aint framework Documentation

Release 1.0prototype

Alexander Steshenko

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Introduction

Overview

aint framework is a functional and procedural programming framework for modern PHP (at least PHP version 5.4 is required).

Object Oriented Programming has conquered the world of PHP and now is the mainstream programming paradigm for creating PHP web applications. **aint framework** offers an alternative way of approaching, with the two key concepts being: *data* and *functions* to process it. It does not apply any additional restrictions.

The framework consists of namespaces or packages. For example, `aint\mvc` namespace includes functions for routing and dispatching an http request.

There is practically no learning curve to **aint framework**. Just like with the good old plain PHP: you have functions, you call them with parameters and you write your own functions. However, novice developers may find it easier to write potentially bad code, because of the lack of restrictions (unlike in OOP).

A few facts about **aint framework**:

1. PHPUnit library is used for testing
2. Unique *coding standard*
3. Data is presented with PHP's built-in types such as integer, string, array
4. PHP classes are only used to *present errors* (via exceptions mechanism)
5. There are no static dependencies on data in the framework's code

Installation

1. If you use composer for your project, add this to your `composer.json`:

```
"require": {
     "aintframework/aint_framework": "dev-master"
 }
```

and install the dependency.

2. Alternatively, download the framework from the GitHub page.
3. Recommended way of starting a project is to use *Skeleton Application*.

Usage

At the moment, packages (namespaces) autoloading is not achievable with PHP, so all namespaces have to be included into the project explicitly.

It's recommended to add **aint framework** to the `include_path`:

```
<?php
set_include_path(get_include_path()
    . PATH_SEPARATOR . realpath(dirname(__FILE__) . '/../vendor/aintframework/aint_
↪framework/library'));
```

with that you can use framework's features as follows:

```
<?php
require_once 'aint/mvc/dispatching.php';
use aint\mvc\dispatching;

const actions_namespace = 'my_actions';
const error_handler = 'my_error_handler';

dispatching\run_default(actions_namespace, error_handler);
```

Read more about using the framework for building a simple web application in the *QuickStart* section.

CHAPTER 2

QuickStart

Contents:

Application Overview

The application is an online albums manager, allowing you to create, edit, delete and view music albums in your collection. As any other CRUD task, this one will need four pages:

- list albums in the collection
- add new album to the collection
- edit (or update) an existing album
- delete an existing album

To store information we'll use a relational database. Could be anything, but for demonstration purposes we'll go with a SQLite3 database. One table is needed to store the albums records with these being an album characteristics:

- id (integer, auto-increment) - a unique album identifier
- artist (string)
- title (string)

Skeleton Application

To start development with **aint framework** we'll create a sample project from the skeleton application git repository.

You can either download and unzip it or, better, clone the repository:

```
cd my/projects/dir
git clone git://github.com/aintframework/skeleton_application.git albums_manager
cd albums_manager
```

