins

Look at the comments and ratings

Before downloading and using a plugin, it is always a good idea to read through the comments that others have left. If you see people complaining that the plugin does not work or makes their site unstable, you probably want to stay away from that plugin. The caveat to that is that sometimes users ignore installation instructions or incorrectly install a plugin and then leave negative feedback. Further, some plugin authors have chosen to not allow comments.

Install on a test site

If you are trying out a plugin for the first time, it is a bad idea to install it on your production site. You should maintain a separate test site for evaluating plugins. It is a good idea to slowly roll out new plugins to your production site even after they pass your evaluation on your test site. This enables you to isolate problems introduced by a new plugin.

Types of plugins

Themes

Themes are plugins that modify the look-and-feel of your site. They generally include stylesheets, client-side scripts and views that alter the default presentation and behavior of Elgg.

Language Packs

Language packs are plugins that provide support for other languages.

Language packs can extend and include translations for language strings found in the core, core plugins and/or third-party plugins.

Some of the language packs are already included in the core, and can be found in `languages` directory off Elgg's root directory. Individual plugins tend to include their translations under the `languages` directory within the plugin's root.

This structure makes it easy to create new language packs that supercede existing language strings or add support for new languages.

Installation

All plugins reside in the `mod` directory of your Elgg installation.

To install a new plugin:

- extract (unzip) contents of the plugin distribution package
- copy/FTP the extracted folder into the `mod` directory of your Elgg installation, making sure that `manifest.xml` is directly under the plugin directory (e.g. if you were to install a plugin called `my_elgg_plugin`, plugin's manifest would need to be found at `mod/my_elgg_plugin/manifest.xml`)
- activate the plugin from your admin panel

To activate a plugin:

- Log in to your Elgg site with your administrator account
- Go to Administration -> Configure -> Plugins
- Find your plugin in the list of installed plugins and click on the 'enable' button.

Plugin order

Plugins are loaded according to the order they are listed on the Plugins page. The initial ordering after an install is more or less random. As more plugins are added by an administrator, they are placed at the bottom of the list.

Some general rules for ordering plugins:

- A theme plugin should be last or at least near the bottom
- A plugin that modifies the behavior of another plugin should be lower in the plugin list

3.2.5 Performance

Make your site run as smoothly and responsively as possible.

Contents

Can Elgg scale to X million users?

People often ask whether Elgg can scale to large installations.

First, we might stop and ask, "where are you planning to get all those users?" Seriously, though, this is a really interesting problem. Making Elgg scale is, if anything, an issue of technical engineering. It's interesting but more or less a solved problem. Computer science doesn't work differently for Elgg than for Google, for example. Getting millions of users? That's like the Holy Grail of the entire tech industry.

Second, as with most things in life, the answer is "it depends":

- How active are your users?
- What hardware is Elgg running on?
- Are your plugins behaving well?

Improving the efficiency of the Elgg engine is an ongoing project, although there are limits to the amount that any script can do.

If you are serious about scalability you will probably want to look at a number of things yourself.

Measure first

There is no point in throwing resources at a problem if you don't know:

- what the problem is
- what resources the problem needs
- where those resources are needed

Invest in some kind of profiling to tell you where your bottleneck is, especially if you're considering throwing significant money at a problem.

Tune MySQL

Elgg makes extensive use of the back end database, making many trips on each pageload. This is perfectly normal and a well configured database server will be able to cope with thousands of requests per second.

Here are some configuration tips that might help:

- Make sure that MySQL is configured to use an appropriate my.cnf for the size of your website.
- Increase the amount of memory available to PHP and MySQL (you will have to increase the amount of memory available to the php process in any case)

Enable caching

Generally, if a program is slow, that is because it is repeatedly performing an expensive computation or operation. Caching allows the system to avoid doing that work over and over again by using memory to store the results so that you can skip all the work on subsequent requests. Below we discuss several generally-available caching solutions relevant to Elgg.

Simplecache

By default, views are cached in the Elgg data directory for a given period of time. This removes the need for a view to be regenerated on every page load.

This can be disabled by setting `$CONFIG->simplecache_enabled = false;` For best performance, make sure this value is set to `true`.

This does lead to artifacts during development if you are editing themes in your plugin as the cached version will be used in preference to the one provided by your plugin.

The simple cache can be disabled via the administration menu. It is recommended that you do this on your development platform if you are writing Elgg plugins.

This cache is automatically flushed when a plugin is enabled, disabled or reordered, or when upgrade.php is executed.

For best performance, you can also create a symlink from `/cache/` in your www root dir to the `assetroot` directory specified in your config (by default it's located under `/path/to/dataroot/caches/views_simplecache/`:

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

If your webserver supports following symlinks, this will serve files straight off disk without booting up PHP each time.

For security reasons, some webservers (e.g. Apache in version 2.4) might follow the symlinks by default only if the owner of the symlink source and target match. If the cache symlink fails to work on your server, you can change the owner of the cache symlink itself (and not the `/views_simplecache/` directory) with

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

In this example it's assumed that the `/views_simplecache/` directory in the data directory is owned by the wwwrun account that belongs to the www group. If this is not the case on your server, you have to modify the chown command accordingly.

System cache

The location of views are cached so that they do not have to be discovered (profiling indicated that page load took a non-linear amount of time the more plugins were enabled due to view discovery). Elgg also caches information like the language mapping and class map.

This can be disabled by setting `$CONFIG->system_cache_enabled = false;` For best performance, make sure this value is set to `true`.

This is currently stored in files in your dataroot (although later versions of Elgg may use memcache). As with the simple cache it is flushed when a plugin is enabled, disabled or reordered, or when upgrade.php is executed.

The system cache can be disabled via the administration menu, and it is recommended that you do this on your development platform if you are writing Elgg plugins.

Boot cache

Elgg has the ability to cache numerous resources created and fetched during the boot process. To configure how long this cache is valid you must set a TTL in your `settings.php` file: `$CONFIG->boot_cache_ttl = 3600;`

Look at the Stash documentation for more info about the TTL.

Database query cache

For the lifetime of a given page's execution, a cache of all `SELECT` queries is kept. This means that for a given page load a given select query will only ever go out to the database once, even if it is executed multiple times. Any write to the database will flush this cache, so it is advised that on complicated pages you postpone database writes until the end of the page or use the `execute_delayed_*` functionality. This cache will be automatically cleared at the end of a page load.

You may experience memory problems if you use the Elgg framework as a library in a PHP CLI script. This can be disabled by setting `$CONFIG->db_disable_query_cache = true;`

Etags and Expires headers

These technologies tell your users' browsers to cache static assets (CSS, JS, images) locally. Having these enabled greatly reduces server load and improves user-perceived performance.

Use the Firefox yslow plugin or Chrome DevTools Audits to confirm which technologies are currently running on your site.

If the static assets aren't being cached:

- Verify that you have these extensions installed and enabled on your host
- Update your .htaccess file, if you are upgrading from a previous version of Elgg
- Enable *Simplecache*, which turns select views into browser-cacheable assets

Memcached

Libmemcached was created by Brian Aker and was designed from day one to give the best performance available to users of Memcached.

See also:

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

Installation requirements:

- php-memcached
- libmemcached
- memcached

Configuration:

Uncomment and populate the following sections in `settings.php`

```
$CONFIG->memcache = true;

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

Optionaly if you run multiple Elgg installations but use ony one Memcache server, you may want to add a namespace prefix. In order to do this, uncomment the following line

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

Squid

We have had good results by using Squid to cache images for us.

Bytecode caching

There are numerous PHP code caches available on the market. These speed up your site by caching the compiled byte code from your script meaning that your server doesn't have to compile the PHP code each time it is executed.

Direct file serving

If your server can be configured to support the X-Sendfile or X-Accel headers, you can configure it to be used in `settings.php`. This allows your web server to directly stream files to the client instead of using PHP's `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
        }
}
```

See also:

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.

Hosting

Don't expect to run a site catering for millions of users on a cheap shared host. You will need to have your own host hardware and access over the configuration, as well as lots of bandwidth and memory available.

Memory, CPU and bandwidth

Due to the nature of caching, all caching solutions will require memory. It is a fairly cheap return to throw memory and CPU at the problem.

On advanced hardware it is likely that bandwidth is going to be your bottleneck before the server itself. Ensure that your host can support the load you are suggesting.

Configuration

Lastly, take a look at your configuration as there are a few gotchas that can catch people.

For example, out of the box, Apache can handle quite a high load. However, most distros of Linux come with mysql configured for small sites. This can result in Apache processes getting stalled waiting to talk to one very overloaded MySQL process.

Check for poorly-behaved plugins

Plugins can be programmed in a very naive way and this can cause your whole site to feel slow.

Try disabling some plugins to see if that noticeably improves performance. Once you've found a likely offender, go to the original plugin author and report your findings.

Use client-rendered HTML

We've found that at a certain point, much of the time spent on the server is simply building the HTML of the page with Elgg's views system.

It's very difficult to cache the output of templates since they can generally take arbitrary inputs. Instead of trying to cache the HTML output of certain pages or views, the suggestion is to switch to an HTML-based templating system so that the user's browser can cache the templates themselves. Then have the user's computer do the work of generating the output by applying JSON data to those templates.

This can be very effective, but has the downside of being significant extra development cost. The Elgg team is looking to integrate this strategy into Elgg directly, since it is so effective especially on pages with repeated or hidden content.

3.2.6 Cron

Contents

What does it do?

Cron is a program available on Unix-based operating systems that enables users to run commands and scripts at set intervals or at specific times.

Elgg's cron handler allows administrators and plugin developers to setup jobs that need to be executed at set intervals.

Most common examples of cron jobs in Elgg include:

- sending out queued notifications
- rotating the system log in the database
- collecting garbage in the database (compacting the database by removing entries that are no longer required)

Plugins can add jobs by registering a plugin hook handler for one of the following cron intervals:

- `minute` - Run every minute
- `fiveminute` - Run every 5 minutes
- `fifteenmin` - Run every 15 minutes
- `halfhour` - Run every 30 minutes
- `hourly` - Run every hour
- `daily` - Run every day
- `weekly` - Run every week

- `monthly` - Run every month
- `yearly` - Run every year

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

   $events = my_plugin_get_upcoming_events();

   foreach ($events as $event) {
      $attendees = $event->getAttendees();

      // notify
   }
});
```

How does it work?

`crontab` must be setup in such a way as to activate Elgg cron handler every minute, or at a specific interval. Once cron tab activates the cron job, Elgg executes all hook handlers attached to that interval.

If you have SSH access to your Linux servers, type `crontab -e` and add your crontab configuration.

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

The above command will run every minute and activate all due cron jobs.

Optionally you can activate handlers for a specific interval:

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

More information about cron can be found at:

3.2.7 Backup and Restore

Contents

Introduction

Why

Shared hosting providers typically don't provide an automated way to backup your Elgg installation. This article will address a method of accomplishing this task.

In IT there are often many ways to accomplish the same thing. Keep that in mind. This article will explain one method to backup and restore your Elgg installation on a shared hosting provider that uses the CPanel application. However, the ideas presented here can be tailored to other applications as well. The following are typical situations that might require a procedure such as this:

- Disaster Recovery
- Moving your Elgg site to a new host
- Duplicating an installation

What

Topics covered:

- Full backups of the Elgg directories and MySQL databases are performed daily (automated)
- The backups are sent to an off-site location via FTP (automated)
- The local backups are deleted after successful transfer to the off-site location (automatic)
- Five days of backups will be maintained (automated)
- Restoration of data to the new host (manual)

This process was composed with assistance from previous articles in the Elgg documentation wiki.

Assumptions

The following assumptions have been made:

- The Elgg program directory is `/home/userx/public_html`
- The Elgg data directory is `/home/userx/elggdata`
- You've created a local directory for your backups at `/home/userx/sitebackups`
- You have an off-site FTP server to send the backup files to

- The directory that you will be saving the off-site backups to is `/home/usery/sitebackups/`
- You will be restoring the site to a second shared hosting provider in the `/home/usery/public_html` directory

Important: Be sure to replace `userx`, `usery`, `http://mynewdomain.com` and all passwords with values that reflect your actual installation!

Creating a usable backup - automatically

Customize the backup script

The script that you will use can be found *here* .

Just copy the script to a text file and name the file with a .pl extension. You can use any text editor to update the file.

Change the following to reflect your directory structure:

```
# 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';
```

Change the following to reflect your database parameters:

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

Change the following to reflect your off-site FTP server parameters:

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

Save the file with the `.pl` extension (for the purposes of this article we will name the file: `elgg-ftp-backup-script.pl`) and upload it to the following directory `/home/userx/sitebackups`

Be aware that you can turn off FTP and flip a bit in the script so that it does not delete the local backup file in the event that you don't want to use off-site storage for your backups.

Configure the backup Cron job

Login to your CPanel application and click on the "Cron Jobs" link. In the Common Settings dropdown choose "Once a day" and type the following in the command field `/usr/bin/perl /home/userx/sitebackups/elgg-ftp-backup-script.pl`

Click on the "Add New Cron Job" button. Daily full backups are now scheduled and will be transferred off-site.

Configure the cleanup Cron job

If you are sending your backups, via FTP, to another shared hosting provider that uses the CPanel application or you've turned off FTP altogether you can configure your data retention as follows.

Login to your CPanel application for your FTP site, or locally if you're not using FTP, and click on the "Cron Jobs" link. In the Common Settings dropdown choose "Once a day" and type the following in the command field `find /home/usery/sitebackups/full_* -mtime +4 -exec rm {} \;`

The `-mtime X` parameter will set the number of days to retain backups. All files older than `x` number of days will be deleted. Click on the "Add New Cron Job" button. You have now configured your backup retention time.

Restoring from backup

Prepare your backup files

The assumption is that you're restoring your site to another shared hosting provider with CPanel.

When the script backed the files up the original directory structure was maintained in the zip file. We need to do a little cleanup. Perform the following:

- Download the backup file that you wish to restore from
- Extract the contents of the backup file
- **Drill down and you will find your site backup and SQL backup. Extract both of these. You will then have:**
  - a MySQL dump file with a `.sql` extension
  - **another directory structure with the contents of:**
    * `/home/userx/public_html`
    * `/home/userx/elggdata`
- **Repackage the contents of the `/home/userx/public_html` directory as a zip file so that the files are in the root of the**
  - The reason for doing this is simple. It's much more efficient to upload one zip file than it is to ftp the contents of the `/home/userx/public_html` directory to your new host.
- Repackage the contents of the /home/userx/elggdata directory as a zip file so that the files are in the root of the zip file

You should now have the following files:

- the `.sql` file
- the zip file with the contents of `/home/userx/public_html` in the root
- the zip file with the contents of `/home/userx/elggdata` in the root

Restore the files

This is written with the assumption that you're restoring to a different host but maintaining the original directory structure. Perform the following:

- Login to the CPanel application on the host that you wish to restore the site to and open the File Manager.
- **Navigate to `/home/usery/public_html`**

  - Upload the zip file that contains the `/home/userx/public_html` files
  - **Extract the zip file** You should now see all of the files in `/home/usery/public_html`
  - Delete the zip file
- **Navigate to `/home/usery/elggdata`**
  - Upload the zip file that contains the `/home/userx/elggdata` files
  - **Extract the zip file** You should now see all of the files in /home/usery/elggdata
  - Delete the zip file

Program and data file restoration is complete

Restore the MySQL Database

Note: Again, the assumption here is that you're restoring your Elgg installation to a second shared hosting provider. Each shared hosting provider prepends the account holder's name to the databases associated with that account. For example, the username for our primary host is `userx` so the host will prepend `userx_` to give us a database name of `userx_elgg`. When we restore to our second shared hosting provider we're doing so with a username of `usery` so our database name will be `usery_elgg`. The hosting providers don't allow you to modify this behavior. So the process here isn't as simple as just restoring the database from backup to the usery account. However, having said that, it's not terribly difficult either.

Edit the MySQL backup

Open the `.sql` file that you extracted from your backup in your favorite text editor. Comment out the following lines with a hash mark:

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

Save the file.

Create the new database

Perform the following:

- **Login to the CPanel application on the new host and click on the "MySQL Databases" icon**
  - Fill in the database name and click the "create" button. For our example we are going to stick with `elgg` which will give us a database name of `usery_elgg`
  - **You can associate an existing user with the new database, but to create a new user you will need to:**
    - Go to the "Add New User" section of the "MySQL Databases" page
    - Enter the username and password. For our example we're going to keep it simple and use `elgg` once again. This will give us a username of `usery_elgg`
  - **Associate the new user with the new database**

* Go to the "Add User To Database" section of the "MySQL Databases" page. Add the `usery_elgg` user to the `usery_elgg` database
* Select "All Privileges" and click the "Make Changes" button

Restore the production database

Now it's time to restore the MySQL backup file by importing it into our new database named "usery_elgg".

- **Login to the CPanel application on the new host and click on the "phpMyAdmin icon**
  - Choose the `usery_elgg` database in the left hand column
  - Click on the "import" tab at the top of the page
  - Browse to the `.sql` backup on your local computer and select it
  - Click the "Go" button on the bottom right side of the page

You should now see a message stating that the operation was successful

Bringing it all together

The restored elgg installation knows **nothing** about the new database name, database username, directory structure, etc. That's what we're going to address here.

Edit `/public_html/elgg-config/settings.php` on the new hosting provider to reflect the database information for the database that you just created.

```
// 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';

// (For most configurations, you can leave this as 3306)
$CONFIG->dbport = 3306;

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

Upload the `settings.php` file back to the new host - overwriting the existing file.

