Cohesion Documentation

Release 0.1

Adric Schreuders

December 09, 2015

Contents

The primary objective of this framework is to provide a simple framework to help developers create good clean PHP code. It forces developers to write code that follows many common software development principles.

Cohesion is more than just a Framework, it's a development guideline that tries to enforce best practices to improve maintainability.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 Origins

Cohesion started from me deciding to start a new project in PHP. I didn't have much experience with many PHP frameworks so rather than looking through all the existing frameworks and picking the one that seemed to make the most sense to me, I decided to list all the 'features' I wanted in a framework before trying to find one suitable.

Coming from a long history of programming in a myriad of languages, I realised that one of my main requirements was to have a strong code structure to provide a strong backbone of the application so that it's very easy to maintain in the future. I have managed large teams of software engineers maintaining large code bases and I know how painful it can be to add new features or change existing ones when every engineer decides to implement his own structure.

In going through all the existing frameworks I found that none of them really gave me what I wanted. Probably the closest ones would be Symfony and Laravel but still they didn't really give me everything I needed. So I set out to develop a simple framework that will give me everything I need and will help make sure that I will be able to continue to easily add more features in the future even when my project has grown to hundreds of classes and will be able to easily go in and optimize the data access etc without having to worry about effecting any business logic etc.

One of the issues I have with many of the other frameworks is that they provide a good foundation for developing "anything" but then let the developers do whatever they want on top of it. They pride themselves as being extremely customizable and letting users choose how they want to use it. While that's fantastic for many people, I want something that will provide more than a foundation, I don't want other people to "use it how ever they want". In going with this theme I've tightened "how" Cohesion can be used down a bit and don't have as many "options".

1.1.2 Philosophy

Cohesion is for people who envision their product continuing to grow and place a high value on maintainability and low technical debt.

Cohesion tries to bring many of the well established development "best practices" into the main development pipeline for PHP projects.

I'm sure Cohesion won't be for everyone, as most people are happy with something they can quickly hack a website on top of and continue just quickly hacking stuff on top.

It provides the framework for you to create a service based architecture for your applications.

Principles

High Cohesion High Cohesion is when objects are appropriately focused, manageable and understandable. It basically means that everything within the same component should be strongly related to each other.

Low Coupling A loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. Not relying on the implementation of other components means that if any individual component should change then it shouldn't effect any other component.

Single Responsibility The single responsibility principle states that every context (class, function, variable, etc.) should have a single responsibility, and that responsibility should be entirely encapsulated by the context. All its services should be narrowly aligned with that responsibility.

Dependency Inversion Components should "use" abstractions of other libraries / utilities / modules. Then the concrete implementation can be passed to the object as run time. This allows for the interchanging of implementations without having to deal with modifying other components that "use" the class.

Controller The Controller is a component often referenced as a part of MVC frameworks. It's role is to separate the Business Logic and the Presentation Logic.

Creator The factory pattern allows for a specific class that is responsible for creating new instances of objects. This allows for the code to delegate creating of instances meaning that each module doesn't need to know "how" to create an instance of that object.

1.2 Installation

This documentation currently focuses on installing Cohesion on a Debian/Ubuntu system.

1.2.1 Requirements

Once PHP is installed and set up adding additional libraries will be handled by Composer but there are still a few things you need to install yourself before then.

- PHP version 5.4 or above
- PHP-FPM
- MySQLnd extension
- APCu extension
- CURL extension

Installing PHP and required extensions

```
sudo apt-get install php5 mysql-client php5-mysqlnd php5-fpm php5-curl php-pear php5-dev
```

Install APCu for fast access caching:

```
sudo pecl install apcu
```

You'll most likely have to add the extension to your php.ini:

```
sudo echo "extension=apcu.so" > /etc/php5/mods-available/apcu.ini
sudo ln -s /etc/php5/mods-available/apcu.ini /etc/php5/conf.d/20-apcu.ini
sudo ln -s /etc/php5/mods-available/apcu.ini /etc/php5/fpm/conf.d/20-apcu.ini
```

The paths may be different depending on your distribution

Make PHP-FPM use a Unix socket

By default PHP-FPM is listening on port 9000 on 127.0.0.1 in some distributions. You should make PHP-FPM use a Unix socket which avoids the TCP overhead. To do this, open `/etc/php5/fpm/pool.d/www.conf`:

```
sudo vi /etc/php5/fpm/pool.d/www.conf
```

Find the existing `listen` line and comment it out and add the new one as shown here:

```
[...]
;listen = 127.0.0.1:9000
listen = /var/run/php5-fpm.sock
[...]
```