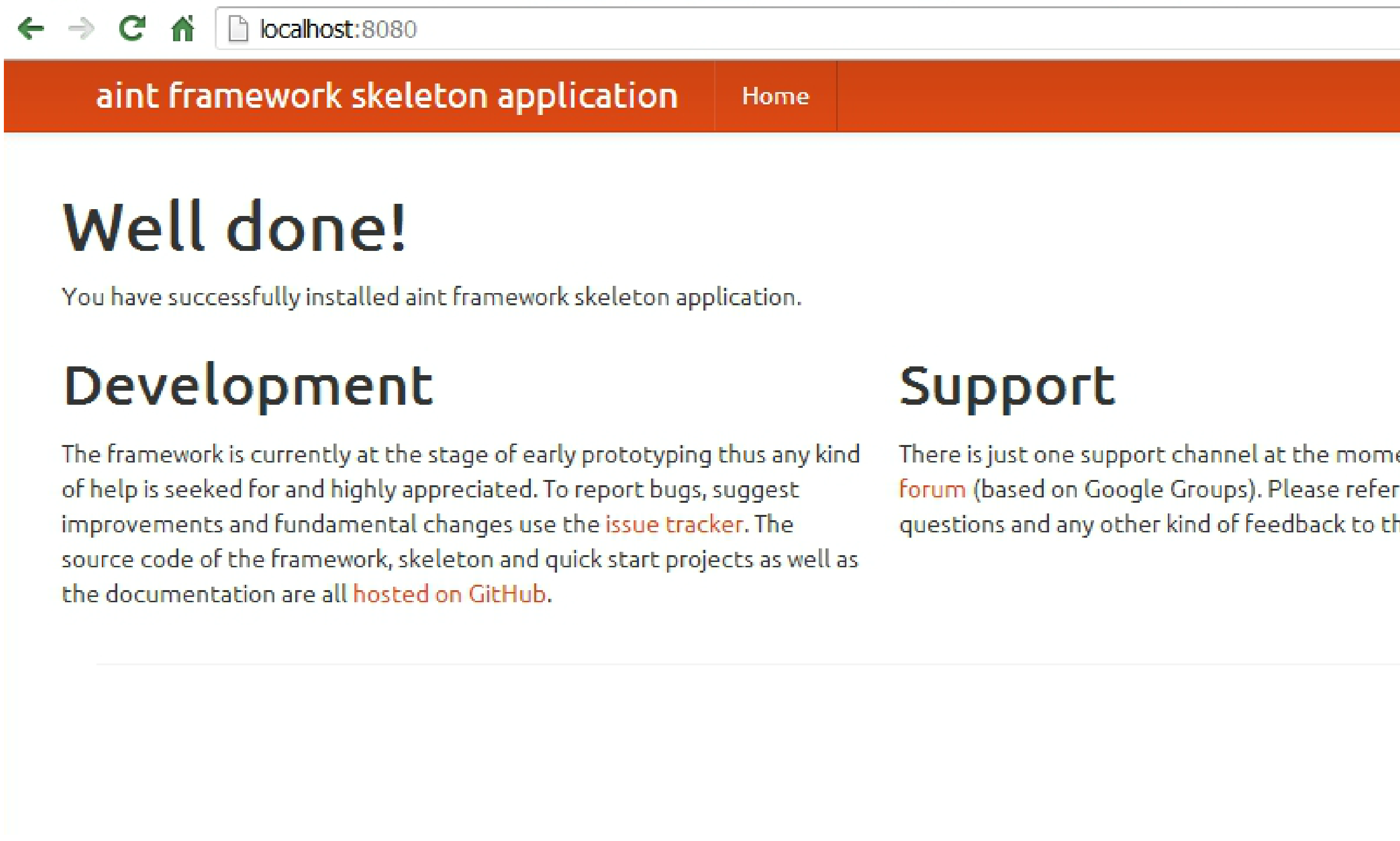
Install **aint framework** using composer:

```
php composer.phar self-update
php composer.phar install
```

Next, run the built-in web server:

```
cd www
php -S localhost:8080 dev-router.php
```

Your project is now running. Access http://localhost:8080 in your browser:



Well done!

You have successfully installed aint framework skeleton application.

Development

The framework is currently at the stage of early prototyping thus any kind of help is seeked for and highly appreciated. To report bugs, suggest improvements and fundamental changes use the issue tracker. The source code of the framework, skeleton and quick start projects as well as the documentation are all hosted on GitHub.

Support

There is just one support channel at the mome
forum (based on Google Groups). Please refer
questions and any other kind of feedback to th

Controllers and Routing

Skeleton application uses `aint\mvc\routing\route_segment` function for routing. We'll use it for our four new pages as well. We'll create a package to hold the action functions: `/src/app/controller/actions/albums.php` and add there the ones that we need:

```
<?php
namespace app\controller\actions\albums;

function list_action($request, $params) {

}

function add_action($request, $params) {
}

function edit_action($request, $params) {

}

function delete_action($request, $params) {

}
```

Notice, here and further the name of the package file always corresponds to the namespace. It is by convention we use for this demo and for the framework itself. (Nothing is enforced in **aint framework**).