Open the phpMyAdmin tool on the new host from the CPanel. Select the `usery_elgg` database on the left and click the SQL tab on the top of the page. Run the following SQL queries against the `usery_elgg` database:

Change the installation path

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

Change the data directory

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

Change the filestore data directory

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

Finalizing the new installation

Run the upgrade script by visiting the following URL: `http://mynewdomain.com/upgrade.php` . Do this step twice - back to back.

Update your DNS records so that your host name resolves to the new host's IP address if this is a permanent move.

Congratulations!

If you followed the steps outlined here you should now have a fully functional copy of your primary Elgg installation.

Related

FTP backup script

Here is an automated script for backing up an Elgg installation.

```
#!/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";
```

(continues on next page)

(continued from previous page)

```
# ENTER DATABASE NAME
$database_names_elgg = 'userx_elgg';

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

(continues on next page)

(continued from previous page)

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

Duplicate Installation

Contents

Introduction

Why Duplicate an Elgg Installation?

There are many reasons you may want to duplicate an Elgg installation: moving the site to another server, creating a test or development server, and creating functional backups are the most common. To create a successful duplicate of an Elgg site, 3 things need to be copied:

- Database
- Data from the data directory
- Code

Also at least 5 pieces of information must be changed from the copied installation:

- `elgg-config/settings.php` file which could also be in the pre 2.0 location `engine/settings.php`
- `.htaccess` file (Apache) or Nginx configuration depending on server used
- database entry for your site entity
- database entry for the installation path
- database entry for the data path

What Is Not Covered in This Tutorial

This tutorial expects a basic knowledge of Apache, MySQL, and Linux commands. As such, a few things will not be covered in this tutorial. These include:

- How to backup and restore MySQL databases
- How to configure Apache to work with Elgg
- How to transfer files to and from your production server

Before You Start

Before you start, make sure the Elgg installation you want to duplicate is fully functional. You will also need the following items:

- A backup of the live Elgg database
- A place to copy the live database
- **A server suitable for installing duplicate Elgg site** (This can be the same server as your production Elgg installation.)

Backups of the database can be obtained various ways, including phpMyAdmin, the MySQL official GUI, and the command line. Talk to your host for information on how to backup and restore databases or use Google to find information on this.

During this tutorial, we will make these assumptions about the production Elgg site:

- The URL is `http://www.myelgg.org/`
- The installation path is `/var/www/elgg/`
- The data directory is `/var/data/elgg/`
- The database host is `localhost`
- The database name is `production_elgg`
- The database user is `db_user`
- The database password is `db_password`
- The database prefix is `elgg`

At the end of the tutorial, our test Elgg installation details will be:

- The URL is `http://test.myelgg.org/`
- The installation path is `/var/www/elgg_test/`
- The data directory is `/var/data/elgg_test/`
- The database host is `localhost`

- The database name is `test_elgg`
- The database user is `db_user`
- The database password is `db_password`
- The database prefix is `elgg`

Copy Elgg Code to the Test Server

The very first step is to duplicate the production Elgg code. In our example, this is as simple as copying `/var/www/elgg/` to `/var/www/elgg_test/`.

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

Copy Data to the Test Server

In this example, this is as simple as copying `/var/data/elgg/` to `/var/data/elgg_test/`.

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

If you don't have shell access to your server and have to ftp the data, you may need to change ownership and permissions on the files.

Note: You also need to delete cache directories from your disk. These correspond to `cacheroot` and `assetroot` directories in your config.

Edit settings.php

The `elgg-config/settings.php` file contains the database configuration details. These need to be adjusted for your new test Elgg installation. In our example, we'll look in `/var/www/elgg_test/elgg-config/settings.php` and find the lines that look like this:

```
// 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';
// (For most configurations, you can leave this as 3306)
$CONFIG->dbport = 3306;

// 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';
```

We need to change these lines to match our new 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';
// (For most configurations, you can leave this as 3306)
$CONFIG->dbport = 3306;

// 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: Notice the `$CONFIG->dbname` has changed to reflect our new database.

Copy Elgg Database

Now the database must be copied from `elgg_production` to `elgg_test`. See your favorite MySQL manager's documentation for how to make a duplicate database. You will generally export the current database tables to a file, create the new database, and then import the tables that you previously exported.

You have two options on updating the values in the database. You could change the values in the export file or you could import the file and change the values with database queries. One advantage of modifying the dump file is that you can also change links that people have created to content within your site. For example, if people have bookmarked pages using the bookmark plugin, the bookmarks will point to the old site unless your update their URLs.

Database Entries

We must now change 4 entries in the database. This is easily accomplished with 4 simple SQL commands:

Change the installation path

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

Change the data directory

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

Check .htaccess

If you have made changes to .htaccess that modify any paths, make sure you update them in the test installation.

Update Webserver Config

For this example, you must edit the Apache config to enable a subdomain with a document root of `/var/www/elgg_test/`. If you plan to install into a subdirectory of your document root, this step is unnecessary.

If you're using Nginx, you need to update server config to match new paths based on `install/config/nginx.dist`.

Run upgrade.php

To regenerate cached data, make sure to run `http://test.myelgg.org/upgrade.php`

Tips

It is a good idea to keep a test server around to experiment with installing new mods and doing development work. If you automate restorations to the `elgg_test` database, changing the `$CONFIG` values and adding the follow lines to the end of the `elgg_test/elgg-config/settings.php` file will allow seamless re-writing of the MySQL database entries.

```
$con = mysql_connect($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();
```

Related

See also:

Backup and Restore

3.2.8 Getting Help

Having a problem with Elgg? The best way to get help is to ask at the Community Site. This site is community supported by a large group of volunteers. Here are a few tips to help you get the help you need.

Contents

Getting help

Don't be a Help Vampire

We were all newbies at one time, but we can all learn. Not showing that you are making attempts to learn on your own or do your own research is off putting for those helping. Also, very generic questions like "How do I build a forum?" are almost impossible to answer.

Search first

Be sure to search the documentation (this site), the Community Site, and Google before asking a question. New users to Elgg frequently have the same questions, so please search. People are less inclined to reply to a post that has been answered many other times or that can be answered easily by Googling.

Ask once

Posting the same questions in multiple places makes it hard to answer you. Ask your question in one place only. Duplicate questions may be moderated.

Include Elgg Version

Different versions of Elgg have different features (and different bugs). Including the version of Elgg that you are using will help those helping you.

Have a reasonable profile

Profiles that look like spam or have silly names will often be ignored. Joviality is fine, but people are more likely to help Michael than 1337elggHax0r.

Post in the appropriate forum

Check to make sure you're posting in the right forum. If you have a question about creating a plugin, don't post to the Elgg Feedback forum. If you need help installing Elgg, post to Technical Support instead of the Theming group.

Use a descriptive topic title

Good topic titles concisely describe your problem or question. Bad topic titles are vague, contain all capital letters, and excessive punctuation.

Good title: "White screen after upgrading to 1.7.4."

Bad title: "URGENT!!!!! site broke ;-(losing money help!!!!!!!!!!"

Be detailed

Include as many details about your problem as possible. If you have a live site, include a link. Be forthcoming if community members might ask for more information. We can't help you if you won't give any details!

Keep it public

This is a public forum for the good of the Elgg project. Keep posts public. There's no reason for anyone to ask you to send a private message or email. Likewise, there's no reason to ask anyone to send a private email to you. Post in the public.

Guidelines

In addition to the site-wide Terms and Policies, following these guidelines keeps our community site useful and safe for everyone.

Content

All content must be safe for work: PG in the US and UK. If your Elgg site has adult content and you have been asked to post a link, please mark it NSFW (Not Safe For Work) so people know.

Excessive swearing in any language will not be tolerated.

Mood

Working with technical problems can be frustrating. Please keep the community site free of frustration. If you're feeling anxious, take a step away and do something else. Threatening or attacking community members, core developers, or plugin developers will not help solve your problem and will likely get you banned.

Advertising

Advertising is not allowed. Posts with any sort of advertising will be moderated.

Asking for money / Offering to pay

Don't ask for money on the community site. Likewise, don't offer to pay for answers. If you are looking for custom development, post to the Professional Services group. Posts asking for money or recommending a commercial plugin may be moderated.

Links

If you're having a problem with a live site, please provide a link to it.

That said, the community site is not a back linking service or SEO tool. Excessive linking will be moderated and your account may be banned.

Signatures

There's a reason Elgg doesn't have an option for signatures: they cause clutter and distract from the conversation. Users are discouraged from using signatures on the community site, and signatures with links or advertising will be removed.

Bumping, +1, me too

Don't do it. If your question hasn't been answered, see the top of this document for tips. These types of post add nothing to the conversation and may be moderated.

Posting Code

Long bits of code are confusing to read through in a forums context. Please use http://elgg.pastebin.com to post long bits of code and provide the Paste Bin link instead of directly posting the code.

Good Ideas

Not policies, but good ideas.

Say thanks

Did someone help you? Be sure to thank them! The community site is run by volunteers. No one has to help you with your problem. Be sure to show your appreciation!

Give back

Have a tip for Elgg? See someone with a similar problem you had? You've been there and can help them out, so give them a hand!

3.2.9 Security

As of Elgg 3.0 several hardening settings have been added to Elgg. You can enable/disable these settings as you like.

Contents

Upgrade protection

The URL of http://your-elgg-site.com/upgrade.php can be protected by a unique token. This will prevent random users from being able to run this file. The token is not needed for logged in site administrators.

Cron protection

The URLs of the *cron* can be protected by a unique token. This will prevent random users from being able to run the cron. The token is not needed when running the cron from the commandline of the server.

Disable password autocomplete

Data entered in these fields will be cached by the browser. An attacker who can access the victim's browser could steal this information. This is especially important if the application is commonly used in shared computers such as cyber cafes or airport terminals. If you disable this, password management tools can no longer autofill these fields. The support for the autocomplete attribute can be browser specific.

Email address change requires password

When a user wishes to change their email address associated with their account, they need to also supply their current password.

Email address change requires confirmation

When a user wishes to change their email address associated with their account, they need to confirm the new email address. This is done by sending an email to the new address with a validation link. After clicking this link the new email address will be used.

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 to site administrators

When a new site administrator is added or when a site administrator is removed all the site administrators get a notification about this action.

Notifications to user

Site administrator

When the site administrator role is added to or removed from the account, send a notification to the user whos account this is affecting.

(Un)ban

When the account of a user gets banned or unbanned, let the affected user know about this action.

Minimal username length

You can configure the minimal length the username should have upon registration of a user.

Minimal password requirements

You can configure several requirements for new passwords of the users

- **length**: the password should be at least x characters long
- **lower case**: minimal number of lower case (a-z) characters in the password
- **upper case**: minimal number of upper case (A-Z) characters in the password
- **numbers**: minimal number of numbers (0-9) characters in the password
- **specials**: minimal number of special (like !@#$%^&*(), etc.) characters in the password

3.2.10 User validation

Plugins can influence how users are validated before they can use the website.

Contents

Listing of unvalidated users

In the Admin section of the website is a list of all unvalidated users. Some actions can be taken on the users, like delete them from the system or validate them.

Plugins have the option to add additional features to this list.

See also:

An example of this is the *User validation by e-mail* plugin which doesn't allow users onto the website until their e-mail address is validated.

Require admin validation

In the Site settings under the Users section there is a setting which can be enabled to require admin validation of a new user account before the user can use their account. After registration the user gets notified that their account is awaiting validation by an administrator.

Site administrators can receive an e-mail notification that there are users awaiting validation.

After validation the user is notified that they can use their account.

3.2.11 Spam

Keep spam under control.

Spam is a very common problem that admins need to deal with and which we aim to improve in core, but there are some actions site admins can take to mitigate the problem.

Install an anti-spam plugin

There are several available on the community

http://community.elgg.org/plugins/search?f{[}c{]}{[}{]}=spam&sb=Search

Change the registration url

Some have said they have good success changing the registration URL, since the spammers are naive bots and can no longer find where to create the fake accounts.

Disable open registration

If the flow of registrations is low enough, you can vet all users that come in to keep the quality of interaction high.

Contribute to anti-spam measures in core

https://github.com/Elgg/Elgg/issues?labels=spam&state=open

3.3 Developer Guides

Customize Elgg's behavior with plugins.

3.3.1 Don't Modify Core

Warning: 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

See also:

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');
```

(continues on next page)

(continued from previous page)

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

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

(continues on next page)

(continued from previous page)

```
// 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).

Ignoring access

If for some case you need entities retrieved ignoring the access rules you can wrap your code in `elgg_call`. There are different flags you can use.

- ELGG_IGNORE_ACCESS: no access rules are applied
- ELGG_ENFORCE_ACCESS: access rules are forced to be applied
- ELGG_SHOW_DISABLED_ENTITIES: will retrieve entities that are disabled
- ELGG_HIDE_DISABLED_ENTITIES: will never retrieve entities that are disabled

```
$options = [
        'type' => 'user'
];

$entities = elgg_call(ELGG_IGNORE_ACCESS, function() use ($options) {
        return elgg_get_entities($options);
});
```

You can also combine flags.

```
$entities = elgg_call(ELGG_IGNORE_ACCESS | ELGG_SHOW_DISABLED_ENTITIES, function() {
        return elgg_get_entities([
                'type' => 'user'
        ]);
});
```

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' => [
```

(continues on next page)

(continued from previous page)

```
            '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,
        ],
    ],
];
```

Warning: 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',
            ],
        ],
];
```

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:validate', $action, null, true);
```

Where `$action` is the action being called. If the hook returns `false` then the action will not be executed. Don't return anything if your validation passes.

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 `false` if the captcha validation fails (which prevents the associated action from executing).

This is done as follows:

```
elgg_register_plugin_hook_handler("action:validate", "register", "captcha_verify_
↪action_hook");
elgg_register_plugin_hook_handler("action:validate", "user/requestnewpassword",
↪"captcha_verify_action_hook");

...

function captcha_verify_action_hook(\Elgg\Hook $hook) {
  $token = get_input('captcha_token');
  $input = get_input('captcha_input');

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

  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' => elgg_generate_action_url('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 [
   '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' => [
      'draft' => elgg_echo('status:draft'),
      'published' => elgg_echo('status:published'),
   ],
   // most input views will render additional parameters passed to the view
   // as tag attributes
```

(continues on next page)

(continued from previous page)

```
    '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([
    '#type' => 'select',
    '#label' => elgg_echo('blog:status:label'),
    '#help' => elgg_view_icon('help') . elgg_echo('blog:status:help'),
    'required' => true,
    'name' => 'status',
    'options_values' => [
        '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 `<button 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

Elgg offers some helper input types

- `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', ['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($name)`.

Overview

The basic flow of using sticky forms is:

1. Call `elgg_make_sticky_form($name)` at the top of actions for forms you want to be sticky.
2. Use `elgg_is_sticky_form($name)` and `elgg_get_sticky_values($name)` to get sticky values when rendering a form view.

3. 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.

```
// 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");
$url = elgg_generate_action_url('myplugin/myaction');
```

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,
```

(continues on next page)

(continued from previous page)

```
    'b' => $b,
    'mac' => elgg_build_hmac([$a, $b])->getToken(),
]);
$url = "action/foo?$query";

// 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");
```

Warning: 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 */

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.

Warning: 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) {
        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`.

Warning: 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();

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!';
```

(continues on next page)

(continued from previous page)

```
    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

    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

Warning: 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.

Warning: 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.

Warning:

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`.

Warning:

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);
    }
});
```

`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

Warning: 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(\Elgg\Hook $hook) {
    $start_time = $hook->getParam('time');
}
```

Custom intervals

Plugin developers can configure there own custom intervals.

> **Warning:** It's **NOT** recommended to do this, as the users of your plugin may also need to configure your custom interval. Try to work with the default intervals. If you only need to do a certain task at for example 16:30 you can use the `halfhour` interval and check that `date('G', $start_time) == 16` and `date('i', $start_time) == 30`

```
elgg_register_plugin_hook_hander('cron:intervals', 'system', 'my_custom_cron_interval
↪');

function my_custom_cron_interval(\Elgg\Hook $hook) {
        $cron_intervals = $hook->getValue();

        // add custom interval
        $cron_intervals['my_custom_interval'] = '30 16 * * *'; // every day at 16:30␣
↪hours

        return $cron_intervals;
}
```

See also:

- *Events and Plugin Hooks* has more information about hooks
- For more information about the supported cron interval definition see the PHP Scheduler documentation

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');
```

(continues on next page)

(continued from previous page)

```
// 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.

```
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',
                'no-reply@mysite.com',
                \Monolog\Logger::CRITICAL
        )
);
```

3.3.11 List of events in core

For more information on how events work visit *Events and Plugin Hooks*.

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();

// 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();
```

(continues on next page)

(continued from previous page)

```
// 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',
```

(continues on next page)

(continued from previous page)

```
];

$mime_type = ElggFile::detectMimeType($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) {
        $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.

Warning: 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>`. This view will only be called if the tool is enabled.

If you simply wish to list some content in the group you can use the `groups/profile/module` view with some additional parameters.

- `entity_type`: in combination with the `entity_subtype` it can generate everything the module needs
- `entity_subtype`: in combination with the `entity_type` it can generate everything the module needs
- `no_results`: custom no results found text

The following will be automaticly generated:

- `title`: based on the the language key `collection:<entity_type>:<entity_subtype>:group`
- `content`: `elgg_list_entities()` based on given type/subtype
- `all_link`: based on the route name `collection:<entity_type>:<entity_subtype>:group`
- `add_link`: based on the route name `add:<entity_type>:<entity_subtype>:group` and with a permissions check to the given type/subtype

```
// file: groups/profile/module/my-tool.php

// if you wish to list some content (eg. files) in the group
// you can use the following
$params = [
        'entity_type' => 'object',
        'entity_subtype' => 'file',
        'no_results' => elgg_echo('file:none'),
];
$params = $params + $vars;

echo elgg_view('groups/profile/module', $params);
```

Alternatively you can generate your own title and content

```
// 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);
```

See also:

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.