1.2.2 Installing Composer

Composer is the package manager used by modern PHP applications and is the only recommended way to install Cohesion. To install composer run these commands:

```
curl -sS https://getcomposer.org/installer | php
sudo mv composer.phar /usr/local/bin/composer
```

1.2.3 Installing Nginx

It's recommended to use Nginx with Cohesion. Nginx is a very lightweight web server and is much more efficient than Apache and very easy to configure:

```
sudo apt-get install nginx
```

1.2.4 Installing Cohesion with Composer

Once composer is installed run the following command to create a new project in a `myproject` directory using Cohesion:

```
composer create-project cohesion/cohesion-framework -sdev myproject
```

Composer will then download all required dependencies and create the project directory structure for you.

After composer finishes the installation process the installer will ask you `Do you want to remove the existing VCS (.git, .svn..) history? [Y,n]?` just hit `<Enter>` to safely remove the Cohesion git history. This will prevent you from polluting your projects version history with Cohesion commits. It will also make it easier to set up your own version control for your project.

1.2.5 Setting up Nginx server

Default Nginx configurations are provided within the `/config/nginx/` directory of your project. So that we can keep our server configurations in version control we'll just link to the configuration file you want to use for the current environment within our project:

```
sudo ln -s /full/path/to/myproject/conf/nginx/local.conf /etc/nginx/sites-available/myproject-local
sudo ln -s /etc/nginx/sites-available/myproject-local /etc/nginx/sites-enabled/myproject
```

Note: Your nginx directory might be somewhere else depending on your distribution.

Open up the configuration and set the `root` path to the `www` directory within your project:

```
vi config/nginx/local.conf
```

You can create additional Nginx configuration files for your different environments just remember to change the `server_name`, `root` path and `APPLICATION_ENV`.

Restart nginx:

```
sudo service nginx restart
```

Now you should be able to see a default Cohesion welcome page when you go to http://localhost.

1.3 Quickstart

After installing a clean Cohesion project you should modify your composer.json file and set the project name etc for your project.

You main application code will go into the `src/` directory. You should create separate subdirectories within that folder for each of your services.

All front end code goes within the `www/` directory. With your controllers in `www/controllers`, views in `www/veiws` and templates in `www/templates`.

1.3.1 Frontend

If you haven't changed your configuration yet your default controller should be 'Home' with the default function on 'index'. This means that if you go to http://localhost/ it should look for the `HomeController` and execute the `index` function on it.

The Cohesion Framework comes with a very simple home page. Open up `www/controllers/HomeController.php` you should see it looks like this:

```
use \Cohesion\Structure\Controller;
use \Cohesion\Structure\Factory\ViewFactory;

class HomeController extends Controller {

    public function index() {
        $view = ViewFactory::createView('Home');
        return $view->generateView();
    }
}
```

Here we can see it's creating a 'Home' view. This can be found at `www/views/HomeView.php`.

```
class HomeView extends MyView {
    public function __construct($template, $engine, $vars) {
        parent::__construct($template, $engine, $vars);
        $vars['page'] = 'home';
        $vars['title'] = $vars['site_name'] . ' Home';
```

```
        $this->addVars($vars);
    }
}
```

As you can see this is extending `MyView` class. All your views should extend that class (feel free to rename it) and that is where you can put any common logic that all your views should have access to.

Next we see it's calling it's parent constructor and passing through the template engine and variables. Don't worry too much about this, this is just setting up the templating engine.

The last thing in this file we see is that it's adding a new variable, 'page' and setting it to 'home'. You can add as many variables as you want here and they will all be available within the front end template. **page** is a special variable though and must be set in every view. The variable set in `page` is used to decide which template to use. In this case it will be using the `home.html` template. `title` is the variable that will be used for the title of this page. You can see here it's using one of the default variables set in the configuration 'site_name'.