`route_segment` will route `/albums/add` uri to `app\controller\actions\albums\add_action` function. `$request` holds data about the current HTTP request, while `$params` contains parameters of the route. For instance, `/albums/edit/id/123` will be routed to `app\controller\actions\albums\edit_action` with `$params` = `['id' => 123]`.

We'll need to "enable" the new actions namespace by adding it to `app\controller`:

```
<?php
// actions
require_once 'app/controller/actions/index.php';
require_once 'app/controller/actions/errors.php';
require_once 'app/controller/actions/albums.php'; // - adding this
```

To make the application list albums on the index page instead of the default "Welcome" page, we'll change the `app\controller\actions\index\index_action` like this:

```
<?php
function index_action() {
    // does nothing, simply delegates
    return \app\controller\actions\albums\list_action();
}
```

To have all this working now, we'll also need to create some simple templates and have them rendered. Create these four files and leave them empty for now:

- `/src/app/view/templates/albums/add.phtml`
- `/src/app/view/templates/albums/edit.phtml`
- `/src/app/view/templates/albums/delete.phtml`
- `/src/app/view/templates/albums/list.phtml`

And make the change in the controllers:

```
<?php
namespace app\controller\actions\albums;

// including the app's view package to be able to render response:
require_once 'app/view.php';
use app\view;

function list_action() {
    return view\render('albums/list');
}
```

```
function add_action($request) {
    return view\render('albums/add');
}

function edit_action($request, $params) {
    return view\render('albums/edit');
}

function delete_action($request, $params) {
    return view\render('albums/delete');
}
```

Make sure the following links work: Index/List | Add | Edit | Delete

By the way, you can make sure the default error handler also works: http://localhost:8080/badurl

Let's now drop in some logic, i.e. *The Model*

Note: Find out more about recommended application structure *in this tutorial*

Model

The Service Layer of our model will be the `app\model\albums` namespace. As it's a very simple, typical CRUD application, the facade of the model will look almost the same as the public interface (the controller):

```
<?php
namespace app\model\albums;

/**
 * Parameters of an album
 */
const param_id = 'id',
      param_title = 'title',
      param_artist = 'artist';

function list_albums() {
    // ...
}

function add_album($data) {
    // ...
}

function get_album($id) {
    // ...
}

function delete_album($id) {
    // ...
}

function edit_album($id, $data) {
    // ...
}
```

What's different is the additional `get_album` function to view one particular album. It's needed to show the information in the html form, when modifying an existing album.

As decided we'll store albums records in a SQLite database:

```
mkdir database
sqlite3 database/data
```

of one table:

```
create table albums (
    id integer primary key,
    title varchar(250),
    artist varchar(250)
);
```

The database is created in `/albums_manager/database/data` file. We're placing it in a separate directory as it's not really the source code. To connect to the database and share this connection statically with the whole app, we'll need `app\model\db` namespace:

```
<?php
namespace app\model\db;

require_once 'app/model.php';
use app\model;
require_once 'aint/common.php';
use aint\common;
// including sqlite platform and driver packages
require_once 'aint/db/platform/sqlite.php';
require_once 'aint/db/driver/pdo.php';

const driver = 'aint\db\driver\pdo';
const platform = 'aint\db\platform\sqlite';

function db_connect() {
    static $resource;
    if ($resource === null) { // we'll only connect to the db once
        $db_connect = driver . '\db_connect';
        $resource = $db_connect(model\get_app_config()['db']);
    }
    return $resource;
}
```

This function uses model configuration that we add to `src/app/model/configs/app.inc`:

```
<?php
return [
    'db' => [
        'dns' => 'sqlite:/my/projects/dir/database/data'
    ]
];
```

Note: You can override this and any other setting locally, by creating `app.local.inc` file in the same directory.