See also:

Be sure to follow the *Plugin skeleton* for your plugin's layout.

Warning: *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 *Events and Plugin Hooks*.

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.

cron:intervals, system Allow the configuration of custom cron intervals

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.

generate, password Allows plugins to generate new random cleartext passwords.

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.

> **Warning:** 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.

> **Warning:** 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:subtypes, user Returns an array of access collection subtypes to be used when retrieving access collections owned by a user as part of the `get_write_access_array()` function.

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'];
```

(continues on next page)

(continued from previous page)

```
        }

        // 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,
```

(continues on next page)

(continued from previous page)

```
        '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
<?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
<?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:

```php
echo elgg_echo('example:text');
```

It also supports variable replacement using `vsprintf` syntax:

```php
// '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:

```php
echo elgg_echo('welcome', [], $user->language);
```

To first test whether `elgg_echo()` can find a translation:

```php
$key = 'key:that:might:not:exist';
if (!elgg_language_key_exists($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`.

> **Warning:** 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) {
    // 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
```

(continues on next page)

(continued from previous page)

```
    // ...
});
```

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` section of the `elgg-plugin.php` file in the root of your plugin:

```
<?php // elgg-plugin.php
return [
    'views' => [
            '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 // elgg-plugin.php
return [
    'views' => [
            '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.

Warning: 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_current_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.

Warning: 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) {
           // ...
       }
   });
});
```

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, {
            '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);
```

(continues on next page)

(continued from previous page)

```
        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',
```

(continues on next page)

(continued from previous page)

```
    '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() {
                                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, and there is no Elgg API for loading the scripts, you can register scripts in a hook handler to add elements to the head links;

```
elgg_register_plugin_hook_handler('head', 'page', $callback);
```

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) {
        // 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

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 three hooks that can be used to modify a menu:

- `'parameters', 'menu:<menu name>'` to add or modify parameters use for the menu building (eg. sorting)
- `'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'];

        // 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',
                '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'),
]);

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);
```

(continues on next page)

(continued from previous page)

```
    // 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
 *
```

(continues on next page)

(continued from previous page)

```
 * @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;
        }

        // (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
```

(continues on next page)

(continued from previous page)

```
 */
function discussion_get_subscriptions($hook, $type, $subscriptions, $params) {
        $reply = $params['event']->getObject();

        if (!$reply instanceof \ElggDiscussionReply) {
                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

Warning: 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;
   }

   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();
}

/**
```

(continues on next page)

(continued from previous page)

```
 * 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()`
- `events` - eliminates the need for calling `elgg_register_event_handler()`
- `hooks` - eliminates the need for calling `elgg_register_plugin_hook_handler()`

```
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
                '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,
```

(continues on next page)

(continued from previous page)

```
                ],

                // 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',
        ],

        'views' => [
                'default' => [
                        'cool_lib/' => __DIR__ . '/vendors/cool_lib/dist/',
                ],
        ],

        'hooks' => [
                'register' => [
                        'menu:owner_block' => [
                                'blog_owner_block_menu' => [
                                        'priority' => 700,
                                ],
                        ],
                ],
                'likes:is_likable' => [
                        'object:blog' => [
                                'Elgg\Values::getTrue' => [],
                        ],
                ],
                'usersettings:save' => [
                        'user' => [
                                '_elgg_save_notification_user_settings' => [
↪'unregister' => true],
                        ],
                ],
        ],

        'events' => [
                'delete' => [
                        'object' => [
                                'file_handle_object_delete' => [
                                        'priority' => 999,
                                ],
```

(continues on next page)

(continued from previous page)

```
                        ],
                ],
                'create' => [
                        'relationship' => [
                                '_elgg_send_friend_notification' => [],
                        ],
                ],
                'log' => [
                        'systemlog' => [
                                'system_log_default_logger' => ['unregister' => true],
                        ],
                ],
        ],
];
```

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.

See also:

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',
                ],
        ],
];
```

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.
- 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.

See also:

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 no dependencies, or additional information about the plugin.</description>

        <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 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>

        <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": {
```

(continues on next page)

(continued from previous page)

```
                "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.

See also:

Writing 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/
        PluginNamespace/
            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
                owner.css
                owner.js
                owner.php
        widgets/
            example_widget/
                content.php
                edit.php
elgg-plugin.php
CHANGES.txt
COPYRIGHT.txt
INSTALL.txt
LICENSE.txt
manifest.xml
README.txt
start.php
```

(continues on next page)

(continued from previous page)

```
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.

When organizing you classes Elgg does not require a specific structure. Use what works best for your plugin but keep in mind that it should be easy to read, funtionality should be easy to find and having seperated functions into different classes will improve maintainability and testability.

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*.

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.

> **Warning:** 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+',
        ],
]);
```

(continues on next page)

(continued from previous page)

```
// 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
];
```

Plugin dependent routes

If a route requires a specific plugin to be active this can be configured in the route configuration.

```
// elgg-plugin.php
return [
        'routes' => [
                'collection:object:blog:friends' => [
                        'path' => '/blog/friends/{username?}/{lower?}/{upper?}',
                        'resource' => 'blog/friends',
                        'required_plugins' => [
                                'friends', // route only allowed when friends plugin␣
↪is active
                        ],
                ],
        ]
];
```

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. The following middleware can be found in the namespace `\Elgg\Router\Middleware`:

Gatekeeper

This gatekeeper will prevent access by non-authenticated users.

AdminGatekeeper

This gatekeeper will prevent access by non-admin users.

LoggedOutGatekeeper

This gatekeeper will prevent access by authenticated users.

AjaxGatekeeper

This gatekeeper will prevent access with non-xhr requests.

PageOwnerCanEditGatekeeper

This gatekeeper will prevent access if there is a pageowner detected and the pageowner can't be editted.

GroupPageOwnerCanEditGatekeeper

This gatekeeper extends the `PageOwnerCanEditGatekeeper` but also requires the pageowner to be a `ElggGroup` entity.

UserPageOwnerCanEditGatekeeper

This gatekeeper extends the `PageOwnerCanEditGatekeeper` but also requires the pageowner to be an `ElggUser` entity.

CsrfFirewall

This middleware will prevent access without the correct CSRF tokens. This middleware will automatically be applied to actions.

ActionMiddleware

This middleware will provide action related logic. This middleware will automatically be applied to actions.

SignedRequestGatekeeper

This gatekeeper will prevent access if the url has been tampered with. A secure URL can be generated using the `elgg_http_get_signed_url` function.

UpgradeGatekeeper

This gatekeeper will prevent access if the upgrade URL is secured and the URL is invalid.

WalledGarden

This middleware will prevent access to a route if the site is configured for authenticated users only and there is no authenticated user logged in. This middleware is automatically enabled for all routes. You can disable the walled garden gatekeeper with a *route config* option.

Custom Middleware

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) {
```

(continues on next page)

(continued from previous page)

```
                        // 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?}',
        '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
↪');
```

Warning: 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']);
    }

    $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;
    }

    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'),
        ]);
    };

    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:

```php
// 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',
    '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));
```

(continues on next page)

(continued from previous page)

```
        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);
```

Warning: 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

- *Customizing the front page*

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 */
.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 */
```

(continues on next page)

(continued from previous page)

```
.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 {
        order: 1;
        margin-right: auto;
}
.heading > .controls {
        order: 2;
}
```

Symmetrical We maintain symmetry.

```
/* BAD */
```

(continues on next page)

(continued from previous page)

```
.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 */
<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>
```

(continues on next page)

(continued from previous page)

```
/* 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 {

        /**
         * 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;
        }
```

(continues on next page)

(continued from previous page)

```
        /**
         * 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
        }
}
```

Warning: 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.

> **Warning:** 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
```

(continues on next page)

(continued from previous page)

```
     * @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>
```

Warning: 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.

Warning: 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

On the page you want to load the css, call:

```
elgg_require_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',

    '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, [
            '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

See also:

- *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

- *Determining the authentication available*
- *Related*

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',
                        '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. As of Elgg 3.2 API keys can be generated by the webservices plugin. 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
```

(continues on next page)

(continued from previous page)

```
    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.

API results

Contents

Success result structure

A successful API result looks like this:

```
{
        "status": 0,
        "result": "API result"
}
```

Depending on the API call `result` can contain any type of content (string, number, array, object, etc.).

An example of a numberic result (for example a user count):

```
{
        "status": 0,
        "result": 10
}
```

An example of an object result (for example a user):

```
{
        "status": 0,
        "result": {
                "name": "Some user",
                "username": "apiexample",
                "email": "user@example.com"
        }
}
```

Error result structure

When an API call fails the result will look like this:

```
{
        "status": -1,
        "message": "The reason the API call failed"
}
```

Default status codes

The `status` field always contains a number representing the result. Any value other than `0` is considered an error.

- `0`: This is a success result
- `-1`: This is a generic error result
- `-20`: The user authentication token is missing, is invalid or has expired
- `-30`: The api key has been disabled
- `-31`: The api key is inactive
- `-32`: The api key is invalid

Developers can implement their own status codes to represent different error states, so the request doesn't have to rely on the error message to know what went wrong.

Note: `result` and `message` can contain messages in different languages. This is depending on the user language when using user authenticated API calls or the site langauge for other API calls. Keep in mind that the language can change, eighter by the user or by a site administrator for the site language.

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
```

(continues on next page)

(continued from previous page)

```
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 Tutorials

Walk through all the required steps in order to customize Elgg.

The instructions are detailed enough that you don't need much previous experience with Elgg.

3.4.1 Hello world

This tutorial shows you how to create a new plugin that consists of a new page with the text "Hello world" on it.

Before anything else, you need to *install Elgg*.

In this tutorial we will pretend your site's URL is `https://elgg.example.com`.

First, create a directory that will contain the plugin's files. It should be located under the `mod/` directory which is located in your Elgg installation directory. So in this case, create `mod/hello/`.

Manifest file

Elgg requires that your plugin has a manifest file that contains information about the plugin. Therefore, in the directory you just created, create a file called `manifest.xml` and copy this code into it:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin_manifest xmlns="http://www.elgg.org/plugin_manifest/1.8">
    <name>Hello world</name>
```

(continues on next page)

(continued from previous page)

```
    <id>hello</id>
    <author>Your Name Here</author>
    <version>0.1</version>
    <description>Hello world, testing.</description>
    <requires>
        <type>elgg_release</type>
        <version>2.0</version>
    </requires>
</plugin_manifest>
```

This is the minimum amount of information in a manifest file:

- `<name>` is the display name of the plugin
- `<id>` must be the same as the directory you just created
- `<requires>` must include which version of Elgg your plugin requires
- `<author>`, `<version>` and `<description>` should have some appropriate values but can be filled freely

Initializer

Next, create `start.php` in the `mod/hello/` directory and copy this code into it:

```
<?php

elgg_register_event_handler('init', 'system', 'hello_world_init');

function hello_world_init() {

}
```

The above code tells Elgg that it should call the function `hello_world_init()` once the Elgg core system is initiated.

Registering a route

The next step is to register a route which has the purpose of handling request that users make to the URL `https://elgg.example.com/hello`.

Update `elgg-plugin.php` to look like this:

```
<?php

return [
    'routes' => [
                'default:hello' => [
                        'path' => '/hello',
                        'resource' => 'hello',
                ],
        ],
];
```

This registration tells Elgg that it should call the resource view `hello` when a user navigates to `https://elgg.example.com/hello`.

View file

Create `mod/hello/views/default/resources/hello.php` with this content:

```
<?php

$body = elgg_view_layout('content', [
    'title' => 'Hello world!',
    'content' => 'My first page!',
    'filter' => '',
]);

echo elgg_view_page('Hello', $body);
```

The code creates an array of parameters to be given to the `elgg_view_layout()` function, including:

- The title of the page
- The contents of the page
- Filter which is left empty because there's currently nothing to filter

This creates the basic layout for the page. The layout is then run through `elgg_view_page()` which assembles and outputs the full page.

Last step

Finally, activate the plugin through your Elgg administrator page: `https://elgg.example.com/admin/plugins` (the new plugin appears at the bottom).

You can now go to the address `https://elgg.example.com/hello/` and you should see your new page!

3.4.2 Customizing the Home Page

To override the homepage, just override Elgg's `resources/index` view by creating a file at `/views/default/resources/index.php`.

Any output from this view will become your new homepage.

You can take a similar approach with any other page in Elgg or official plugins.

3.4.3 Building a Blog Plugin

This tutorial will teach you how to create a simple blog plugin. The basic functions of the blog will be creating posts, saving them and viewing them. The plugin duplicates features that are found in the bundled `blog` plugin. You can disable the bundled `blog` plugin if you wish, but it is not necessary since the features do not conflict each other.

Contents

Prerequisites:

- *Install Elgg*

Create the plugin's directory and manifest file

First, choose a simple and descriptive name for your plugin. In this tutorial, the name will be `my_blog`. Then, create a directory for your plugin in the `/mod/` directory found in your Elgg installation directory. Other plugins are also located in `/mod/`. In this case, the name of the directory should be `/mod/my_blog/`. This directory is the root of your plugin and all the files that you create for the new plugin will go somewhere under it.

Next, in the root of the plugin, create the plugin's manifest file, `manifest.xml`:

```
<?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>
```

See *Plugins* for more information about the manifest file.

Create the form for creating a new blog post

Create a file at `/mod/my_blog/views/default/forms/my_blog/save.php` that contains the form body. The form should have input fields for the title, body and tags of the my_blog post. It does not need form tag markup.

```
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',
```

(continues on next page)

(continued from previous page)

```
    'required' => true,
]);

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);
```

Notice how the form is calling `elgg_view_field()` to render inputs. This helper function maintains consistency in field markup, and is used as a shortcut for rendering field elements, such as label, help text, and input. See *Forms + Actions*.

You can see a complete list of input views in the `/vendor/elgg/elgg/views/default/input/` directory.

It is recommended that you make your plugin translatable by using `elgg_echo()` whenever there is a string of text that will be shown to the user. Read more at *Internationalization*.

Create a page for composing the blogs

Create the file `/mod/my_blog/views/default/resources/my_blog/add.php`. This page will view the form you created in the above section.

```
<?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);
```

The function `elgg_view_form("my_blog/save")` views the form that you created in the previous section. It also automatically wraps the form with a `<form>` tag and the necessary attributes as well as anti-csrf tokens.

The form's action will be `"<?= elgg_get_site_url() ?>action/my_blog/save"`.

Create the action file for saving the blog post

The action file will save the my_blog post to the database. Create the file `/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
}
```

As you can see in the above code, Elgg objects have several fields built into them. The title of the my_blog post is stored in the `title` field while the body is stored in the `description` field. There is also a field for tags which are stored as metadata.

Objects in Elgg are a subclass of something called an "entity". Users, sites, and groups are also subclasses of entity. An entity's subtype allows granular control for listing and displaying, which is why every entity should have a subtype. In this tutorial, the subtype "`my_blog`" identifies a my_blog post, but any alphanumeric string can be a valid subtype. When picking subtypes, be sure to pick ones that make sense for your plugin.

The `getURL` method fetches the URL of the new post. It is recommended that you override this method. The overriding will be done in the `start.php` file.

Create elgg-plugin.php

The `/mod/my_blog/elgg-plugin.php` file is used to declare various functionalities of the plugin. It can, for example, be used to configure entities, actions, widgets and 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+',
                        ],
                ],
        ],
];
```

Create start.php

The `/mod/my_blog/start.php` file needs to register a hook to override the URL generation.

```
<?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');
}
```

Registering the save action will make it available as `/action/my_blog/save`. By default, all actions are available only to logged in users. If you want to make an action available to only admins or open it up to unauthenticated users, you can pass 'admin' or 'public' as the third parameter of `elgg_register_action`.

The URL overriding function will extract the ID of the given entity and use it to make a simple URL for the page that is supposed to view the entity. In this case the entity should of course be a my_blog post. Add this function to your `start.php` file:

```
function my_blog_set_url($hook, $type, $url, $params) {
    $entity = $params['entity'];
    if ($entity->getSubtype() === 'my_blog') {
        return "my_blog/view/{$entity->guid}";
    }
}
```

The page handler makes it possible to serve the page that generates the form and the page that views the post. The next section will show how to create the page that views the post. Add this function to your `start.php` file:

```
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;
}
```

The `$segments` variable contains the different parts of the URL as separated by /.

Page handling functions need to return `true` or `false`. `true` means the page exists and has been handled by the page handler. `false` means that the page does not exist and the user will be forwarded to the site's 404 page (requested page does not exist or not found). In this particular example, the URL must contain either `/my_blog/add` or `/my_blog/view/id` where id is a valid ID of an entity with the `my_blog` subtype. More information about page handling is at *Page handler*.

Create a page for viewing a blog post

To be able to view a my_blog post on its own page, you need to make a view page. Create the file `/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' => '',
];
```

(continues on next page)

(continued from previous page)

```
$body = elgg_view_layout('content', $params);

echo elgg_view_page($my_blog->getDisplayName(), $body);
```

This page has much in common with the `add.php` page. The biggest differences are that some information is extracted from the my_blog entity, and instead of viewing a form, the function `elgg_view_entity` is called. This function gives the information of the entity to something called the object view.

Create the object view

When `elgg_view_entity` is called or when my_blogs are viewed in a list for example, the object view will generate the appropriate content. Create the file `/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));
```

As you can see in the previous section, each my_blog post is passed to the object view as `$vars['entity']`. (`$vars` is an array used in the views system to pass variables to a view.)

The last line takes the tags on the my_blog post and automatically displays them as a series of clickable links. Search is handled automatically.

(If you're wondering about the "`default`" in `/views/default/`, you can create alternative views. RSS, OpenDD, FOAF, mobile and others are all valid view types.)