Let's look at the root template `www/templates/index.html`. All pages use the same template by default. It's possible to override this behaviour in the `View` class but most of the time you'll probably want all your pages to have the same header and footer.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>{{title}}</title>
    </head>
    <body>
        {{> header}}
        {{> page}}
        {{> footer}}
    </body>
</html>
```

As you can see this template is extremely simple and you'll probably want to fill it out much more. Here you can start to see the Mustache templating library in action. `{{title}}` will be replaced with the `title` set in the view.

In Mustache the > modifier is used to load a sub template (partial). In Cohesion the partial template name can be overwritten by setting a variable with that name. Since we set the variable `page` to `home` it will load the `www/templates/home.html` template in the middle. But the header and footer weren't overwritten so they will just use the templates `www/templates/header.html` and `www/templates/footer.html`. So if you modify the header template it will update the header on every page then we can just change the `page` variable to set the main content of the page.

1.3.2 Backend / Services

Your main code for all your business logic will go into folders within the `src/` directory.

Let's run through creating a user.

We'll start by creating a directory `src/user/` and adding a `User.php` file to it as our DTO:

```
use \Cohesion\Structure\DTO;

class User extends DTO {
    protected $id;
    protected $username;
    protected $email;
    protected $password;
```

```
    const MIN_USERNAME_LENGTH = 3;
    const MAX_USERNAME_LENGTH = 30;
    const MIN_PASSWORD_LENGTH = 6;

    public function setUsername($username) {
        if (strlen($username) < self::MIN_USERNAME_LENGTH) {
            throw new UserSafeException('Username must be at least ' . self::MIN_USERNAME_LENGTH . '
        } else if (strlen($username) > self::MAX_USERNAME_LENGTH) {
            throw new UserSafeException('Username cannot be longer than ' . self::MAX_USERNAME_LENGTH
        }
        $this->username = $username;
    }

    public function setEmail($email) {
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new UserSafeException('Invalid email address');
        }
        $this->email = $email;
    }

    public function getUsername() {
        return $this->username;
    }

    public function getEmail() {
        return $this->email;
    }
}
```

That's all that's needed for now. The constructor of the parent `DTO` class will be able to take in an associative array and call the appropriate mutator functions. And also provides a `getVars()` function that will return the protected variables as an associative array.

Next lets create a `UserService` at `src/user/UserService.php`. The service is the external API for everything other components need to do with "users". Normally it would include a bunch of authorization code etc but for now let's just keep it simple.

```
use \Cohesion\Structure\Service;
use \Cohesion\Auth\AuthException;

class UserService extends Service {
    public function getUser($id) {
        return $this->dao->getUser($id);
    }

    public function getUserByUsername($username) {
        return $this->dao->getUserFromUsername($username);
    }

    public function createUser($user) {
        $user = $this->dao->getUserFromUsername($user->getUsername());
        if ($user) {
            throw new AuthException("Username {$user->getUsername()} is already taken");
        }
        try {
            $this->dao->createUser($user);
        } catch (DBException $e) {
            throw new AuthException('Unable to register new user');
```

```
        }
        return $user;
    }
}
```

The constructor of the parent `Service` will create the `dao` for us and will look for a class named `UserDAO` so let's create that now. `src/user/UserDAO.php`:

```
use \Cohesion\Structure\DAO;

class UserDAO extends DAO {

    public function getUser($id) {
        $result = $this->db->query('
            SELECT u.id, username, email
            FROM users
            WHERE id = {{id}}
            ', array('id' => $id));
        if ($row = $result->nextRow()) {
            return new User($row);
        } else {
            return null;
        }
    }

    public function getUserFromUsername($username) {
        $result = $this->db->query('
            SELECT u.id, username, email
            FROM users
            WHERE username = {{username}}
            ', array('username' => $username));
        if ($row = $result->nextRow()) {
            return new User($row);
        } else {
            return null;
        }
    }

    public function createUser(&$user) {
        $result = $this->db->query('
            INSERT INTO users
            (username, email, password)
            VALUES
            ({{username}}, {{email}}, {{password}})
            ', $user->getVars());
        $user->setId($result->insertId());
        return $user;
    }
}
```

The parent `DAO` constructor will set the `db` for us so we don't need to worry about getting the database connection. If we wanted to be able to access other libraries from within the `DAO` we can override the constructor and define the extra data accesses we need.

For example if we wanted to use a cache we could add a constructor like this:

```
use \Cohesion\Structure\DAO;
use \Cohesion\DataAccess\Database\Database;
use \Cohesion\DataAccess\Cache\Cache;
```