Model for this app is designed to use the Table Data Gateway pattern, with `app\model\db\albums_table` being this gateway. Let's create it as well, adding functions required to read, write, update and delete data from the `albums` table. We'll need them all:

```php
<?php
namespace app\model\db\albums_table;

require_once 'app/model/db.php';
use app\model\db;
require_once 'aint/db/table.php';

const table = 'albums';

/**
 * Partial application,
 * function delegating calls to aint\db\table package
 * adding platform and driver parameters
 *
 * @return mixed
 */
function call_table_func() {
    $args = func_get_args();
    $func = 'aint\db\table\\' . array_shift($args);
    $args = array_merge([db\db_connect(), db\platform, db\driver, table], $args);
    return call_user_func_array($func, $args);
}

function select(array $where = []) {
    return call_table_func('select', $where);
}

function insert(array $data) {
    return call_table_func('insert', $data);
}

function update($data, $where = []) {
    return call_table_func('update', $data, $where);
}

function delete(array $where = []) {
    return call_table_func('delete', $where);
}
```

Notice, while framework is being used for the actual work, to wire it into your app you have to write all the functions you need inside the app's namespace. This idea is used for extending anything in **aint framework** and has functional programming paradigm behind it.

Instead of configuring instances, changing the *state* to suit your needs, like you would do in other popular frameworks you go right to extension the base code with your own.

Note: Read more *here*

Every function, essentially, is a partial application, a proxy to the table gateway implementation provided by the framework. We specify namespaces for `platform` and `driver` to use.

Note: Read more about managing shared and not shared dependencies *in this tutorial*

Let's return to the Service Layer, `app\model\albums` now and fill in missing details:

```
<?php
namespace app\model\albums;

// app uses table gateway pattern:
require_once 'app/model/db/albums_table.php';
use app\model\db\albums_table;

/**
 * Parameters of an album
 */
const param_id = 'id',
      param_title = 'title',
      param_artist = 'artist';

function list_albums() {
    // simply return all records from the table
    return albums_table\select();
}

function add_album($data) {
    // insert data into the table
    albums_table\insert($data);
}

function get_album($id) {
    // look up all records in the table with id provided and return the first one
    return current(albums_table\select(['id' => $id]));
}

function delete_album($id) {
    // removes records from db with id provided
    albums_table\delete(['id' => $id]);
}

function edit_album($id, $data) {
    // updates records in db fulfilling the id = ? constraint with the data array␣
↪provided
    albums_table\update($data, ['id' => $id]);
}
```

Wiring Model and Controller together

Let's return to the controller we prepared in the previous section:

```
<?php

namespace app\controller\actions\albums;

require_once 'app/model/albums.php';
use app\model\albums as albums_model;
require_once 'app/view.php';
use app\view;
require_once 'aint/http.php';
use aint\http;

function list_action() {
    return view\render('albums/list',
        // passing the list of albums to the template
        ['albums' => albums_model\list_albums()]);
}

function add_action($request) {
    if (!http\is_post($request)) // if this isn't a POST request
        return view\render('albums/add'); // we simply show the HTML form
    else {
        // if it is a POST request, we add the new
        albums_model\add_album($request['params']);
        // and redirect to the index page
        return http\build_redirect('/');
    }
}

function edit_action($request, $params) {
    if (!http\is_post($request)) // if this isn't a POST request
        return view\render('albums/edit',  // we show the HTML form
            // filling current album data in the form
            ['album' => albums_model\get_album($params['id'])]);
    else {
        // if it is a POST request, we update the data in the model
        albums_model\edit_album($params['id'], $request['params']);
        // and redirect to the index page
        return http\build_redirect('/');
    }
}

function delete_action($request, $params) {
    // ask the model to delete the album
    albums_model\delete_album($params['id']);
    // and redirect to the index page
    return http\build_redirect('/');
}
```