Trying it out

Go to your Elgg site's administration page, list the plugins and activate the `my_blog` plugin.

The page to create a new my_blog post should now be accessible at `https://elgg.example.com/my_blog/add`, and after successfully saving the post, you should see it viewed on its own page.

Displaying a list of blog posts

Let's also create a page that lists my_blog entries that have been created.

Create `/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));

echo elgg_view_page($titlebar, $body);
```

The `elgg_list_entities` function grabs the latest my_blog posts and passes them to the object view file. Note that this function returns only the posts that the user can see, so access restrictions are handled transparently. The function (and its cousins) also transparently handles pagination and even creates an RSS feed for your my_blogs if you have defined that view.

The list function can also limit the my_blog posts to those of a specified user. For example, the function `elgg_get_logged_in_user_guid` grabs the Global Unique IDentifier (GUID) of the logged in user, and by giving that to `elgg_list_entities`, the list only displays the posts of the current user:

```
echo elgg_list_entities(array(
    'type' => 'object',
    'subtype' => 'my_blog',
    'owner_guid' => elgg_get_logged_in_user_guid()
));
```

Next, you will need to modify your my_blog page handler to grab the new page when the URL is set to `/my_blog/all`. Change the `my_blog_page_handler` function in `start.php` to look like this:

```
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;
}
```

Now, if the URL contains `/my_blog/all`, the user will see an "All Site My_Blogs" page. Because of the default case, the list of all my_blogs will also be shown if the URL is something invalid, like `/my_blog` or `/my_blog/xyz`.

You might also want to update the object view to handle different kinds of viewing, because otherwise the list of all my_blogs will also show the full content of all my_blogs. Change `/mod/my_blog/views/default/object/my_blog.php` to look like this:

```
<?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));

// list view or short view
} else {
    // make a link out of the post's title
    echo elgg_view_title(
        elgg_view('output/url', array(
```

(continues on next page)

(continued from previous page)

```
            'href' => $vars['entity']->getURL(),
            'text' => $vars['entity']->getDisplayName(),
            'is_trusted' => true,
    )));
    echo elgg_view('output/tags', array('tags' => $vars['entity']->tags));
}
```

Now, if `full_view` is `true` (as it was pre-emptively set to be in *this section*), the object view will show the post's content and tags (the title is shown by `view.php`). Otherwise the object view will render just the title and tags of the post.

The end

There's much more that could be done, but hopefully this gives you a good idea of how to get started.

3.4.4 Integrating a Rich Text Editor

Build your own wysiwyg plugin.

Elgg is bundled with a plugin for CKEditor, and previously shipped with TinyMCE support. However, if you have a wysiwyg that you prefer, you could use this tutorial to help you build your own.

All forms in Elgg should try to use the provided input views located in `views/default/input`. If these views are used, then it is simple for plugin authors to replace a view, in this case `input/longtext`, with their wysiwyg.

Add the WYSIWYG library code

Now you need to upload TinyMCE into a directory in your plugin. We strongly encourage you to use `composer` to manage third-party dependencies, since it is so much easier to upgrade and maintain that way:

```
.. code-block:: sh
```

    composer require bower-asset/tinymce

Tell Elgg when and how to load TinyMCE

Now that you have:

- created your start file
- intialized the plugin
- uploaded the wysiwyg code

It is time to tell Elgg how to apply TinyMCE to longtext fields.

We're going to do that by extending the input/longtext view and including some javascript. Create a view tinymce/longtext and add the following code:

```
<?php

    /**
     * Elgg long text input with the tinymce text editor intacts
     * Displays a long text input field
```

(continues on next page)

(continued from previous page)

```
     *
     * @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>
```

Then, in your plugin's init function, extend the input/longtext view

```
function tinymce_init() {
    elgg_extend_view('input/longtext', 'tinymce/longtext');
}
```

That's it! Now every time someone uses input/longtext, TinyMCE will be loaded and applied to that textarea.

3.4.5 Basic Widget

Create a widget that will display "Hello, World!" and optionally any text the user wants.

In Elgg, widgets are those components that you can drag onto your profile or admin dashboard.

This tutorial assumes you are familiar with basic Elgg concepts such as:

- *Views*
- *Plugins*

You should review those if you get confused along the way.

Contents

Adding the widget view code

Elgg automatically scans particular directories under plugins looking for particular files. *Views* make it easy to add your display code or do other things like override default Elgg behavior. For now, we will just be adding the view code for your widget. Create a file at `/views/default/widgets/helloworld/content.php`. "helloworld" will be the name of your widget within the hello plugin. In this file add the code:

```
<?php

echo "Hello, world!";
```

This will add these words to the widget canvas when it is drawn. Elgg takes care of loading the widget.

Registering your widget

Elgg needs to be told explicitly that the plugin contains a widget so that it will scan the widget views directory. This is done by calling the elgg_register_widget_type() function. Edit `/start.php`. In it add these lines:

```
<?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');
}
```

Now go to your profile page using a web browser and add the "hello, world" widget. It should display "Hello, world!".

Note: For real widgets, it is always a good idea to support *Internationalization*.

Allow user customization

Click on the edit link on the toolbar of the widget that you've created. You will notice that the only control it gives you by default is over access (over who can see the widget).

Suppose you want to allow the user to control what greeting is displayed in the widget. Just as Elgg automatically loads `content.php` when viewing a widget, it loads `edit.php` when a user attempts to edit a widget. Put the following code into `/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>
```

Notice the relationship between the values passed to the 'name' and the 'value' fields of input/text. The name of the input text box is `params[message]` because Elgg will automatically handle widget variables put in the array `params`. The actual php variable name will be `message`. If we wanted to use the field `greeting` instead of `message` we would pass the values `params[greeting]` and `$widget->greeting` respectively.

The reason we set the 'value' option of the array is so that the edit view remembers what the user typed in the previous time he changed the value of his message text.

Now to display the user's message we need to modify content.php to use this *message* variable. Edit `/views/default/widgets/helloworld/content.php` and change it to:

```
<?php

$widget = $vars['entity'];

// Always use the corresponding output/* view for security!
echo elgg_view('output/text', array('value' => $widget->message));
```

You should now be able to enter a message in the text box and see it appear in the widget.

3.5 Design Docs

Gain a deep understanding of how Elgg works and why it's built the way it is.

3.5.1 Actions

Actions are the primary way users interact with an Elgg site.

Overview

An action in Elgg is the code that runs to make changes to the database when a user does something. For example, logging in, posting a comment, and making a blog post are actions. The action script processes input, makes the appropriate modifications to the database, and provides feedback to the user about the action.

Action Handler

Actions are registered during the boot process by calling `elgg_register_action()`. All actions URLs start with `action/` and are served by Elgg's front end controller through the action service. This approach is different

from traditional PHP applications that send information to a specific file. The action service performs *CSRF security checks*, and calls the registered action script file, then optionally forwards the user to a new page. By using the action service instead of a single script file, Elgg automatically provides increased security and extensibility.

See *Forms + Actions* for details on how to register and construct an action. To look at the core actions, check out the directory /actions.

3.5.2 Database

A thorough discussion of Elgg's data model design and motivation.

Contents

Overview

In Elgg, everything runs on a unified data model based on atomic units of data called entities.

Plugins are discouraged from interacting directly with the database, which creates a more stable system and a better user experience because content created by different plugins can be mixed together in consistent ways. With this approach, plugins are faster to develop, and are at the same time much more powerful.

Every entity in the system inherits the `ElggEntity` class. This class controls access permissions, ownership, containment and provides consistent API for accessing and updating entity properties.

You can extend entities with extra information in two ways:

`Metadata`: This information describes the entity, it is usually added by the author of the entity when the entity is created or updated. Examples of metadata include tags, ISBN number or a third-party ID, location, geocoordinates etc. Think of metadata as a simple key-value storage.

`Annotations`: This information extends the entity with properties usually added by a third party. Such properties include ratings, likes, and votes.

The main differences between metadata and annotations:

- metadata does not have owners, while annotations do
- metadata is not access controlled, while annotations are
- metadata is preloaded when entity is constructed, while annotations are only loaded on demand

These differences might have implications for performance and your business logic, so consider carefully, how you would like to attach data to your entities.

In certain cases, it may be benefitial to avoid using metadata and annotations and create new entities instead and attaching them via `container_guid` or a relationship.

Datamodel

Entities

`ElggEntity` is the base class for the Elgg data model and supports a common set of properties and methods.

- A numeric Globally Unique IDentifier (See *GUIDs*).
- Access permissions. (When a plugin requests data, it never gets to touch data that the current user doesn't have permission to see.)
- An arbitrary subtype (more below).
- An owner.

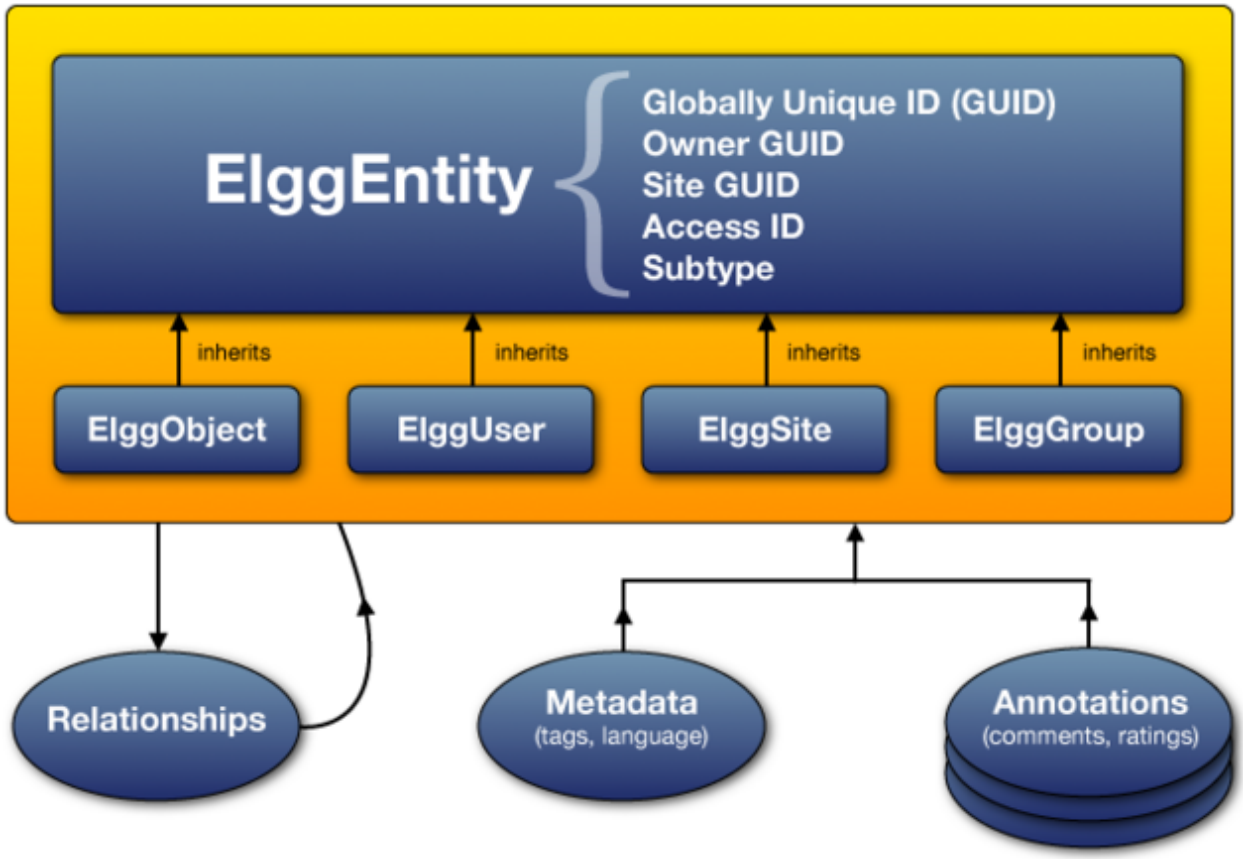

Fig. 9: The Elgg data model diagram

- The site that the entity belongs to.
- A container, used to associate content with a group or a user.

Types

Actual entities will be instances of four different subclasses, each having a distinct **type** property and their own additional properties and methods.

Type	PHP class	Represents
object	`ElggObject`	Most user-created content, like blog posts, uploads, and bookmarks.
group	`ElggGroup`	An organized group of users with its own profile page
user	`ElggUser`	A user of the system
site	`ElggSite`	The site served by the Elgg installation

Each type has its own extended API. E.g. users can be friends with other users, group can have members, while objects can be liked and commented on.

Subtypes

Each entity must define a **subtype**, which plugins use to further specialize the entity. Elgg makes it easy to query specific for entities of a given subtype(s), as well as assign them special behaviors and views.

Subtypes are most commonly given to instances of `ElggEntity` to denote the kind of content created. E.g. the blog plugin creates objects with subtype `"blog"`.

By default, users, groups and sites have the the subtypes of `user`, `group` and `site` respectively.

Plugins can use custom entity classes that extend the base type class. To do so, they need to register their class at runtime (e.g. in the `'init','system'` handler), using `elgg_set_entity_class()`. For example, the blog plugin could use `elgg_set_entity_class('object', 'blog', \ElggBlog::class)`.

Plugins can use `elgg-plugin.php` to define entity class via shortcut `entities` parameter.

Subtype Gotchas

- Before an entity's `save()` method is called, the subtype must be set by writing a string to the `subtype` property.
- *Subtype cannot be changed after saving.*

GUIDs

A GUID is an integer that uniquely identifies every entity in an Elgg installation (a Globally Unique IDentifier). It's assigned automatically when the entity is first saved and can never be changed.

Some Elgg API functions work with GUIDs instead of `ElggEntity` objects.

ElggObject

The `ElggObject` entity type represents arbitrary content within an Elgg install; things like blog posts, uploaded files, etc.

Beyond the standard ElggEntity properties, ElggObjects also support:

- `title` The title of the object (HTML escaped text)
- `description` A description of the object (HTML)

Most other data about the object is generally stored via metadata.

ElggUser

The `ElggUser` entity type represents users within an Elgg install. These will be set to disabled until their accounts have been activated (unless they were created from within the admin panel).

Beyond the standard ElggEntity properties, ElggUsers also support:

- `name` The user's plain text name. e.g. "Hugh Jackman"
- `username` Their login name. E.g. "hjackman"
- `password` A hashed version of their password
- `email` Their email address
- `language` Their default language code.
- `code` Their session code (moved to a separate table in 1.9).
- `last_action` The UNIX timestamp of the last time they loaded a page
- `prev_last_action` The previous value of `last_action`
- `last_login` The UNIX timestamp of their last log in
- `prev_last_login` the previous value of `last_login`

ElggSite

The `ElggSite` entity type represents your Elgg installation (via your site URL).

Beyond the standard ElggEntity properties, ElggSites also support:

- `name` The site name
- `description` A description of the site
- `url` The address of the site

ElggGroup

The `ElggGroup` entity type represents an association of Elgg users. Users can join, leave, and post content to groups.

Beyond the standard ElggEntity properties, ElggGroups also support:

- `name` The group's name (HTML escaped text)
- `description` A description of the group (HTML)

`ElggGroup` has addition methods to manage content and membership.

The Groups plugin

Not to be confused with the entity type `ElggGroup`, Elgg comes with a plugin called "Groups" that provides a default UI/UX for site users to interact with groups. Each group is given a discussion forum and a profile page linking users to content within the group.

You can alter the user experience via the traditional means of extending plugins or completely replace the Groups plugin with your own.

Writing a group-aware plugin

Plugin owners need not worry too much about writing group-aware functionality, but there are a few key points:

Adding content

By passing along the group as `container_guid` via a hidden input field, you can use a single form and action to add both user and group content.

Use `ElggEntity->canWriteToContainer()` to determine whether or not the current user has the right to add content to a group.

Be aware that you will then need to pass the container GUID or username to the page responsible for posting and the accompanying value, so that this can then be stored in your form as a hidden input field, for easy passing to your actions. Within a "create" action, you'll need to take in this input field and save it as a property of your new element (defaulting to the current user's container):

```
$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());
```

Ownership

Entities have a `owner_guid` GUID property, which defines its owner. Typically this refers to the GUID of a user, although sites and users themselves often have no owner (a value of 0).

The ownership of an entity dictates, in part, whether or not you can access or edit that entity.

Containers

In order to easily search content by group or by user, content is generally set to be "contained" by either the user who posted it, or the group to which the user posted. This means the new object's `container_guid` property will be set to the GUID of the current ElggUser or the target ElggGroup.

E.g., three blog posts may be owned by different authors, but all be contained by the group they were posted to.

Note: This is not always true. Comment entities are contained by the object commented upon, and in some 3rd party plugins the container may be used to model a parent-child relationship between entities (e.g. a "folder" object containing a file object).

Annotations

Annotations are pieces of data attached to an entity that allow users to leave ratings, or other relevant feedback. A poll plugin might register votes as annotations.

Annotations are stored as instances of the `ElggAnnotation` class.

Each annotation has:

- An internal annotation type (like *comment*)
- A value (which can be a string or integer)
- An access permission distinct from the entity it's attached to
- An owner

Like metadata, values are stored as strings unless the value given is a PHP integer (`is_int($value)` is true), or unless the `$vartype` is manually specified as `integer`.