```
class UserDAO extends DAO {

    protected $cache;

    public function __construct(Database $db, Cache $cache) {
        parent::__construct($db);
        $this->cache = $cache;
    }

    public function getUser($id) {
        $userData = $this->cache->get("user_$id");
        if (!$userData) {
            $result = $this->db->query('
                SELECT u.id, username, email
                FROM users
                WHERE id = {{id}}
                ', array('id' => $id));
            if ($userData = $result->nextRow()) {
                $this->cache->save("user_$id", $userData);
            }
        }
        if ($userData) {
            return new User($userData);
        } else {
            return null;
        }
    }
    // ...
}
```

We don't need to change any code anywhere else other than making sure that the Cache is set up properly in the configuration files.

1.4 Code Structure

Cohesion lays out the ground work for an extended MVC set up. I say extended because it goes one step further at separation of responsibilities by splitting the 'Model' into the **Business Logic**, **Object Data** and **Data Access Logic**.

For each "object" that you will use in your application you should create an associated service and data access object. For example our `User` object will just be a dumb object and all associated business logic should be done within the `UserService`. The `UserService` will then use it's `DAO` to access and store related user data.

If you are developing an application with many module you can create a service hierarchy where the high level services will be the high level business logic for the module and it will make use of the individual object services for the object specific business logic.

Nothing within any Service, DTO, or DAO should have any idea about the environment in which it's being used. This means that we should be able to reuse the same services for web application, command line scripts, and anything else that may need to reuse the same business logic such as mobile application back ends etc.

1.4.1 Services - Business Logic

Services contain all the business logic and only the business logic for a specific object. I've called the business logic section 'Services' because I want people to think of them as an independent Service for one portion of the application. The service will contain all the authorisations and validations before carrying out an operation. Services are the entry

point for the DAOs, where only the UserService accesses the UserDAO. The UserService can be thought of as the internal API for everything to do with Users.

1.4.2 Data Transfer Objects (DTOs) - Object Data

DTOs are simple instance objects that contain the data about a specific object and don't contain any business logic or data access logic. DTOs have accessors and mutators (getter and setters) and *can* contain minimal validation in the mutators.

1.4.3 Data Access Objects (DAOs) - Data Access Logic

DAOs contain all the logic to store and retrieve objects and data from external data sources such as RDBMSs, non relational data stores, cache, etc. The DAO does not contain any business logic and will simply perform the operations given to it, therefore it's extremely important that all accesses to DAOs come from the correlating Service so that can perform the necessary checks before performing the storage or retrieval operations. The Service doesn't care how the DAO stores or retrieves the data only that it does. If later down the line a system is needed to be converted from using MySQL to MongoDB the only code that should ever need to be touched would be to within the DAOs.

1.4.4 Controllers

The Controllers are the external facing code that access the input variables and returns the output of the relevant view. The Controller handles the authentication, accesses the relevant Service(s) then constructs the relevant view. It doesn't contain any business logic including authorisation.

1.4.5 Views

Views take in the required parameters and return a rendering of what the user should see. Views don't perform any logic and don't do any calls to any function to get additional data. All data that the view will need should be provided to the view from the Controller.

1.5 Configuration

The configuration is a big part of the backbone of Cohesion. It contains all the information for the *Dependency Injection* and how each utility should be used.

Configuration files are JSON formatted and can include comments. The configuration files are set up in a cascading fashion where loading subsequent configurations will overwrite just the values that are specified in that config and the unspecified configurations are left unchanged.

The default Cohesion configuration is located at `myproject/config/cohesion-default-conf.json`. It is not recommended to make any changes to this file directly as it may make it harder to resolve any conflicts etc if we update it. Instead you should put all your default configurations in the `myproject/config/default-conf.json` file. Remember you don't need to implement all the settings, only add the ones that you want to do differently from the cohesion defaults.

1.5.1 File Structure

The configuration is very well structured and documented so make sure you take the time to read the comments in the cohesion default configuration file about what each setting does.

When constructing various objects and libraries they will be given a sub section of the configuration so it's important to get the structure right.

All objects and libraries will have access to the `global` section.

Each Service that you create will get a copy of the `application` section. The view and templating library will get the `view` section. The database class will use a copy of the `data_access.database`. And so on and so forth.

1.5.2 Environment specific configurations

Cohesion can load an additional separate configuration for different environments such as different database settings, etc for your dev, staging and production environments. These are just examples you can have separate configuration for whatever different environments you might have. Simply create additional configuration files in the form `{environment}-conf.json`. For example, `local-conf.json` or `production-conf.json`.

Again these are cascaded so you only need to include the settings that are different for that environment and it will get the rest of the settings from your `default-conf.json` and the `cohesion-default-conf.json`.

1.5.3 Configuration Options

global