The only missing part now is *the View*

View (presentation)

First of all, to take it off the table: the delete controller doesn't really need a template to be rendered. After an album is successfully removed we simply redirect the browser to the albums index:

```
<?php
function delete_action($request, $params) {
    // ask the model to delete the album
    albums_model\delete_album($params['id']);
    // and redirect to the index page
    return http\build_redirect('/');
}
```

So we can delete `templates/albums/delete.phtml` now. The other templates we do need. Let's start with `list.phtml`:

```
<?php
require_once 'app/view/helpers.php';
```

```
use app\view\helpers;

helpers\head_title($title = helpers\translate('My Albums'));
?>
<h1><?= htmlspecialchars($title) ?></h1>
<p>
    <a href="<?= helpers\uri('albums\add_action') ?>">Add new album</a>
</p>
<table class="table table-bordered table-striped">
    <thead>
    <tr>
        <th>Title</th>
        <th>Artist</th>
        <th>Controls</th>
    </tr>
    </thead>
    <tbody>
    <? foreach ($albums as $album) : ?>
    <tr>
        <td><?= htmlspecialchars($album['title']) ?></td>
        <td><?= htmlspecialchars($album['artist']) ?></td>
        <td>
            <a href="/albums/edit/id/<?= $album['id'] ?>">Edit</a> |
            <a href="/albums/delete/id/<?= $album['id'] ?>">Delete</a>
        </td>
    </tr>
        <? endforeach ?>
    </tbody>
</table>
```

It outputs the albums in an HTML table, populating it using the extracted `$albums` variable. A few interesting things to notice are:

1. Usage of `helpers\uri` - the function to convert a target action-function back to an uri.
2. Usage of `helpers\translate` - a simple translation function to help with localization of your app.
3. `htmlspecialchars` is the PHP's function used to escape strings.

A bit more interesting are `edit.phtml`:

```
<?php
require_once 'app/view/helpers.php';
use app\view\helpers;

helpers\head_title($title = helpers\translate('Edit album'));
?>
<h1><?= htmlspecialchars($title) ?></h1>
<p><?= helpers\album_form('/albums/edit/id/' . $album['id'], $album) ?></p>
```

and `add.phtml`:

```
<?php
require_once 'app/view/helpers.php';
use app\view\helpers;

helpers\head_title($title = helpers\translate('Add new Album'));
?>
<h1><?= htmlspecialchars($title) ?></h1>
```

```
<p><?= helpers\album_form('/albums/add') ?></p>
```

As the HTML form in both cases is almost the same and duplication of code is never good we move the common html to another file, `/src/app/view/templates/album_form.phtml`:

```
<form action="<?= $action ?>" method="post" class="form-horizontal well">
    <fieldset>
        <div class="control-group">
            <label class="control-label" for="title">Title</label>

            <div class="controls">
                <input type="text" class="input-xlarge" id="title" name="title"
                       value="<?= htmlspecialchars($album['title']) ?>"/>
            </div>
        </div>
        <div class="control-group">
            <label class="control-label" for="artist">Artist</label>

            <div class="controls">
                <input type="text" class="input-xlarge" id="artist" name="artist"
                       value="<?= htmlspecialchars($album['artist']) ?>"/>
            </div>
        </div>
        <div class="form-actions">
            <button type="submit" class="btn btn-primary">Save</button>
        </div>
    </fieldset>
</form>
```

to simplify things further, we introduce the `app\view\helpers\album_form` function:

```
<?php
function album_form($action, $album = []) {
    $default_album_data = [
        'title' => '',
        'artist' => '',
    ];
    $album = array_merge($default_album_data, $album);
    return view\render_template('album_form', ['album' => $album, 'action' =>
↪$action]);
}
```

This is the beauty of the simplicity **aint framework** gives you. No more plugins, partials, helpers, dependency headaches. You are free to do the simplest thing possible.