Adding an annotation

The easiest way to annotate is to use the `annotate` method on an entity, which is defined as:

```
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'
)
```

For example, to leave a rating on an entity, you might call:

```
$entity->annotate('rating', $rating_value, $entity->access_id);
```

Reading annotations

To retrieve annotations on an object, you can call the following method:

```
$annotations = $entity->getAnnotations(
    $name,    // The type of annotation
    $limit,   // The number to return
    $offset,  // Any indexing offset
```

(continues on next page)

(continued from previous page)

```
    $order,   // 'asc' or 'desc' (default 'asc')
);
```

If your annotation type largely deals with integer values, a couple of useful mathematical functions are provided:

```
$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
```

Useful helper functions

Comments

If you want to provide comment functionality on your plugin objects, the following function will provide the full listing, form and actions:

```
function elgg_view_comments(ElggEntity $entity)
```

Metadata

Metadata in Elgg allows you to store extra data on an `entity` beyond the built-in fields that entity supports. For example, `ElggObjects` only support the basic entity fields plus title and description, but you might want to include tags or an ISBN number. Similarly, you might want users to be able to save a date of birth.

Under the hood, metadata is stored as an instance of the `ElggMetadata` class, but you don't need to worry about that in practice (although if you're interested, see the `ElggMetadata` class reference). What you need to know is:

- Metadata has an owner, which may be different to the owner of the entity it's attached to
- You can potentially have multiple items of each type of metadata attached to a single entity
- Like annotations, values are stored as strings unless the value given is a PHP integer (`is_int($value)` is true), or unless the `$value_type` is manually specified as `integer` (see below).

Note: As of Elgg 3.0, metadata no longer have `access_id`.

The simple case

Adding metadata

To add a piece of metadata to an entity, just call:

```
$entity->metadata_name = $metadata_value;
```

For example, to add a date of birth to a user:

```
$user->dob = $dob_timestamp;
```

Or to add a couple of tags to an object:

```
$object->tags = array('tag one', 'tag two', 'tag three');
```

When adding metadata like this:

- The owner is set to the currently logged-in user
- Reassigning a piece of metadata will overwrite the old value

This is suitable for most purposes. Be careful to note which attributes are metadata and which are built in to the entity type that you are working with. You do not need to save an entity after adding or updating metadata. You do need to save an entity if you have changed one of its built in attributes. As an example, if you changed the access id of an ElggObject, you need to save it or the change isn't pushed to the database.

Note: As of Elgg 3.0, metadata's `access_id` property is ignored.

Reading metadata

To retrieve metadata, treat it as a property of the entity:

```
$tags_value = $object->tags;
```

Note that this will return the absolute value of the metadata. To get metadata as an ElggMetadata object, you will need to use the methods described in the *finer control* section below.

If you stored multiple values in this piece of metadata (as in the "tags" example above), you will get an array of all those values back. If you stored only one value, you will get a string or integer back. Storing an array with only one value will return a string back to you. E.g.

```
$object->tags = array('tag');
$tags = $object->tags;
// $tags will be the string "tag", NOT array('tag')
```

To always get an array back, simply cast to an array;

```
$tags = (array)$object->tags;
```

Reading metadata as objects

`elgg_get_metadata` is the best function for retrieving metadata as ElggMetadata objects:

E.g., to retrieve a user's DOB

```
elgg_get_metadata(array(
    'metadata_name' => 'dob',
    'metadata_owner_guid' => $user_guid,
));
```

Or to get all metadata objects:

```
elgg_get_metadata(array(
    'metadata_owner_guid' => $user_guid,
    'limit' => 0,
));
```

Common mistakes

"Appending" metadata

Note that you cannot "append" values to metadata arrays as if they were normal php arrays. For example, the following will not do what it looks like it should do.

```
$object->tags[] = "tag four";
```

Trying to store hashmaps

Elgg does not support storing ordered maps (name/value pairs) in metadata. For example, the following does not work as you might first expect it to:

```
// Won't work!! Only the array values are stored
$object->tags = array('one' => 'a', 'two' => 'b', 'three' => 'c');
```

You can instead store the information like so:

```
$object->one = 'a';
$object->two = 'b';
$object->three = 'c';
```

Storing GUIDs in metadata

Though there are some cases to store entity GUIDs in metadata, *Relationships* are a much better construct for relating entities to each other.

Relationships

Relationships allow you to bind entities together. Examples: an artist has fans, a user is a member of an organization, etc.

The class `ElggRelationship` models a directed relationship between two entities, making the statement:

> "**{subject}** is a **{noun}** of **{target}**."

API name	Models	Represents
`guid_one`	The subject	Which entity is being bound
`relationship`	The noun	The type of relationship
`guid_two`	The target	The entity to which the subject is bound

The type of relationship may alternately be a verb, making the statement:

> "**{subject} {verb} {target}**."
>
> E.g. User A "likes" blog post B

Each relationship has direction. Imagine an archer shoots an arrow at a target; The arrow moves in one direction, binding the subject (the archer) to the target.

A relationship does not imply reciprocity. **A** follows **B** does not imply that **B** follows **A**.

Relationships_ do not have access control. They're never hidden from view and can be edited with code at any privilege level, with the caveat that *the entities* in a relationship may be invisible due to access control!

Working with relationships

Creating a relationship

E.g. to establish that "**$user** is a **fan** of **$artist**" (user is the subject, artist is the target):

```
// option 1
$success = add_entity_relationship($user->guid, 'fan', $artist->guid);

// option 2
$success = $user->addRelationship($artist->guid, 'fan');
```

This triggers the event [create, relationship], passing in the created `ElggRelationship` object. If a handler returns `false`, the relationship will not be created and `$success` will be `false`.

Verifying a relationship

E.g. to verify that "**$user** is a **fan** of **$artist**":

```
if (check_entity_relationship($user->guid, 'fan', $artist->guid)) {
    // relationship exists
}
```

Note that, if the relationship exists, `check_entity_relationship()` returns an `ElggRelationship` object:

```
$relationship = check_entity_relationship($user->guid, 'fan', $artist->guid);
if ($relationship) {
    // use $relationship->id or $relationship->time_created
}
```

Deleting a relationship

E.g. to be able to assert that "**$user** is no longer a **fan** of **$artist**":

```
$was_removed = remove_entity_relationship($user->guid, 'fan', $artist->guid);
```

This triggers the event [delete, relationship], passing in the associated `ElggRelationship` object. If a handler returns `false`, the relationship will remain, and `$was_removed` will be `false`.

Other useful functions:

- `delete_relationship()` : delete by ID
- `remove_entity_relationships()` : delete those relating to an entity

Finding relationships and related entities

Below are a few functions to fetch relationship objects and/or related entities. A few are listed below:

- `get_entity_relationships()` : fetch relationships by subject or target entity

- `get_relationship()` : get a relationship object by ID
- `elgg_get_entities()` : fetch entities in relationships in a variety of ways

E.g. retrieving users who joined your group in January 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
));
```

Access Control

Granular access controls are one of the fundamental design principles in Elgg, and a feature that has been at the centre of the system throughout its development. The idea is simple: a user should have full control over who sees an item of data he or she creates.

Access controls in the data model

In order to achieve this, every entity and annotation contains an `access_id` property, which in turn corresponds to one of the pre-defined access controls or an entry in the `access_collections` database table.

Pre-defined access controls

- `ACCESS_PRIVATE` (value: 0) Private.
- `ACCESS_LOGGED_IN` (value: 1) Logged in users.
- `ACCESS_PUBLIC` (value: 2) Public data.

User defined access controls

You may define additional access groups and assign them to an entity, or annotation. A number of functions have been defined to assist you; see the *Access Control Lists* for more information.

How access affects data retrieval

All data retrieval functions above the database layer - for example `elgg_get_entities` will only return items that the current user has access to see. It is not possible to retrieve items that the current user does not have access to. This makes it very hard to create a security hole for retrieval.

Write access

The following rules govern write access:

- The owner of an entity can always edit it

- The owner of a container can edit anything therein (note that this does not mean that the owner of a group can edit anything therein)
- Admins can edit anything

You can override this behaviour using a *plugin hook* called `permissions_check`, which passes the entity in question to any function that has announced it wants to be referenced. Returning `true` will allow write access; returning `false` will deny it. See *the plugin hook reference for permissions_check* for more details.

Schema

The database contains a number of primary and secondary tables. You can follow schema changes in `engine/schema/migrations/`

The character set of the database should be `utf8mb4`, this will provide full unicode character support when storing data.

InnoDB

As of Elgg 3.0 the database uses the InnoDB engine. In order for a correct installation or migration some settings may need to be adjusted in the database settings.

- `innodb_large_prefix` should be `on`
- `innodb_file_format` should be `Barracuda`
- `innodb_file_per_table` should be `1`

Main tables

This is a description of the main tables. Keep in mind that in a given Elgg installation, the tables will have a prefix (typically "elgg_").

Table: entities

This is the main *Entities* table containing Elgg users, sites, objects and groups. When you first install Elgg this is automatically populated with your first site.

It contains the following fields:

- **guid** An auto-incrementing counter producing a GUID that uniquely identifies this entity in the system
- **type** The type of entity - object, user, group or site
- **subtype** A subtype of entity
- **owner_guid** The GUID of the owner's entity
- **container_guid** The GUID this entity is contained by - either a user or a group
- **access_id** Access controls on this entity
- **time_created** Unix timestamp of when the entity is created
- **time_updated** Unix timestamp of when the entity was updated
- **enabled** If this is 'yes' an entity is accessible, if 'no' the entity has been disabled (Elgg treats it as if it were deleted without actually removing it from the database)

Table: metadata

This table contains *Metadata*, extra information attached to an entity.

- **id** A unique IDentifier
- **entity_guid** The entity this is attached to
- **name** The name string
- **value** The value string
- **value_type** The value class, either text or an integer
- **time_created** Unix timestamp of when the metadata is created
- **enabled** If this is 'yes' an item is accessible, if 'no' the item has been disabled

Table: annotations

This table contains *Annotations*, this is distinct from *Metadata*.

- **id** A unique IDentifier
- **entity_guid** The entity this is attached to
- **name** The name string
- **value** The value string
- **value_type** The value class, either text or an integer
- **owner_guid** The owner GUID of the owner who set this annotation
- **access_id** An Access controls on this annotation
- **time_created** Unix timestamp of when the annotation is created.
- **enabled** If this is 'yes' an item is accessible, if 'no' the item has been disabled

Table: relationships

This table defines *Relationships*, these link one entity with another.

- **guid_one** The GUID of the subject entity.
- **relationship** The type of the relationship.
- **guid_two** The GUID of the target entity.

Secundairy tables

Table: access_collections

This table defines Access Collections, which grant users access to *Entities* or *Annotations*.

- **id** A unique IDentifier
- ***name** The name of the access collection
- **owner_guid** The GUID of the owning entity (eg. a user or a group)

- **subtype** the subtype of the access collection (eg. *friends* or *group_acl*)

3.5.3 Events and Plugin Hooks

Contents

Overview

Elgg has an event system that can be used to replace or extend core functionality.

Plugins influence the system by creating handlers (callables such as functions and methods) and registering them to handle two types of events: *Elgg Events* and *Plugin Hooks*.

When an event is triggered, a set of handlers is executed in order of priority. Each handler is passed arguments and has a chance to influence the process. After execution, the "trigger" function returns a value based on the behavior of the handlers.

See also:

- *List of events in core*
- *List of plugin hooks in core*

Elgg Events vs. Plugin Hooks

The main differences between *Elgg Events* and *Plugin Hooks* are:

1. Most Elgg events can be cancelled; unless the event is an "after" event, a handler that returns *false* can cancel the event, and no more handlers are called.
2. Plugin hooks cannot be cancelled; all handlers are always called.
3. Plugin hooks pass an arbitrary value through the handlers, giving each a chance to alter along the way.

Elgg Events

Elgg Events are triggered when an Elgg object is created, updated, or deleted; and at important milestones while the Elgg framework is loading. Examples: a blog post being created or a user logging in.

Unlike *Plugin Hooks*, *most Elgg events can be cancelled*, halting the execution of the handlers, and possibly cancelling an some action in the Elgg core.

Each Elgg event has a name and an object type (system, user, object, relationship name, annotation, group) describing the type of object passed to the handlers.

Before and After Events

Some events are split into "before" and "after". This avoids confusion around the state of the system while in flux. E.g. Is the user logged in during the [login, user] event?

Before Events have names ending in ":before" and are triggered before something happens. Like traditional events, handlers can cancel the event by returning `false`.

After Events, with names ending in ":after", are triggered after something happens. Unlike traditional events, handlers *cannot* cancel these events; all handlers will always be called.

Where before and after events are available, developers are encouraged to transition to them, though older events will be supported for backwards compatibility.

Elgg Event Handlers

Elgg event handlers are callables with one of the following prototypes:

```
<?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) {
    ...
}
```

In `event_handler1`, the `Event` object has various methods for getting the name, object type, and object of the event. See the `Elgg\Event` interface for details.

In both cases, if a handler returns `false`, the event is cancelled, preventing execution of the other handlers. All other return values are ignored.

Note: If the event type is "object" or "user", use type hint `\Elgg\ObjectEvent` or `\Elgg\UserEvent` instead, which clarify the return type of the `getObject()` method.

Register to handle an Elgg Event

Register your handler to an event using `elgg_register_event_handler`:

```
<?php

elgg_register_event_handler($event, $object_type, $handler, $priority);
```

Parameters:

- **$event** The event name.
- **$object_type** The object type (e.g. "user" or "object") or 'all' for all types on which the event is fired.
- **$handler** The callback of the handler function.
- **$priority** The priority - 0 is first and the default is 500.

Object here does not refer to an `ElggObject` but rather a string describing any object in the framework: system, user, object, relationship, annotation, group.

Example:

```
<?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);
```

Warning: If you handle the "update" event on an object, avoid calling `save()` in your event handler. For one it's probably not necessary as the object is saved after the event completes, but also because `save()` calls another "update" event and makes `$object->getOriginalAttributes()` no longer available.

Invokable classes as handlers

You may use a class with an `__invoke()` method as a handler. Just register the class name and it will be instantiated (with no arguments) for the lifetime of the event (or hook).

```
<?php

namespace MyPlugin;

class UpdateObjectHandler {
    public function __invoke(\Elgg\ObjectEvent $event) {

    }
}

// in init, system
elgg_register_event_handler('update', 'object', MyPlugin\UpdateObjectHandler::class);
```

Trigger an Elgg Event

You can trigger a custom Elgg event using `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.
}
```

For events with ambiguous states, like logging in a user, you should use *Before and After Events* by calling `elgg_trigger_before_event` or `elgg_trigger_after_event`. This makes it clear for the event handler what state to expect and which events can be cancelled.

```
<?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);
```

Parameters:

- **$event** The event name.
- **$object_type** The object type (e.g. "user" or "object").
- **$object** The object (e.g. an instance of `ElggUser` or `ElggGroup`)

The function will return `false` if any of the selected handlers returned `false` and the event is stoppable, otherwise it will return `true`.

Plugin Hooks

Plugin Hooks provide a way for plugins to collaboratively determine or alter a value. For example, to decide whether a user has permission to edit an entity or to add additional configuration options to a plugin.

A plugin hook has a value passed into the trigger function, and each handler has an opportunity to alter the value before it's passed to the next handler. After the last handler has completed, the final value is returned by the trigger.

Plugin Hook Handlers

Hook handlers are callables with one of the following prototypes:

```
<?php

/**
 * @param \Elgg\Hook $hook The hook object
 *
 * @return mixed if not null, this will be the new value of the plugin hook
 */
```

(continues on next page)

(continued from previous page)

```
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) {
    ...
}
```

In `plugin_hook_handler1`, the `Hook` object has various methods for getting the name, type, value, and parameters of the hook. See the `Elgg\Hook` interface for details.

In both cases, if the handler returns no value (or `null` explicitly), the plugin hook value is not altered. Otherwise the returned value becomes the new value of the plugin hook, and it will then be available as `$hook->getValue()` (or `$value`) in the next handler.

Register to handle a Plugin Hook

Register your handler to a plugin hook using `elgg_register_plugin_hook_handler`:

```
<?php

elgg_register_plugin_hook_handler($hook, $type, $handler, $priority);
```

Parameters:

- **$hook** The name of the plugin hook.
- **$type** The type of the hook or 'all' for all types.
- **$handler** The callback of the handler function.
- **$priority** The priority - 0 is first and the default is 500.

Type can vary in meaning. It may mean an Elgg entity type or something specific to the plugin hook name.

Example:

```
<?php

// Register the function myPlugin_hourly_job() to be called with priority 400.
elgg_register_plugin_hook_handler('cron', 'hourly', 'myPlugin_hourly_job', 400);
```

Trigger a Plugin Hook

You can trigger a custom plugin hook using `elgg_trigger_plugin_hook`:

```
<?php

// filter $value through the handlers
$value = elgg_trigger_plugin_hook($hook, $type, $params, $value);
```

Parameters:

- **$hook** The name of the plugin hook.
- **$type** The type of the hook or 'all' for all types.
- **$params** Arbitrary data passed from the trigger to the handlers.
- **$value** The initial value of the plugin hook.

Warning: The *$params* and *$value* arguments are reversed between the plugin hook handlers and trigger functions!

Unregister Event/Hook Handlers

The functions `elgg_unregister_event_handler` and `elgg_unregister_plugin_hook_handler` can be used to remove handlers already registered by another plugin or Elgg core. The parameters are in the same order as the registration functions, except there's no priority parameter.

```
<?php

elgg_unregister_event_handler('login', 'user', 'myPlugin_handle_login');
```

Anonymous functions or invokable objects cannot be unregistered, but dynamic method callbacks can be unregistered by giving the static version of the callback:

```
<?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');
```

Even though the event handler references a dynamic method call, the code above will successfully remove the handler.

Handler Calling Order

Handlers are called first in order of priority, then registration order.

Note: Before Elgg 2.0, registering with the `all` keywords caused handlers to be called later, even if they were registered with lower priorities.

3.5.4 Internationalization

Elgg 1.0+ departs from previous versions in that it uses a custom text array rather than gettext. This improves system performance and reliability of the translation system.