Global configurations are settings that may effect multiple areas of the code. Global should only be used as a last resort if the configuration doesn't fit in better in any of the other setting sections. Options provided within the global section will be available to all Configurable objects.

Default global settings

Key	Type	Default	Description
global.domain_name	string	'www.example.com'	Domain name of your application. Used for setting link URLs as well as other places. This will be overwritten by `$_SERVER['HOST']` if it's set but it's important to still set this as it will be used in things like sending emails from the command line etc that don't have access to the host parameter
global.production	boolean	`false`	Whether or not the current system is being used in production. This is used for things such as setting the error output level as well as other business logic determinant on whether it's in production or not.
global.autoloader_cache_key	string	'autoloader_cache'	Cache key used to cache the class paths for your files.

application

The application configuration section is what will be passed to your business logic classes. This is where you should be placing your own configurations for your application settings and should contain all your business rule settings.

Default application settings

Key	Type	Default	Description
class.default	string	*null*	Default service to use if the `ServiceFactory::getService()` function isn't provided a class name. The default of `null` means that you must specify the service as there's no default.
class.prefix	string	''	Prefix used in the service class name before the service name. For example if the prefix was "Application" and you called `ServiceFactory::getService('User')` it will look for a `ApplicationUser` class.
class.suffix	string	'Service'	Suffix used in the service class name after the service name. For example with the default suffix of "Service" when you call `ServiceFactory::getService('User')` it will look for a `UserService` class.

routing

The routing configuration section is used to decided how to deal with incoming requests and decide which Controller should be used to handle it.

We currently don't provide a way to specify specific routes and how they will be handled. Rather Cohesion works on a simple yet effective default route system.

It will look for the first path segment that is a valid Controller then check if the following path segment is a function of that Controller, if it is then it will call that function with all the following path segments as parameters to that function, if it's not a valid function then all segments after the Controller will be considered parameters to the Controller's index function.

Given the path `/nothing/something/action/param1/param2?foo=bar` "nothing" is not a Controller and will be ignored. `something` matches the controller `SomethingController` (based on the class prefix and suffix) and therefore that Controller will be used. `action` is a function of `SomethingController` therefore the resulting call based on a request to that URI will be: `SomethingController->action('param1', 'param2')`. To access the GET and POST parameters you can to use `$this->input->get('foo');`

For the home page (route '`/`') the default controller will be used with the default function.

Default routing settings

Key	Type	Default	Description
class.default	string	'Home'	Default controller to use if no other Controller is found in the path.
class.prefix	string	''	Prefix to use when matching the Controller class name with the path.
class.suffix	string	'Controller'	Suffix to use when matching the Controller class name with the path. The default of "Controller" means that `something` will try to match the `SomethingController`
function.default	string	'index'	Default function to use within the matched Controller when no other function is found on the path.
function.prefix	string	''	Prefix to use when matching the Controller function name with the path.
function.suffix	string	''	Suffix to use when matching the Controller function name with the path. For example if this is set to "Action" and the path had `view` then it would look for the `viewAction` function within the Controller.
redirects	mappings	`{ "^/favicon.ico$": "/assets/images/favicon.ico" }`	Redirects can be used to send a redirect. Each of the keys within the redirects configuration are used as regular expressions to be used to match the path. If the regex matches the page will be redirected to the value here.

view

The view configuration section defines how the views will be generated.

Default view settings

Key	Type	Default	Description
class			Used to define the class names so that the ViewFactory can create the views based on just the base name. Eg, 'home' will create a HomeView instance.
class.default	string	''	Default view to use when no view name is given to the ViewFactory. The default of `''` means that it will default using the prefix and suffix. If you don't want it to use any default set this to *null*.
class.prefix	string	''	Prefix used in the view class name before the view name. For example if the prefix was "My" and you called `ViewFactory::createView('page')` it will look for a `MyPage` class.
class.suffix	string	'View'	Suffix used in the view class name after the view name. With the default value a call to `ViewFactory::createView('page')` will look for the `PageView` class.
template.engine	string	'CohesionTemplatingCohesionMo'	The default templating engine is basically an extended version of Mustache with a couple of added functionalities. You can change this to use a different templating engine but the engine must implement the TemplateEngine interface. So if you want to use a different currently unsupported template engine you can wrap it in a new class that implements the interface.
template.directory	string	'www/templates'	Root template directory containing all your template files.
template.extension	string	'html'	Extension used on your template files.
template.default_layout	string	'index'	Default template file to use which contains the root layout of all your pages.
template.vars	mapping	*see file*	Set of default variables to be available in every template.
cache	mixed	`{ "ttl": 3600 }`	To cache prerendered templates set either a directory or a ttl. If you set the directory it will store the prerendered templates on disk within that directory. The directory can either be a relative path to the base directory or an absolute path starting from '/'. Make sure the application server has write access to the directory. If you don't set a directory it will use the default system cache and you can set the `ttl` for the cache keys. Setting `cache` to `false` will disable caching although this is not recommended.