Conclusion

This is it! Make sure the app is working: http://localhost:8080

Covering your application with tests

Whether you prefer TDD or cover the code with test after you have written it: both are perfectly possible with **aint framework**.

While adding tests to your application is not covered in this particular Quick Start, feel free to check some of these additional resources on testing:

PHPUnit Manual

aint framework tests

aint framework testing guidelines

On testing static dependencies

This guide will showcase an implementation of a typical CRUD web application, using **aint framework**. This demo is hugely inspired by the Zend Framework 2 Getting Started Example and solves the same task in a similar way. If you like ZF2 and has done that tutorial, you will be able to compare the approaches and the amount of coding needed.

To proceed with the demo, all you need is PHP 5.4 installed in your system and *php_pdo_sqlite* extension enabled.

Proceed to application description

CHAPTER 3

User Guides

There may be many gotchas while starting with **aint framework**, especially if all you have dealt with in the past were popular Object Oriented PHP frameworks. Tutorials presented in this section demonstrate peculiarities of the framework as well as highlight interesting use cases of specific components.

Application Structure

Being a framework is not only providing tools to use, but recommendations on how it's best to use them, as well.

This is the top directory of the *skeleton project*:

```
/albums_manager
    /src
    /vendor
    /www
```

`vendor` - is the directory where all external application dependencies will reside, including the **aint framework** itself.

`src` - is the container for the actual code. All the backend programming happens inside.

`www` is the document root for the web server, it contains various public resources such as images and styles. It also serves the `index.php`, the entry point of your application. It looks like this:

```
<?php
set_include_path(get_include_path()
    . PATH_SEPARATOR . realpath(dirname(__FILE__) . '/../src')
    . PATH_SEPARATOR . realpath(dirname(__FILE__) . '/../vendor/aintframework/aint_
↪framework/library'));

require_once 'app/controller.php';
app\controller\run();
```

It does three things:

1. Adds both application source and **aint framework** to `include_path` so their components can be included easily when needed.
2. Includes `app/controller.php` file, the package containing the Front Controller function (the actual entry point of any HTTP request).
3. Runs the app (through invoking the function playing the role of front controller).

MVC

When it comes to writing and organizing the actual code, the suggested pattern to follow for a web application is Model-View-Controller:

```
/src
    /app
        /controller
        /model
        /view
        controller.php
        model.php
        view.php
```

An important thing about this structure: it's very specific and thorough about implementing the Model-View-Controller pattern. For each piece of code, whether it's a config file, a function, a constant or a template: you will have to decide what it is: Model, Controller or View, the code for which is located in namespaces `app\model`, `app\controller`, `app\view` accordingly.

`controller.php`, `model.php`, `view.php` are simply files/namespaces to put some "upper-level", general code for which you don't feel like creating a subpackage.

Dealing with Dependencies

PHP frameworks have evolved along with PHP itself from using global state, registry pattern, singleton to Dependency Injection, Service Locator and Inversion of Control principles.

aint framework has learned from this path, from practical use cases and tasks being solved using PHP. The result is a solution as obvious and simple as it gets.

Imagine you have a resource (an object in OOP, simply data in **aint framework**): *a db connection*. You want this resource to be shared within the application. Essentially what you want is this resource to be **static**. PHP has a keyword for this purpose and **aint framework** encourages you to use it, to keep things simple:

```php
<?php
function get_db() {
    static $db;
    if ($db === null)
        $db = aint\db\db_connect(/* ... */);
    return $db;
}
```

When you first ask for this resource/data it'll be created/fetched/composed/calculated and then the same one will be returned to all consecutive calls.