TODO: more plz

3.5.5 AMD

Overview

There are two JavaScript system in Elgg: the deprecated 1.8 system, and the newer AMD (Asynchronous Module Definition) compatible system introduced in 1.9.

This discusses the benefits of using AMD in Elgg.

Why AMD?

We have been working hard to make Elgg's JavaScript more maintainable and useful. We made some strides in 1.8 with the introduction of the "`elgg`" JavaScript object and library, but have quickly realized the approach we were taking was not scalable.

The size of JS on the web is growing quickly, and JS in Elgg is growing too. We want Elgg to be able to offer a solution that makes JS development as productive and maintainable as possible going forward.

The reasons to choose AMD are plenteous and well-documented. Let's highlight just a few of the most relevant reasons as they relate to Elgg specifically.

1. Simplified dependency management

AMD modules load asynchronously and execute as soon as their dependencies are available, so this eliminates the need to specify "priority" and "location" when registering JS libs in Elgg. Also, you don't need to worry about explicitly loading a module's dependencies in PHP. The AMD loader (RequireJS in this case) takes care of all that hassle for you. It's also possible have text dependencies with the RequireJS text plugin, so client-side templating should be a breeze.

2. AMD works in all browsers. Today.

Elgg developers are already writing lots of JavaScript. We know you want to write more. We cannot accept waiting 5-10 years for a native JS modules solution to be available in all browsers before we can organize our JavaScript in a maintainable way.

3. You do not need a build step to develop in AMD.

We like the edit-refresh cycle of web development. We wanted to make sure everyone developing in Elgg could continue experiencing that joy. Synchronous module formats like Closure or CommonJS just weren't an option for us. But even though AMD doesn't require a build step, *it is still very build-friendly*. Because of the `define()` wrapper, it's possible to concatenate multiple modules into a single file and ship them all at once in a production environment.[1]

[1] This is not currently supported by Elgg core, but we'll be looking into it since reducing round-trips is critical for a good first-view experience, especially on mobile devices.

AMD is a battle-tested and well thought out module loading system for the web today. We're very thankful for the work that has gone into it, and are excited to offer it as the standard solution for JavaScript development in Elgg starting with Elgg 1.9.

3.5.6 Security

Elgg's approach to the various security issues common to all web applications.

Tip: To report a potential vulnerability in Elgg, email security@elgg.org.

Contents

Passwords

Password validation

The only restriction that Elgg places on a password is that it must be at least 6 characters long by default, though this may be changed in `/elgg-config/settings.php`. Additional criteria can be added by a plugin by registering for the `registeruser:validate:password` plugin hook.

Password hashing

Passwords are never stored in plain text, only salted hashes produced with bcrypt. This is done via the standard `password_hash()` function. On older systems, the `password-compat` polyfill is used, but the algorithm is

identical.

Elgg installations created before version 1.10 may have residual "legacy" password hashes created using salted MD5. These are migrated to bcrypt as users log in, and will be completely removed when a system is upgraded to Elgg 3.0. In the meantime we're happy to assist site owners to manually remove these legacy hashes, though it would force those users to reset their passwords.

Password throttling

Elgg has a password throttling mechanism to make dictionary attacks from the outside very difficult. A user is only allowed 5 login attempts over a 5 minute period.

Password resetting

If a user forgets his password, a new random password can be requested. After the request, an email is sent with a unique URL. When the user visits that URL, a new random password is sent to the user through email.

Sessions

Elgg uses PHP's session handling with custom handlers. Session data is stored in the database. The session cookie contains the session id that links the user to the browser. The user's metadata is stored in the session including GUID, username, email address.

The session's lifetime is controlled through the server's PHP configuration and additionally through options in the `/elgg-config/settings.php`.

Session fixation

Elgg protects against session fixation by regenerating the session id when a user logs in.

"Remember me" cookie

To allow users to stay logged in for a longer period of time regardless of whether the browser has been closed, Elgg uses a cookie (default called `elggperm`) that contains what could be considered a super session identifier. This identifier is stored in a cookies table. When a session is being initiated, Elgg checks for the presence of the `elggperm` cookie. If it exists and the session code in the cookie matches the code in the cookies table, the corresponding user is automatically logged in.

When a user changes their password all existing permanent cookie codes are removed from the database.

The lifetime of the persistent cookie can be controlled in the */elgg-config/settings.php* file. The default lifetime is 30 days. The database records for the persistent cookies will be removed after the lifetime expired.

Alternative authentication

Note: This section is very hand-wavy

To replace Elgg's default user authentication system, a plugin could replace the default `login` action with its own. Better would be to register a PAM handler using `register_pam_handler()` which handles the authentication of the user based on the new requirements.

HTTPS

Note: You must enable SSL support on your server for any of these techniques to work.

You can serve your whole site over SSL by changing the site URL to include "https" instead of just "http".

XSS

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. 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.

CSRF / XSRF

Elgg generates security tokens to prevent cross-site request forgery. These are embedded in all forms and state-modifying AJAX requests as long as the correct API is used. Read more in the *Forms + Actions* developer guide.

Signed URLs

It's possible to protect URLs with a unique signature. Read more in the *Forms + Actions* developer guide.

SQL Injection

Elgg's API sanitizes all input before issuing DB queries. Read more in the *Database* design doc.

Privacy

Elgg uses an ACL system to control which users have access to various pieces of content. Read more in the *Database* design doc.

Hardening

Site administrators can configure settings which will help with hardening the website. Read more in the Administrator guide *Security*.

3.5.7 Loggable

Loggable is an interface inherited by any class that wants events relating to its member objects to be saved to the system log. `ElggEntity` and `ElggExtender` both inherit `Loggable`.

Loggable defines several class methods that are used in saving to the default system log, and can be used to define your own (as well as for other purposes):

- `getSystemLogID()` Return a unique identifier for the object for storage in the system log. This is likely to be the object's GUID
- `getClassName()` Return the class name of the object
- `getType()` Return the object type
- `getSubtype()` Get the object subtype
- `getObjectFromID($id)` For a given ID, return the object associated with it

Database details

The default system log is stored in the `system_log` *database table*. It contains the following fields:

- **id** - A unique numeric row ID
- **object_id** - The GUID of the entity being acted upon
- **object_class** - The class of the entity being acted upon (eg ElggObject)
- **object_type** - The type of the entity being acted upon (eg object)
- **object_subtype** - The subtype of the entity being acted upon (eg blog)
- **event** - The event being logged (eg create or update)
- **performed_by_guid** - The GUID of the acting entity (the user performing the action)
- **owner_guid** - The GUID of the user which owns the entity being acted upon
- **access_id** - The access restriction associated with this log entry
- **time_created** - The UNIX epoch timestamp of the time the event took place

3.6 Contributor Guides

Participate in making Elgg even better.

Elgg is a community-driven project. It relies on the support of volunteers to succeed. Here are some ways you can help:

3.6.1 Writing Code

Understand Elgg's standards and processes to get your changes accepted as quickly as possible.

Contents

License agreement

By submitting a patch you are agreeing to license the code under a GPLv2 license and MIT license.

Pull requests

Pull requests (PRs) are the best way to get code contributed to Elgg core. The core development team uses them even for the most trivial changes.

For new features, submit a feature request or talk to us first and make sure the core team approves of your direction before spending lots of time on code.

Checklists

Use these markdown checklists for new PRs on github to ensure high-quality contributions and help everyone understand the status of open PRs.

Bugfix PRs:

```
- [ ] 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
```

Feature PRs:

```
- [ ] 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
```

Choosing a branch to submit to

The following table assumes the latest stable release is 2.1.

Type of change	Branch to submit against
Security fix	Don't! Email security@elgg.org for guidance.
Bug fix	1.12 (or 2.1 if the 1.12 fix is too complex)
Performance	2.x
Deprecation	2.x
Minor feature	2.x
Major feature	master
Has any breaking change	master

If you're not sure which branch to submit against, just ask!

The difference between minor and major feature is subjective and up to the core team.

Commit message format

We require a particular format to allow releasing more often, and with improved changelogs and source history. Just follow these steps:

1. Start with the `type` by selecting the *last category which applies* from this list:
   - **docs** - *only* docs are being updated
   - **chore** - this include refactoring, code style changes, adding missing tests, Travis stuff, etc.
   - **perf** - the primary purpose is to improve performance
   - **fix** - this fixes a bug
   - **deprecate** - the change deprecates any part of the API
   - **feature** - this adds a new user-facing or developer feature
   - **security** - the change affects a security issue in any way. *Please do not push this commit to any public repo.* Instead contact security@elgg.org.

   E.g. if your commit refactors to fix a bug, it's still a "fix". If that bug is security-related, however, the type must be "security" and you should email security@elgg.org before proceeding. When in doubt, make your best guess and a reviewer will provide guidance.

2. In parenthesis, add the `component`, a short string which describes the subsystem being changed.

   Some examples: `views`, `i18n`, `seo`, `a11y`, `cache`, `db`, `session`, `router`, `<plugin_name>`.

3. Add a colon, a space, and a brief `summary` of the changes, which will appear in the changelog.

   No line may exceed 100 characters in length, so keep your summary concise.

Good summary	Bad summary (problem)
page owners see their own owner blocks on pages	bug fix (vague)
bar view no longer dies if 'foo' not set	updates views/default/bar.php so bar view no longer... (redundant info)
narrows river layout to fit iPhone	alters the river layout (vague)
elgg_foo() handles arrays for $bar	in elgg_foo() you can now pass an array for $bar and the function will... (move detail to description)
removes link color from comments header in river	fixes db so that... (redundant info)
requires non-empty title when saving pages	can save pages with no title (confusingly summarizes old behavior)

4. (recommended) Skip a line and add a `description` of the changes. Include the motivation for making them, any info about back or forward compatibility, and any rationale of why the change had to be done a certain way. Example:

   > 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.

   Unless your change is trivial/obvious, a description is required.

5. If the commit resolves a GitHub issue, skip a line and add `Fixes #` followed by the issue number. E.g. `Fixes #1234`. You can include multiple issues by separating with commas.

   GitHub will auto-close the issue when the commit is merged. If you just want to reference an issue, use `Refs #` instead.

When done, your commit message will have the format:

```
type(component): summary

Optional body
Details about the solution.
Opportunity to call out as breaking change.

Closes/Fixes/Refs #123, #456, #789
```

Here is an example of a good commit message:

```
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
```

To validate commit messages locally, make sure `.scripts/validate_commit_msg.php` is executable, and make a copy or symlink to it in the directory `.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
```

Rewriting commit messages

If your PR does not conform to the standard commit message format, we'll ask you to rewrite it.

To edit just the last commit:

1. Amend the commit: `git commit --amend` (git opens the message in a text editor).
2. Change the message and save/exit the editor.
3. Force push your branch: `git push -f your_remote your_branch` (your PR with be updated).
4. Rename the PR title to match

Otherwise you may need to perform an interactive rebase:

1. Rebase the last N commits: `git rebase -i HEAD~N` where N is a number. (Git will open the `git-rebase-todo` file for editing)
2. For the commits that need to change, change `pick` to `r` (for reword) and save/exit the editor.

3. Change the commit message(s), save/exit the editor (git will present a file for each commit that needs rewording).
4. `git push -f your_remote your_branch` to force push the branch (updating your PR).
5. Rename the PR title to match

Coding Standards

Elgg uses set of standards that are based partially on PEAR and PSR2 standards. You can view the ruleset in `vendor/elgg/sniffs/elgg.xml`.

To check your code for standard violations (provided you have installed Elgg with dev dependencies), run:

```
phpcs --standard=vendor/elgg/sniffs/elgg.xml -s path/to/dir/to/check
```

To automatically fix fixable violations, run:

```
phpcbf --standard=vendor/elgg/sniffs/elgg.xml path/to/dir/to/fix
```

To check core directories, you can use shortcut `composer lint` and `composer lint-fixer`.

Testing

Elgg has automated tests for both PHP and JavaScript functionality. All new contributions are required to come with appropriate tests.

See also:

Writing tests

General guidelines

Break tests up by the behaviors you want to test and use names that describe the behavior. E.g.:

- Not so good: One big method `testAdd()`.
- Better: Methods `testAddingZeroChangesNothing` and `testAddingNegativeNumberSubtracts`

Strive for *componentized designs* that allow testing in isolation, without large dependency graphs or DB access. Injecting dependencies is key here.

PHP Tests

PHPUnit

Located in `engine/tests/phpunit`, this is our preferred test suite. It uses no DB access, and has only superficial access to the entities API.

- We encourage you to create components that are testable in this suite if possible.
- Consider separating storage from your component so at least business logic can be tested here.
- Depend on the `Elgg\Filesystem\*` classes rather than using PHP filesystem functions.

Testing interactions between services

Ideally your tests would construct your own isolated object graphs for direct manipulation, but this isn't always possible.

If your test relies on Elgg's Service Provider (`_elgg_services()` returns a `Elgg\Di\ServiceProvider`), realize that it maintains a singleton instance for most services it hands out, and many services keep their own local references to these services as well.

Due to these local references, replacing services on the SP within a test often will not have the desired effect. Instead, you may need to use functionality baked into the services themselves:

- The `events` and `hooks` services have methods `backup()` and `restore()`.
- The `logger` service has methods `disable()` and `enable()`.

Jasmine Tests

Test files must be named `*Test.js` and should go in either `js/tests/` or next to their source files in `views/default/**.js`. Karma will automatically pick up on new `*Test.js` files and run those tests.

Test boilerplate

```
define(function(require) {
        var elgg = require('elgg');

        describe("This new test", function() {
                it("fails automatically", function() {
                        expect(true).toBe(false);
                });
        });
});
```

Running the tests

Elgg uses Karma with Jasmine to run JS unit tests.

You will need to have nodejs and yarn installed.

First install all the development dependencies:

```
yarn
```

Run through the tests just once and then quit:

```
yarn test
```

You can also run tests continuously during development so they run on each save:

```
karma start js/tests/karma.conf.js
```

Debugging JS tests

You can run the test suite inside Chrome dev tools:

```
yarn run chrome
```

This will output a URL like `http://localhost:9876/`.

1. Open the URL in Chrome, and click "Debug".
2. Open Chrome dev tools and the Console tab.
3. Reload the page.

If you alter a test you'll have to quit Karma with `Ctrl-c` and restart it.

Coding best practices

Make your code easier to read, easier to maintain, and easier to debug. Consistent use of these guidelines means less guess work for developers, which means happier, more productive developers.

General coding

Don't Repeat Yourself

If you are copy-pasting code a significant amount of code, consider whether there's an opportunity to reduce duplication by introducing a function, an additional argument, a view, or a new component class.

E.g. If you find views that are identical except for a single value, refactor into a single view that takes an option.

Note: In a bugfix release, *some duplication is preferrable to refactoring*. Fix bugs in the simplest way possible and refactor to reduce duplication in the next minor release branch.

Embrace SOLID and GRASP

Use these principles for OO design to solve problems using loosely coupled components, and try to make all components and integration code testable.

Whitespace is free

Don't be afraid to use it to separate blocks of code. Use a single space to separate function params and string concatenation.

Variable names

Use self-documenting variable names. `$group_guids` is better than `$array`.

Avoid double-negatives. Prefer `$enable = true` to `$disable = false`.

Interface names

Use the pattern `Elgg\{Namespace}\{Name}`.

Do not include an `I` prefix or an `Interface` suffix.

We do not include any prefix or suffix so that we're encouraged to:

- name implementation classes more descriptively (the "default" name is taken).
- type-hint on interfaces, because that is the shortest, easiest thing to do.

Name implementations like `Elgg\{Namespace}\{Interface}\{Implementation}`.

Functions

Where possible, have functions/methods return a single type. Use empty values such as `array()`, `""`, or `0` to indicate no results.

Be careful where valid return values (like `"0"`) could be interpreted as empty.

Functions not throwing an exception on error should return `false` upon failure.

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.

Functions returning only boolean should be prefaced with `is_` or `has_` (eg, `elgg_is_logged_in()`, `elgg_has_access_to_entity()`).

Ternary syntax

Acceptable only for single-line, non-embedded statements.

Minimize complexity

Minimize nested blocks and distinct execution paths through code. Use Return Early to reduce nesting levels and cognitive load when reading code.

Use comments effectively

Good comments describe the "why." Good code describes the "how." E.g.:

Bad:

```
// increment $i only when the entity is marked as active.
foreach ($entities as $entity) {
        if ($entity->active) {
                $i++;
        }
}
```

Good:

```
// find the next index for inserting a new active entity.
foreach ($entities as $entity) {
        if ($entity->active) {
                $i++;
        }
}
```

Always include a comment if it's not obvious that something must be done in a certain way. Other developers looking at the code should be discouraged from refactoring in a way that would break the code.