cdn			Using Content Delivery Network will usually increase the speed of viewing your site as well as drastically reducing server load as all your static assets will be cached on a third party server that usually have end points all around the globe.
cdn.hosts	array	`[]`	Hosts is an array of fully qualified domain names to use. It's recommended that you use multiple CDN domains as nearly all browsers have a limit of concurrent connections to the same domain. Adding multiple CDN hosts will allow clients to download more assets concurrently. If no hosts are set then CDN will be disabled and all requests will go to the application server.
cdn.version			Versioning is used to invalidate assets when they change. Because the CDN will cache your static assets they will need a new resource name every time you update your files, such as updates to JavaScript files etc. It's important that if your CDN asks you whether or not to include the query string you include it. File versions are basically just an MD5 check sum of the file. To improve performance the version is stored in the local cache and is only re-evaluated after the ttl has expired.
cdn.version.cache_prefix	string	'set_version_'	Prefix to use when caching the versions into the local cache.
cdn.version.ttl	int	300	Expiry of the cached version in seconds. *300* is 5 minutes
formats	object	*see file*	These are the accepted formats and the view class that matches them If you want to use your own default view for HTML you should overwrite it

data_access

The data_access configuration section contains each of the data access types that your application will use. The key should match a data access interface or class. If a driver is specified it will use that class otherwise it will try to use the setting value.

`public SomethingDAO(Database $db)` will use the database setting and instantiate the driver (MySQL).

default data access settings

Key	Type	Default	Description
class.default	string	`null`	Default Data Access Object to use when no name is given to the `DAOFactory::getDAO()` function. `null` means that you must supply a DAO name whenever calling that function.
class.prefix	string	''	Prefix for DAO classes.
class.suffix	string	'DAO'	Suffix for DAO classes.
database			Database settings.
database.driver	string	'MySQL'	Driver class name used to connect to the database.
database.host	string	'localhost'	Database host for the main database connection used for writes.
database.port	int	3306	Database port to use when connecting to the host.
database.user	string	'root'	User used when connecting to the host.
database.password	string	''	Database password.
database.database	string	'cohesion'	Default database used when connected to the server.
database.charset	string	'UTF8'	Default character set to use.
database.slave			The slave is used in a master-slave database set up. You can override the user, password and database settings here if you want to use different ones for the slaves, otherwise it will use the same ones.
database.slave.hosts	array	`[ 'localhost' ]`	Array of slave hosts to use for read only statements. If none are set then it will just use the same as the main.
database.slave.user	string	*unset*	Set this if you want to use a different user to connect to the slaves. The same user will be used for all slaves.
database.slave.password	string	*unset*	Set this if you want to use a different password to connect to the slaves.
database.slave.database	string	*unset*	Set this if you want to use a different database on the slaves.
cache			Default cache to use for system caching.
cache.driver	string	'APC'	Class used for system caching.

object

The object configuration section sets up the basic objects. There shouldn't really be any reason to add anything extra in here as objects are just dumb classes that store data.

default object settings

Key	Type	Default	Description
class.prefix	string	''	Prefix used for DTO classes.
class.suffix	string	''	Suffix used for DTO classes. The default of no prefix or suffix means that the 'user' DTO will be the 'User' class.

utility

The utility configuration section provides the config used by the various utility classes. Each of these sections are optional and only required if you want to use the utility.

1.6 Dependency Injection

1.6.1 Dependency Injection simplified

If you're not aware of what dependency injection is you can read about it on Wikipedia. But if you do you might get a bit scared off because they make it sound quite complex when it's really actually quite simple. The idea is that rather than hard coding the other classes your class will use it takes them in via the constructor parameters or a setter method.

For example rather than in your class creating a MySQL connection to talk to the database you can specify that your class needs an instance of something that implements the `Database` interface and then it will be up to the code that calls your class to pass in the instance.

1.6.2 How Cohesion performs Dependency Injection

In Cohesion we've gone one step further and abstracted away creating the instances anywhere within your code and is instead up to the configuration to supply the information needed to decide how to create instances that implement the interface you need.

So basically you can just say, I need a `Cache` library, then in the config you can decide whether to use APC, Memcache, Redis, or whatever. And at any point you can change which type of cache you're using just by updating your configuration.

The dependency injection relies heavily on PHP Reflection to inspect the object you want to create, look at the parameters the constructor is asking for, get the required library based on the config and pass in an instance. This is the job of the Factories.

Creating a Service

The Service Factory is used to create instances of services for your application to use. It makes sure that the service knows who the current user is and passes in the application configuration. When a service is created the Service constructor uses the Data Access Factory to create a DAO for that Service. For example, the `UserService` should only ever use the `UserDAO` and shouldn't have access to any other DAOs therefore when you create a `UserService` it will set the object parameter `dao` to an instance of the `UserDAO`.

Creating DAOs

DAOs are created using the Data Access Factory. It will inspect the constructor and pass in the accesses it needs. DAOs can reference other DAOs and can specify them in the constructor.