If you need the dependency management to follow some other logic, e.g. new resource each time, - you can code it in accordingly:

```php
<?php
function get_db() {
    // new connection each time
    $db = aint\db\db_connect(/* ... */);
    return $db;
}
```

Here you're not limited with the features provided by a particular DI container implementation. You manage dependencies yourself.

Note: The only tricky bit is testing static dependencies. On testing static dependencies

Error Handling

To handle exceptional situations (when an operation/function cannot be completed for some reason) **aint framework** uses the PHP's mechanism of exceptions.

There may be many exceptions defined per package:

```php
<?php
namespace app\model\posts;

/**
 * Error thrown when the title of a post exceeds allowed length
 */
class title_too_long_error extends \DomainException{};

/**
 * Error thrown when connection to db failed
 */
class db_connection_error extends \Exception{};
```

Usage is as per official documentation:

```php
<?php
namespace app\controller\actions\posts;

use aint\http;
use app\model\posts as posts_model;

function add_action($request, $params) {
    try {
        posts_model\new_post($request);
    } catch (posts_model\title_too_long_error $error) {
        // if title was too long when adding a post, we redirect to main page
        return http\build_redirect('/');
    }
}
```

Note: Exceptions have to extend PHP's built-in `Exception` class and are all objects. This solution is not ideal but was chosen for now because there is no decent alternatives for exceptions handling in PHP.

Conceptually, using what functions return would fit **aint framework** ideology perfectly (e.g. like in Go programming language), however PHP doesn't allow for that conveniently.

The area of exceptions handling in **aint framework** is still being discussed. Suggestions, feedback and any other kind of input are highly appreciated.

Extension over Configuration

todo partial application, currying, what's different to OOP frameworks

CHAPTER 4

Namespaces & Functions reference

todo

CHAPTER 5

Development

Sections:

Coding Standard

aint framework coding standard used for the framework itself and also recommended for **aint framework** driven applications.

File Formatting

General

- closing tag `?>` is never permitted for PHP files
- usage of short tags e.g. `<?=` and `<?` is allowed for templates
- one source file should contain one namespace/package
- only Unix-style line termination is allowed (LF or n)
- 80 - 120 characters is the recommended line length
- indentation is 4 spaces, tabs are not allowed.

File extensions

By convention, some of the recommended file name extensions are:

- `.php` for a full PHP source file, a package/namespace
- `.phtml` for PHP-powered templates
- `.inc` for configuration, localization files and alike

Naming Conventions

Namespaces (Packages)

Lowercase alphanumeric characters are permitted, with underscores used as separators:

- `namespace aint\mvc\routing`
- `namespace app\model\my_db`

Filenames

Files are named after the namespace (package) they contain:

- `namespace aint\mvc\routing` => `aint/mvc/routing.php`
- `namespace app\model\my_db` => `app/model/my_db.php`

Functions

Lowercase alphanumeric characters are permitted, with underscores used as separators:

- `function build_response($body = '', $code = 200, $headers = [])`
- `function render_template()`

Variables

Lowercase alphanumeric characters are permitted, with underscores used as separators:

- `$quote_identifier`
- `$router`

Constants

Lowercase alphanumeric characters are permitted, with underscores used as separators:

```
/**
 * Http Request method types
 */
const request_method_post = 'POST',
      request_method_get = 'GET',
      request_method_put = 'PUT',
      request_method_delete = 'DELETE';
```

Exceptions

Lowercase alphanumeric characters are permitted, with underscores used as separators, e.g.:

```
/**
 * Error thrown when an http request cannot be routed
 */
class not_found_error extends \exception {};
```

Coding Style

Strings

todo

Arrays

todo

Functions

todo: declaration and usage

Control Statements

For all control structures (if/switch/for/while) the opening brace is written on the same line as the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented.

The braces are only used when there is more than one line in the content:

```
if ($data[some_boolean_param])
    return 'yes';
else
    return 'no';
```

Code Documentation

todo

Testing Practices

todo

PHPUnit Manual

aint framework tests

On testing static dependencies