```
// Can't use empty()/boolean: "0" is a valid value
if ($str === '') {
    register_error(elgg_echo('foo:string_cannot_be_empty'));
    forward(REFERER);
}
```

Commit effectively

- Err on the side of atomic commits which are highly focused on changing one aspect of the system.
- Avoid mixing in unrelated changes or extensive whitespace changes. Commits with many changes are scary and make pull requests difficult to review.
- Use visual git tools to craft highly precise and readable diffs.

Include tests

When at all possible *include unit tests* for code you add or alter.

Keep bugfixes simple

Avoid the temptation to refactor code for a bugfix release. Doing so tends to introduce regressions, breaking functionality in what should be a stable release.

PHP guidelines

These are the required coding standards for Elgg core and all bundled plugins. Plugin developers are strongly encouraged to adopt these standards.

Developers should first read the PSR-2 Coding Standard Guide.

Elgg's standards extend PSR-2, but differ in the following ways:

- Indent using one tab character, not spaces.
- Opening braces for classes, methods, and functions must go on the same line.
- If a line reaches over 100 characters, consider refactoring (e.g. introduce variables).
- Compliance with PSR-1 is encouraged, but not strictly required.

Documentation

- Include PHPDoc comments on functions and classes (all methods; declared properties when appropriate), including types and descriptions of all parameters.
- In lists of `@param` declarations, the beginnings of variable names and descriptions must line up.
- 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`.
- Use `//` or `/* */` when commenting.
- Use only `//` comments inside function/method bodies.

Naming

- Use underscores to separate words in the names of functions, variables, and properties. Method names are camelCase.
- Names of functions for public use must begin with `elgg_`.
- All other function names must begin with `_elgg_`.
- Name globals and constants in `ALL_CAPS` (`ACCESS_PUBLIC`, `$CONFIG`).

Miscellaneous

For PHP requirements, see `composer.json`.

Do not use PHP shortcut tags `<?` or `<%`. It is OK to use `<?=` since it is always enabled as of PHP 5.4.

When creating strings with variables:

- use double-quoted strings
- wrap variables with braces only when necessary.

Bad (hard to read, misuse of quotes and {}s):

```
echo 'Hello, '.$name."!  How is your {$time_of_day}?";
```

Good:

```
echo "Hello, $name!  How is your $time_of_day?";
```

Remove trailing whitespace at the end of lines. An easy way to do this before you commit is to run `php .scripts/fix_style.php` from the installation root.

Value validation

When working with user input prepare the input outside of the validation method.

Bad:

```
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
}
```

Good:

```
function validate_email($email) {
        // validate
}

$email = get_input($email);
$email = trim($email);

if (validate_email($email)) {
        // green light
}
```

Use exceptions

Do not be afraid to use exceptions. They are easier to deal with than mixed function output:

Bad:

```
/**
* @return string|bool
*/
function validate_email($email) {
        if (empty($email)) {
                return 'Email is empty';
        }

        // validate

        return true;
}
```

Good:

```
/**
* @return void
* @throw InvalidArgumentException
*/
function validate_email($email) {
        if (empty($email)) {
                throw new InvalidArgumentException('Email is empty');
        }
```

(continues on next page)

(continued from previous page)

```
        // validate and throw if invalid
}
```

Documenting return values

Do not use `@return void` on methods that return a value or null.

Bad:

```
/**
* @return bool|void
*/
function validate_email($email) {
        if (empty($email)) {
                return;
        }

        // validate

        return true;
}
```

Good:

```
/**
* @return bool|null
*/
function validate_email($email) {
        if (empty($email)) {
                return null;
        }

        // validate

        return true;
}
```

CSS guidelines

Save the css in files with a `.css` extension.

Use shorthand where possible

Bad:

```
background-color: #333333;
background-image:  url(...);
background-repeat:  repeat-x;
background-position:  left 10px;
padding: 2px 9px 2px 9px;
```

Good:

```
background: #333 url(...) repeat-x left 10px;
padding: 2px 9px;
```

Use hyphens, not underscores

Bad:

```
.example_class {}
```

Good:

```
.example-class {}
```

Note: You should prefix your ids and classnames with text that identifies your plugin.

One property per line

Bad:

```
color: white;font-size: smaller;
```

Good:

```
color: white;
font-size: smaller;
```

Property declarations

These should be spaced like so: `property:  value;`

Bad:

```
color:value;
color :value;
color : value;
```

Good:

```
color: value;
```

Vendor prefixes

- Group vendor-prefixes for the same property together
- Longest vendor-prefixed version first
- Always include non-vendor-prefixed version
- Put an extra newline between vendor-prefixed groups and other properties

Bad:

```
-moz-border-radius: 5px;
border: 1px solid #999999;
-webkit-border-radius: 5px;
width: auto;
```

Good:

```
border: 1px solid #999999;

-webkit-border-radius: 5px;
-moz-border-radius: 5px;
border-radius: 5px;

width: auto;
```

Group subproperties

Bad:

```
background-color: white;
color: #0054A7;
background-position: 2px -257px;
```

Good:

```
background-color: white;
background-position: 2px -257px;
color: #0054A7;
```

Javascript guidelines

Same formatting standards as PHP apply.

All functions should be in the `elgg` namespace.

Function expressions should end with a semi-colon.

```
elgg.ui.toggles = function(event) {
        event.preventDefault();
        $(target).slideToggle('medium');
};
```

Deprecating APIs

Occasionally functions and classes must be deprecated in favor of newer replacements. Since 3rd party plugin authors rely on a consistent API, backward compatibility must be maintained, but will not be maintained indefinitely as plugin authors are expected to properly update their plugins. In order to maintain backward compatibility, deprecated APIs will follow these guidelines:

- Minor version (1.x) that deprecates an API must include a wrapper function/class (or otherwise appropriate means) to maintain backward compatibility, including any bugs in the original function/class. This compatibility layer uses `elgg_deprecated_notice('...', '1.11')` to log that the function is deprecated.

- The next major revision (2.0) removes the compatibility layer. Any use of the deprecated API should be corrected before this.

3.6.2 Database

Contributing database schema changes

Contents

Database Migrations

Elgg uses Phinx to manage the database migrations.

Create a migration

To create a new migration run the following in your console:

```
vendor/bin/phinx create -c engine/conf/migrations.php MigrationClassName
```

This will generate a timestamped skeleton migration in `engine/schema/migrations/`. Follow Phinx documentation to call the necessary methods to modify the database tables.

Executing a migration

Migrations are executed every time your run `upgrade.php`. If you would like to execute migrations manually, you can do so via the command line:

```
// When Elgg is the root project
vendor/bin/phinx migrate -c engine/conf/migrations.php

// When Elgg is installed as a Composer dependency
vendor/bin/phinx migrate -c vendor/elgg/elgg/engine/conf/migrations.php
```

Check Phinx documentation for additional flags that allow you to run a single migration or a set of migrations within a time range.

3.6.3 Writing Documentation

New documentation should fit well with the rest of Elgg's docs.

Contents

- *Avoid first person pronouns*
- *Eliminate fluff*
- *Prefer absolute dates over relative ones*
- *Do not remind the reader to contribute*

Testing docs locally

Elgg has a grunt script that automatically builds the docs, opens them in a browser window, and automatically reloads as you make changes (the reload takes just a few seconds). You need yarn and sphinx installed to be able to use these scripts.

```
cd path/to/elgg/
yarn
grunt
```

It's that easy! Grunt will continue running, watching the docs for changes and automatically rebuilding.

Follow the existing document organization

The current breakdown is not necessarily the One True Way to organize docs, but consistency is better than randomness.

intro/*

This is everything that brand new users need to know (installation, features, license, etc.)

admin/*

Guides for administrators. Task-oriented.

guides/*

API guides for plugin developers. Cookbook-style. Example heavy. Code snippet heavy. Broken down by services (actions, i18n, routing, db, etc.). This should only discuss the public API and its behavior, not implementation details or reasoning.

design/*

Design docs for people who want to get a better understanding of how/why core is built the way it is. This should discuss internal implementation details of the various services, what tradeoffs were made, and the reasoning behind the final decision. Should be useful for people who want to contribute and for communication b/w core devs.

contribute/*

Contributors guides for the various ways people can participate in the project.

appendix/*

More detailed/meta/background information about the project (history, roadmap, etc.)

Use "Elgg" in a grammatically correct way

Elgg is not an acronym, so writing it in all caps (ELGG or E-LGG) is incorrect. Please don't do this.

In English, Elgg does not take an article when used as a noun. Here are some examples to emulate:

- "I'm using Elgg to run my website"
- "Install Elgg to get your community online"

When used as an adjective, the article applies to the main noun, so you should use one. For example:

- "Go to the Elgg community website to get help."
- "I built an Elgg-based network yesterday"

This advice may not apply in languages other than English.

Avoid first person pronouns

Refer to the reader as "you". Do not include yourself in the normal narrative.

Before:

> When we're done installing Elgg, we'll look for some plugins!

After:

> When you're done installing Elgg, look for some plugins!

To refer to yourself (avoid this if possible), use your name and write in the third person. This clarifies to future readers/editors whose opinions are being expressed.

Before:

> I think the best way to do X is to use Y.

After:

> Evan thinks the best way to do X is to use Y.

Eliminate fluff

Before:

> If you want to use a third-party javascript library within the Elgg framework, you should take care to call the `elgg_register_js` function to register it.

After:

> To use a third-party javascript library, call `elgg_register_js` to register it.

Prefer absolute dates over relative ones

It is not easy to tell when a particular sentence or paragraph was written, so relative dates quickly become meaningless. Absolute dates also give the reader a good indication of whether a project has been abandoned, or whether some advice might be out of date.

Before:

> Recently the foo was barred. Soon, the baz will be barred too.

After:

> Recently (as of September 2013), the foo was barred. The baz is expected to be barred by October 2013.

Do not remind the reader to contribute

Focus on addressing only the topic at hand. Constant solicitation for free work is annoying and makes the project look needy. If people want to contribute to the project, they can visit the contributor guide.

3.6.4 Internationalizing documentation

When you change documentation, remember to update the documentation translation templates before you commit:

```
cd docs/
make gettext
```

For more information, see http://www.sphinx-doc.org/en/stable/intl.html#translating-with-sphinx-intl

Special attention

When translating the documentation be aware of special syntax in the documentation files.

Translating links

- Translate text in anonymous links (e.g., `` `pronunciation`__ ``), but maintain the order of all anonymous links in a single block. If there are two anonymous links within a single block for translation, they must not be rearranged relative to each other.
- Translate the text of named links (e.g., `` `demo site`_ ``) but only if you maintain the name using the correct rST syntax. In this case that would be `` `translation of "demo site" <demo site_>`_ ``.

Do NOT translate

- Anything between pipe characters should not be translated (e.g., master).
- Code, unless it's a comment in the code.

3.6.5 Translations

Translations multiply the impact that Elgg can have by making it accessible to a larger percentage of the world.

The community will always be indebted to those of you who work hard to provide high quality translations for Elgg's UI and docs.

Transifex

All translation for the Elgg project is organized through Transifex.

https://www.transifex.com/organization/elgg

Plugin authors are encouraged to coordinate translations via Transifex as well so the whole community can be unified and make it really easy for translators to contribute to any plugin in the Elgg ecosystem.

Pulling translations

The translations made in Transifex need to be periodically pulled into the Elgg code repository. This can be done with the script `.scripts/languages.php` bundled within Elgg's source code.

Prerequisites for running the script are:

- Access to command line
- Git
- Transifex CLI tool

The script will do the following steps:

1. Create a new git branch named `{branch}_i18n_{timestamp}`
2. Pull translations for all languages that have 95% of the strings translated
3. Remove possible invalid language codes
4. Commit the changes to the branch

After this you must push the branch to Github and make a new Pull request.

For example if you want to pull the translations for the `3.x` branch, run the following commands:

```
php .scripts/languages.php 3.x
git push -u your_fork 3.x_i18n_1515151617
```

Run the command without parameters to get more detailed information of the usage.

Transifex configuration

The configuration for Transifex can be found from Elgg's source code in the file `.tx/config`.

This file defines:

- The Transifex project associated with Elgg's major version
- The location of all the files that have translatable content

Read the Transifex documentation for further details.

New major Elgg version

Every major version of Elgg must have its own project in Transifex. This way we can make sure that strings added and removed between versions do not conflict with each other. For example a translation key removed in Elgg 3 should not get removed from translations made for Elgg 2. Respectfully a new string added only to Elgg 3 should not be included in the translations meant for Elgg 2.

The process of setting up a new major version is:

1. Pull latest translations from Transifex to the previous major version
2. Merge the git branch of the previous version to the new to make sure all the latest translation keys are present
3. Create a new Transifex project to https://www.transifex.com/elgg/
4. Update `.tx/config` file in the development branch of the new major version
   - Update the configuration to point to the new Transifex project
   - Remove configuration of removed plugins
   - Add configuration for new plugins
5. Push the translation sources to the new Transifex project with the command:

```
tx push -s
```

6. Copy the new configuration file temporarily (do not commit) to the previous major version, and push the existing translations from it to the new project:

```
tx push -t -f --no-interactive
```

Later, once the dedicated branch (e.g. `3.x` has been created for the major version, configure Transifex to fetch new translation keys from it automatically in https://www.transifex.com/elgg/elgg-core-3/content/. This way you don't have to repeat step 5 manually every time new translation keys are added.

It is important to always have a `n.x` branch besides the branches meant for specific minor versions (`n.1`, `n.2`, etc.). This way the URLs of the auto-update sources do not have to be updated every time a new minor branch is created.

3.6.6 Reporting Issues

Report bugs and features requests to https://github.com/Elgg/Elgg/issues. See below for guidelines.

DISCLAIMERS

Attention: Security issues should be reported to **security @ elgg . org**! Please do not post any security issues on github!!

Note: Support requests belong on the community site. Tickets with support requests will be closed.

Important: We cannot make any guarantees as to when your ticket will be resolved.

Bug reports

Before submitting a bug report:

- Search for an existing ticket on the issue you're having. Add any extra info there.
- Verify the problem is reproducible

  - On the latest version of Elgg
  - With all third-party plugins disabled

Good bug report checklist:

- Expected behavior and actual behavior
- Clear steps to reproduce the problem
- The version of Elgg you're running
- Browsers affected by this problem

Feature requests

Before submitting a feature request:

- Check the community site for a plugin that has the features you need.
- Consider if you can *develop a plugin* that does what you need.
- Search through the closed tickets to see if someone else suggested the same feature, but got turned down. You'll need to be able to explain why your suggestion should be considered this time.

Good feature request checklist:

- Detailed explanation of the feature
- Real-life use-cases
- Proposed API

3.6.7 Becoming a Financial Supporter

All funds raised via the Elgg supporters network go directly into:

- Elgg core development
- Infrastructure provision (elgg.org, github, etc.)

It is a great way to help with Elgg development!

Benefits

For only $50 per year for individuals or $150 per year for organizations, you can get listed as a supporter on our supporters page. Elgg supporters are listed there unless they request not to be.

Supporters are able to put this official logo on their site if they wish:

Disclaimer

We operate a no refund policy on supporter subscriptions. If you would like to withdraw your support, go to PayPal and cancel your subscription. You will not be billed the following year.

Being an Elgg Supporter does not give an individual or organization the right to impersonate, trade as or imply they are connected to the Elgg project. They can, however, mention that they support the Elgg project.

If you have any questions about this disclaimer, email info@elgg.org.

We reserve the right to remove or refuse a listing without any prior warning at our complete discretion. There is no refund policy.

If there is no obvious use of Elgg, your site will be linked to with "nofollow" set.

Sign up

If you would like to become an Elgg supporter:

- read the *disclaimer* above
- on the supporters page, subscribe via PayPal
- send an email to info@elgg.org with:
  - the date you subscribed
  - your name (and organization name, if applicable)
  - your website
  - your Elgg community profile

Once all the details have been received, we will add you to the appropriate list. Thanks for your support!

3.6.8 Adding a Service to Elgg

The *services guide* has general information about using Elgg services.

To add a new service object to Elgg:

1. Annotate your class as `@internal`.
2. Open the class `Elgg\Di\ServiceProvider`.
3. Add a `@property-read` annotation for your service at the top. This allows IDEs and static code analyzers to understand the type of the property.
4. To the constructor, add code to tell the service provider what to return. See the class `Elgg\Di\DiContainer` for more information on how Elgg's DI container works.

At this point your service will be available from the service provider object, but will not yet be accessible to plugins.

Inject your dependencies

Design your class constructor to *ask for* the necessary dependencies rather than creating them or using `_elgg_services()`. The service provider's `setFactory()` method provides access to the service provider instance in your factory method.

Here's an example of a `foo` service factory, injecting the `config` and `db` services into the constructor:

```
// in Elgg\Di\ServiceProvider::__construct()

$this->setFactory('foo', function (ServiceProvider $c) {
    return new Elgg\FooService($c->config, $c->db);
});
```

The full list of internal services can be seen in the `@property-read` declarations at the top of `Elgg\Di\ServiceProvider`.

Warning: Avoid performing work in your service constructor, particularly if it requires database queries. Currently PHPUnit tests cannot perform them.

Making a service part of the public API

If your service is meant for use by plugin developers:

1. Make an interface `Elgg\Services\<Name>` that contains only those methods needed in the public API.
2. Have your service class implement that interface.
3. For methods that are in the interface, move the documentation to the interface. You can simply use `{@inheritdoc}` in the PHPDocs of the concrete class methods.
4. Document your service in `docs/guides/services.rst` (this file).
5. Open the PHPUnit test `Elgg\ApplicationTest` and add your service key to the `$names` array in `testServices()`.
6. Open the class `Elgg\Application`.
7. Add `@property-read` declaration to document your service, but use your **interface** as the type, *not* your service class name.
8. Add your service key to the array in the `$public_services` property, e.g. `'foo' => true,`

Now your service will be available via property access on the `Elgg\Application` instance:

```
// using the public foo service
$three = elgg()->foo->add(1, 2);
```

Note: For examples, see the `config` service, including the interface `Elgg\Services\Config` and the concrete implementation `Elgg\Config`.

Service Life Cycle and Factories

By default, services registered on the service provider are "shared", meaning the service provider will store the created instance for the rest of the request, and serve that same instance to all who request the property.

If you need developers to be able to construct objects that are pre-wired to Elgg services, you may need to add a public factory method to `Elgg\Application`. Here's an example that returns a new instance using internal Elgg services:

```
public function createFoo($bar) {
    $logger = $this->services->logger;
    $db = $this->services->db;
    return new Elgg\Foo($bar, $logger, $db);
}
```

3.6.9 Writing tests

Contents

Vision

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 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.

Running Tests

Elgg Core Test Suite

Currently our tests are split in two pieces:

- PHPUnit tests are located in `/tests/phpunit` – these are split between unit tests and integration tests.

Since we have a `phpunit.xml` configuration at the root of Elgg, testing should be as easy as:

```
git clone http://github.com/Elgg/Elgg
cd Elgg
phpunit
```

Plugin tests

Ideally plugins are configured in such a way that they can be unit-tested much like Elgg core. Plugin developers are free to implement their own methods for unit testing, but we encourage everyone to make it as easy as Elgg core:

```
git clone http://github.com/developer/elgg-plugin plugin
cd plugin
phpunit
```

End-to-end tests

Since Elgg plugins have so much power to override, filter, and modify Elgg's and other plugins' behavior, it's important to be able to run end-to-end tests on a staging server with your final configuration before deploying to production.

Note: ToDo: Make it easy to run all Elgg integration and acceptance tests from the admin area given the current plugin configuration. (without worrying about database corruption!)

Motivation

Briefly, the wins we expect from testing are:

- Increased confidence in the system.
- More freedom to refactor.
- Built-in, up-to-date documentation.

We love community contributions, but in order to maintain stability we cannot accept outside contributions without first verifying their correctness. By promoting automated testing, the core developers can avoid the hassle of manual verification before accepting patches. It also means that external developers don't have to spend time earning trust with the core team. If a patch comes in and has tests to verify it, then we can be confident it works without worrying about the reputation of the contributor.

Note that these benefits can also extend to the plugins repository. Site owners are encouraged to "test plugins thoroughly" before deploying them on a production site. As of March 2013, this translates to manually verifying all the features that the plugin promises to offer. But Elgg provides a huge number of features, and it's not reasonable to test for *all* of them on *every browser* you want to support on *every device* you want to support. But what if plugin developers could write tests for their plugins and site owners could just run the tests for all installed plugins to verify the functionality is maintained? Then they wouldn't be limited to just picking plugins from "trusted" developers or "stable" releases. They could see that, indeed, nothing broke when they upgraded that critical plugin from 1.3 to 2.5, and push the upgrade to production with confidence.

The reason this isn't happening today is because Elgg itself is not easily testable at this level yet. We want to change that.

Strategy

We have several guiding principles that we think will be helpful in bringing our vision into reality.

In short, we are advocating:

- Continuous integration – if Travis isn't happy, we're not happy
- Dependency injection – For creating highly testable, modular code

- BDD – Tests should verify features and provide documentation, not rehash the Class API

Continuous Integration

We run all of our tests on Travis CI so that we can get real time feedback on the correctness of incoming pull requests and development as it progresses. **If Travis isn't happy, we don't commit to the repo.** This empowers us to merge pull requests in at a rapid pace, so long as they pass the tests. It also allows us to reject pull requests without detailed investigation if they do not pass the tests. We can get past the "does it work or not" question and talk about the things that humans need to talk about: API design, usefulness to the project, whether it belongs in core or a plugin, etc. We want as many features as possible provided by Elgg core to be verified automatically by tests running on Travis.

Dependency Injection

In order to maximize testability, **all dependencies need to be explicit**. Global functions, Singletons, and service locators are death for testability because it's impossible to tell what dependencies are hiding under the covers, and it's even harder to mock out those dependencies. Mocking is critical because you want your unit tests to test only one class at a time. Test failures in a TestCase should not result due to brokenness in a dependency; test failures should only indicate brokenness in the class under test. This makes everything much easier to debug. As of March 2013, most of Elgg still assumes and uses global state, and that has made Elgg and Elgg plugins historically very difficult to test. Fortunately we are moving in the opposite direction now, and a lot of work in Elgg 1.9 has gone into refactoring core components to be more dependency injectable. We are already reaping the benefits from that effort.

Behavior-Driven Development

For us this means **we name tests for features rather than methods**. When you test for features, you are encouraged to write fewer, smaller, logical tests. When a test fails, we can know exactly what feature is compromised. Furthermore, when naming your tests for features, the list of tests provides documentation on what features the system supports. Documentation is something that is typically very troublesome to keep up to date, but when documentation and verification are one and the same, it becomes very easy to keep the documentation up to date.

Consider these two test methods:

- `testRegister()`
- `testCanRegisterFilesAsActionHandlers()`

From just looking at the names, `testRegister` tells you that the class under test probably has a method named register. If this test passes, it presumably verifies that it is behaving correctly, but doesn't tell you what correct behavior entails, or what the original author of the test was intending to verify. If that method has multiple correct uses that you need to test for, this terse naming convention also encourages you to write a very long test which tests for all conditions and features of said method. Test failure could be caused by any one of those uses being compromised, and it will take more time to figure out where the true problem lies.

On the other hand, `testCanRegisterFilesAsActionHandlers` tells you that there are these things called "actions" that need to be "handled" and that files can be registered as valid handlers for actions. This introduces newcomers to project terminology and communicates clearly the intent of the test to those already familiar with the terminology.

For a good example of what we're looking for, check out `/tests/phpunit/Elgg/ViewServiceTest.php`

3.6.10 Core tasks

Certain tasks surrounding Elgg are reserved for the core developer as they require special permissions. The guides below show the workflow for these actions.

Moving a plugin to its own repository

Contents

Plugin extraction steps

Move the code to its own repository

Follow the GitHub guide Splitting a subfolder out into a new repository. This will make sure that the commit history is preserved.

Dependencies

If the plugin has dependencies on any external libraries, make sure these dependencies are managed. For example if a PHP library is required which comes bundled with Elgg core, make sure to add it to the `composer.json` of this plugin as you can't rely on Elgg core keeping the library.

Commit the code

During the GitHub guide a new repository is created for the plugin you're trying to move.

Since an attemp was already made to extract some of the plugins to their own repository maybe the repository already exists.

If the repository didn't exist for the plugin, make sure you create it under the Elgg organisation.

If the repository already exists, the best way to update the code would be by a Pull Request. This will however probably fail because of a difference in how the repository was first created (as discussed in this GitHub issue).

The initial repositories where created with

```
git subtree split
```

and the guide calls for

```
git filter-branch --prune-empty --subdirectory-filter
```

This will leave a difference in the commits which GitHub is unable to resolve. In that case you'll have to force push the changes to the existing Elgg plugin repository.

> **Warning:** Since this will override all the history in the plugin repository, make sure you know this is what you want to do.

Packagist

Make sure the `composer.json` of the plugin contains all the relevant information. Here is an example:

```
{
        "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>"
        }
}
```

The `conflict` rule is there to help prevent the installation of this plugin in an unsupported Elgg version.

Add the repository to Packagist, for the existing repositories this was already done. Make sure Packagist is updated correctly with all the commits.

Tag a release

In order for Composer to be able to cache the plugin for faster installation, a release has to be made on the repository. Probably the first version that needs to be tagged is the same version as mentioned in the `manifest.xml`. After this development can begin, following the Semver versioning scheme.

Translations

If the translations for the plugin need to be managed by Transifex, add the plugin to Transifex.

Elgg core cleanup

Now that the plugin has been moved to it's own repository, it's time to make a Pull Request on Elgg core to remove the original code.

Remove the plugin

- Delete the `mod` folder for the plugin
- Search for the plugin name in core to find any references which also need to be removed (eg. old docs, special tests, etc.)

Translations

Since the plugin no longer is part of Elgg core, make sure the configuration of Transifex no longer contains the plugin.

Bundled

If the plugin still comes bundled with the release of a new Elgg version, make sure to add the plugin to the `composer.json`.

Composer

Check the core composer dependencies if requirements that were specific for the removed plugin can also be removed in the core dependencies.

Documentation

Add a mention in the *Upgrade Notes* documentation that the plugin was removed from Elgg core.

Release Process Workflow

Release a new version of Elgg.

This is the process the core team follows for making a new Elgg release. We have published this information in the spirit of openness, and to streamline onboarding of new team members.

Contents

Requirements

- SSH access to elgg.org
- Commit access to http://github.com/Elgg/Elgg
- Admin access to https://elgg.org/
- Access to Twitter account
- Node.js and Yarn installed
- Sphinx installed (`easy_install sphinx && easy_install sphinx-intl`)
- Transifex client installed (`easy_install transifex-client`)
- Transifex account with access to Elgg project

Merge commits up from lower branches

Determine the LTS branch. We need to merge any new commits there up through the other branches.

For each branch

Check out the branch, make sure it's up to date, and make a new work branch with the merge. E.g. here we're merging 1.12 commits into 2.0:

```
git checkout 2.0
git pull
git checkout -b merge112
git merge 1.12
```

Note: If already up-to-date (no commits to merge), we can stop here for this branch.

If there are conflicts, resolve them, `git add .`, and `git merge`.

Make a PR for the branch and wait for automated tests and approval by other dev(s).

```
git push -u my_fork merge112
```

Once merged, we would repeat the process to merge 2.0 commits into 2.1.

First new stable minor/major release

Update the *Support policy* to include the new minor/major release date and fill in the blanks for the previous release.

Update the README.md file badges to point to the correct new release numbers.

Prepare the release

Bring your local git clone up to date.

Merge latest commits up from lowest supported branch.

Visit `https://github.com/Elgg/Elgg/compare/<new>...<old>` and submit the PR if there is anything that needs to be merged up.

Install the prerequisites:

```
yarn install elgg-conventional-changelog
easy_install sphinx
easy_install sphinx-intl
easy_install transifex-client
```

Note: On Windows you need to run these command in a console with admin privileges

Run the `release.php` script. For example, to release 1.12.5:

```
git checkout 1.12
php .scripts/release.php 1.12.5
```

This creates a `release-1.12.5` branch in your local repo.

Next, manually browse to the `/admin/site_settings` page and verify it loads. If it does not, a language file from Transifex may have a PHP syntax error. Fix the error and amend your commit with the new file:

```
# only necessary if you fixed a language file
git add .
git commit --amend
```

Next, submit a PR via GitHub for automated testing and approval by another developer:

```
git push your-remote-fork release-1.12.5
```

Tag the release

Once approved and merged, tag the release:

```
git checkout release-${version}
git tag -a ${version} -m'Elgg ${version}'
git push --tags origin release-${version}
```

Or create a release on GitHub

- Goto releases
- Click 'Draft a new release'
- Enter the version
- Select the correct branch (eg 1.12 for a 1.12.x release, 2.3 for a 2.3.x release, etc)
- Set the release title as 'Elgg {version}'
- Paste the CHANGELOG.md part related to this release in the description

Some final administration

- Mark GitHub release milestones as completed
- Move unresolved tickets in released milestones to later milestones

Update the website

- ssh to elgg.org
- Clone https://github.com/Elgg/elgg-scripts

Build zip package

Use `elgg-scripts/build/elgg-starter-project.sh` to generate the .zip file. Run without arguments to see usage.

```
# 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: For Elgg 2.x releases use the `2.x` branch of the starter-project (eg. `./elgg-starter-project.sh 2.x 2.0.4 /var/www/www.elgg.org/download/`)

- Verify that `vendor/elgg/elgg/composer.json` in the zip file has the expected version.
- If not, make sure GitHub has the release tag, and that the starter project has a compatible `elgg/elgg` item in the composer requires list.

Building 1.x zip packages

Use `elgg-scripts/build/build.sh` to generate the .zip file. Run without arguments to see usage.

```
# 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/
```

Update elgg.org download page

- Clone https://github.com/Elgg/community
- Add the new version to `classes/Elgg/Releases.php`
- Commit and push the changes
- Update the plugin on www.elgg.org

```
composer update elgg/community
```

Update elgg.org

- Clone https://github.com/Elgg/www.elgg.org
- Change the required Elgg version in `composer.json`
- Update vendors

```
composer update
```

- Commit and push the changes
- Pull to live site

```
cd /var/www/www.elgg.org && sudo su deploy && git pull
```

- Update dependencies

```
composer install --no-dev --prefer-dist --optimize-autoloader
```

- **Go to community admin panel**
  - Flush APC cache
  - Run upgrade

Make the announcement

This should be the very last thing you do.

1. Open `https://github.com/Elgg/Elgg/blob/<tag>/CHANGELOG.md` and copy the contents for that version
2. Sign in at https://elgg.org/blog and compose a new blog with a summary
3. Copy in the CHANGELOG contents, clear formatting, and manually remove the SVG anchors
4. Add tags `release` and `elgg2.x` where x is whatever branch is being released
5. Tweet from the elgg Twitter account

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 3.1 to 3.2

Contents

User write access

To fix an issue where user owned access collections like *Friends* or *Friend Collections* would still show in the access drop down when creating content, even if related plugins are disabled, we needed to change some internal logic. If you want to have an access collection subtype available in the write access you now need to register the subtype with a plugin hook. See *the plugin hook reference for 'access:collections:write:subtypes'* for more details.

River items enabled state

The *enabled* state of river items has been deprecated. You should no longer use this property when working with river items.

From 3.0 to 3.1

Contents

PHP Version

PHP 7.0 has reached end of life in January 2019. To ensure that Elgg sites are secure, we now require PHP 7.1 for new installations.

If upgrading from a previous Elgg installation make sure you have the correct PHP version installed.

Plugin screenshots

Screenshots added to plugins are no longer supported and will no longer be shown in the plugin details.

Loading external files

The usage of `elgg_register_js`, `elgg_unregister_js` and `elgg_load_js` is discouraged. Make sure your javascript is an AMD module and use `elgg_require_js` to include it.

The usage of `elgg_register_css`, `elgg_unregister_css` and `elgg_load_css` is discouraged. You can register and include css with the new `elgg_require_css` function.

Setting page owner

Setting the page owner via the `elgg_get_page_owner_guid` function parameter is deprecated. Use `elgg_set_page_owner_guid`.

Simpletests

The core simpletests have been removed from the system. They are all replaced by PHP unit tests or integration tests. The simpletest cli command has been deprecated.

Hook and event callbacks

The legacy style hook and event callback arguments are deprecated. You should switch to the new style as soon as possible.

```
// old style hook callback
function hook_callback($name, $type, $return_value, $params) {

}

// new style hook callback
function hook_callback(\Elgg\Hook $hook) {
        // now you can use a few new functions
        $params = $hook->getParams();
        $return = $hook->getValue();

        $specific_param = $hook->getParam('specific_param', 'default')
        $entity = $hook->getEntityParam();
        $user = $hook->getUserParam();
}

// old style event callback
function event_callback($name, $type, $object) {

}

// new style event callback
function event_callback(\Elgg\Event $event) {
        // now you can use a few new functions
```

(continues on next page)

(continued from previous page)

```
        $object = $event->getObject();
}
```

Deprecated Routes

- `previous:object:thewire` This route was not in use. It now has been marked as deprecated.

Deprecated CSS libraries

- `jquery.imgareaselect` Do not use this external css file.
- `jquery.treeview` Do not use this external css file.

Deprecated JS libraries

- `elgg.avatar_cropper` Do not depend on this external javascript library.
- `jquery.imgareaselect` Do not depend on this external javascript library.
- `jquery.treeview` Do not depend on this external javascript library.

Deprecated APIs

- `access_show_hidden_entities()` Use `elgg_call()` with `ELGG_SHOW_DISABLED_ENTITIES` flag.
- `autoregister_views()`
- `count_messages()` Use `elgg()->system_messages->count()`.
- `disable_user_entities()`
- `elgg_enable_entity()` Use `ElggEntity::enable()`.
- `elgg_get_file_list()` Use a PHP directory iterator.
- `elgg_instanceof()` Use PHP instanceof type operator.
- `elgg_is_admin_user()` Use `ElggUser::isAdmin()`.
- `elgg_set_ignore_access()` Use `elgg_call()` with `ELGG_IGNORE_ACCESS` flag.
- `elgg_sort_3d_array_by_value()`
- `get_access_list()` Use `get_access_array()`.
- `get_language()` Use `get_current_language()`.
- `get_number_users()` Use `elgg_count_entities()`.
- `pages_register_navigation_tree()`
- `ini_get_bool()`
- `is_not_null()`
- `update_access_collection()`

Deprecated actions

- `admin/delete_admin_notice` Replaced by generic entity/delete action.
- `avatar/crop` Handled in avatar/upload action.
- `avatar/remove` Handled in avatar/upload action.
- `blog/delete` Replaced by generic entity/delete action.
- `messages/delete` Replaced by generic entity/delete action.
- `site_notifications/delete` Replaced by generic entity/delete action.

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:

```
{
        "require": {
                "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:

```
{
        "require": {
                "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

- *Improved `elgg/ckeditor` module*

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
```

(continues on next page)

(continued from previous page)

```
  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 {
```

(continues on next page)

(continued from previous page)

```
        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',
);
```

Warning: 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
```

(continues on next page)

(continued from previous page)

```
 * @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');
}

/**
 * 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);
```

(continues on next page)

(continued from previous page)

```
    // 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;
}
```

Warning: 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(
```

(continues on next page)

(continued from previous page)

```
        $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,
```

Warning: 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
```

(continues on next page)

(continued from previous page)

```
 */
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
 *
 * @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}";
}
```

Warning: 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

Warning: 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

See also:

Getting Help

"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 causes

- 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 {
```

(continues on next page)

(continued from previous page)

```
        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.

See also:

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?

> **Warning:** 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.

Warning: 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 *Duplicate 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.

> **Warning:** 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.

General

See also:

Getting Help

"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 causes

- 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.';

```

(continues on next page)

(continued from previous page)

```
$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.

See also:

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?

> **Warning:** 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.

> **Warning:** 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 *Duplicate 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.

Warning: 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
```

(continues on next page)

(continued from previous page)

```
[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 *Contributor Guides* 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.

Warning: 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.

See also:

- *Release Policy*
- *Reporting Issues*

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	April 2019	July 2019	
3.1	July 2019	October 2019	
3.2	October 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.