```
class UserDAO extends DAO {
    protected $cache;
    protected $locationDao;

    public function __construct(Database $db, Cache $cache, LocationDAO $locationDao) {
        parent::__construct($db);
        $this->cache = $cache;
        $this->locationDao = $locationDao;
```

```
    }
    ...
}
```

Creating Utilities

If a utility implements the `Util` interface you can create instances of that utility using the Util Factory which will construct that utility using the configuration set in the config.

1.7 Routing

1.7.1 Route matching

The current routing doesn't get specified but instead in inferred by the path, class and function names of the Controllers.

Based on `/{directory}/../{controller}/{function}/{var}`. For example `/user/list/all` would call `UserController->list('all')`.

hyphens (`-`) in the path are used as word separators so that it can match the correct capitalisation of controller and function names. `/contact-us` will match `ContactUsController`.

Using defaults

It will also accept variations that use the default values. It will first try to match a controller, if it doesn't match a controller then it will try to match a function on the default controller. Also if the function is omitted it will use the default function of the selected controller. The defaults can be configured in the config.

By default `/` will match the default controller's default function, `HomeController->index()`.

`/about` will first look for an `AboutController` if it exists it will use the default `index()` function on that controller. If that controller doesn't exist then it will try to use the `about()` function on the default `HomeController`. If that function doesn't exist it will throw a `NotFound` error.

Parameters

The function must be able to accept the number of parameters on the path otherwise it will throw a `NotFound` error. If the parameter to the function is optional (has a default value) then that parameter may be left off the path.

For example the path `foo/bar/abc/def` could match `FooController->bar('abc', 'def')` but if the `bar()` function can't accept two arguments then it will fail.

Here's an example of a controller class and the url -> call matches.

```
class FooController extends Controller {
    function index($param1, $param2, $param3 = null) {
        // ...
    }

    function bar($param1, $param2) {
        // ...
    }
}
```

- `/foo/abc/def/ghi` -> `FooController->index('abc', 'def', 'ghi');`

- `/foo/abc/def/` -> `FooController->index('abc', 'def', null);`
- `/foo/abc` -> 404 NotFound
- `/foo/bar/abc/def` -> `FooController->bar('abc', 'def');`
- `/foo/bar/abc/def/ghi` -> 404 NotFound
- `/foo/bar/abc` -> 404 NotFound

Note: That last one could actually match `FooController->index('bar', 'abc')` but because `bar` matched a function it doesn't fall back to the index. This is important to note so that you take this into consideration when naming your functions and what data you could be expecting.

Sub directories

Cohesion will itterate through the URI components and will first try to find a controller with the name of the component, then it will look to see if there's a directory with that name and move on to the next URI component.

- `/some/path/foo/bar/abc/def` will match the file `controllers/some/path/FooController.php` and run `FooController->bar('abc', 'def');`

1.7.2 Redirects

Redirects can be specified in the config. Each of the keys within the redirects configuration are used as regular expressions to match on the path. If the regex matches the page will be redirected to the value here.

For example this will match the default favicon most browsers will look for and redirect to the icon in the `assets/images` directory

```
{
    "routing": {
        "redirects": {
            "^/favicon.ico$": "/assets/images/favicon.ico"
        }
    }
}
```

1.8 Templating

Cohesion uses the Mustache templating language (thanks to @bobthecow's implementation) by default and is initialised with a CohesionMo class that set's up some additional useful functionalities for you. The reason for choosing Mustache was to eradicate the ability to perform any logic within the View. It's recommended that you read through the Mustache documentation on the Syntax. The implementation of Mustache being used is extremely efficient as it will pre-compile the templates to PHP code and store it in fast access cache.

1.8.1 Additional functionalities

Variable partials

Partials will check to see if there's a variable with that name first, if there is then the content of that variable will be used, otherwise it will revert to the standard behaviour of using the name as the partial. This means you have to take this into consideration when naming your variables and partials.

If you want to load a template named header.html then a template who's name is set by the variable $content then the footer.html template, you can simply do this:

```
{{> header}}
{{> content}}
{{> footer}}
```

Site URLs

A default lambda is added to generate site URLs for you. Never hard code your URLs or use relative paths. Always use this lambda when generating any links to pages within your site.

```
<a href="{{#site_url}}about{{/site_url}}">About Us</a>
```

Asset URLs

A default lambda has also been added for generating URLs to your assets. Rather than using the site URL to access your assets you should always use the asset lambda which will allow for using CDNs for your static assets. Even if you don't wish to use a CDN for now it's a good practice to still use this lambda for your assets and if no CDN hosts are defined in your config it will just revert to the site URL. Then when you decide you need to relieve the load on your sever you can set up a CDN and just modify your config to point to that.

```
<script src="{{#asset_url}}{{asset_path.javascript}}main.js{{/asset_url}}"></script>
```

Asset versioning

The asset URLs will also contain a "version" as a parameter in the query string which is basically just an md5 checksum of the file. This is used to invalidate the CDN cache whenever you release an update to any of your static assets. The above example will render as:

```
<script src="http://cdn.example.com/assets/js/main.js?v=365424d18c0e435388a859592b8f5e3e"></script>
```

It's very important that when setting up your CDN you tell it to include the query string in the cache key, otherwise you will have to manually invalidate your assets every time you update them.

Don't worry though, we're not recomputing the md5 checksum of every static asset on every page load, these are stored in the local cache (APC) and only updated when the ttl on the cache expires.

1.9 Error Handling

All errors should throw exceptions and only be caught at a point in the code that is equipped to handle the exception.

1.9.1 Exceptions

You should create your own exceptions for your application so that they can be handled appropriately. Cohesion specifies a `UserSafeException`. You should use an implemenation of a `UserSafeException` for any errors that are safe for the user to see. This includes things such as invalid input etc. The message of `UserSafeExceptions` are displayed back to the user.

1.9.2 Error Output

When an exception isn't caught within the code it will generate an error page. Different exceptions may generate different error Views. A `NotFoundException` will show a 404 page, an `UnauthorizedException` will show a 403 page, and other exceptions will show a 500 `ServerError` page. If the environment is set to 'production' non `UserSafeExceptions` will just generate a vague error message. But if the environment isn't 'production' then it will show the exception error message as well as the stack trace for easier debugging.

If there's an uncaught exception in an AJAX call it will determine the expected format and return a valid response in that format. For JSON it will return something similar to:

```
{
    "success": false,
    "errors": [
        "Server Error"
    ]
}
```

1.10 Utility Classes

Cohesion comes with several extremely lightweight utility classes such as input handling, database interaction, configuration, and many more. More will be added over time and I'm always happy to receive pull requests with additional utilities.

1.10.1 Database Interaction

A MySQL database library is included to provide safe interaction with MySQL databases. The database class includes support for master/slave queries, transactions, named variables, as well as many other features.

The Database library isn't an ORM, it requires you to write SQL statements. This allows for better performance optimization and forces you to think about what SQL will actually be executed.

The database library uses Mustache style named parameters.

For more information on the database library read the documentation at the start of the `MySQL.php` file.

1.10.2 Configuration Library

A config object class is provided for easy and extendible configuration. The config object will read one or more JSON formatted files and sections of the config can be given to classes to provide either business rule constants or library configurations. It is designed so that you can have a default config file with all the default configurations for your application then you can have a production config file that will overwrite only the variables within that config file such as database access etc.

You can access sub settings using dot notation.

1.10.3 Input Handler

An extremely simple input handler is provided. The input handler doesn't do anything to prevent SQL injection or XSS as these are handled by the Database library and the Views respectively.

1.11 Examples

I will be adding some examples of using Cohesion for some simple applications soon.

1.12 Contributing

I would love to hear your feedback on Cohesion and welcome any pull requests. Feel free to raise any issues or merge requests on any of the GitHub repositories

1.12.1 Licence

Cohesion is open source and released under the MIT licence.

1.12.2 Code

Feel free to checkout the code directly from GitHub.

The Cohesion Framework repository The `cohesion-framework` is the basic starting point for creating new projects and includes the `cohesion-core` package.

The Cohesion Core Package repository The `cohesion-core` contains the main classes used in the Cohesion framework.

Cohesion Docs repository The repository containing this documentation.

CHAPTER 2

Code

Cohesion on GitHub