LightMVC Documentation

Release 3.1.2

Foreach Code Factory

May 20, 2020

Contents

And, then, there was truly light!

LightMVC Framework Home Page

Easily create PHP applications by using any PHP library within this very modular, event-driven and Swoole-enabled framework!

CHAPTER 1

What's new in version 3.1.2 (2019-05-13)

- Removes some dead code (Skeleton Application).
- Updates the project's dependencies.
- Updates the user documentation.

CHAPTER 2

What's new in version 3.1.1 (2019-05-08)

- Updates all the templates (Skeleton Application).
- Updates the documentation.

CHAPTER 3

What's new in version 3.1.0 (2019-04-30)

- Adds aggregate root controllers for easier usage of aggregate-based functionality.
- Adds asynchronous functionality to the Event Dispatcher.
- Adds asynchronous commands in non-Swoole environments using ReactPHP/Symfony Process.
- Adds a Command Runner to make running commands independent of the PHP environment (Swoole or non-Swoole).
- Updates the documentation.

CHAPTER 4

What's new in version 3.0.0 (2019-04-16)

- Adds controller-based Event Sourcing aggregates to the framework with a PSR-14 compliant Event Dispatcher (event bus).
- Adds the facilities to use CQRS.
- Updates the documentation.

CHAPTER 5

What's new in version 2.1.1 (2019-04-16)

- Fixes a few unit tests and a few minor issues concerning code comments.

CHAPTER 6

What's new in version 2.1.0 (2019-04-10)

- Adds asynchronous non-blocking PHP sessions (Swoole compatible).
- Adds a PSR-6 compliant interface and a corresponding proxy class to Doctrine\Common\Cache classes for session caching.
- Updates the documentation.

CHAPTER 7

What's new in version 2.0.4 (2019-04-16)

- Fixes a few unit tests and a few minor issues concerning code comments.

CHAPTER 8

What's new in version 2.0.3 (2019-04-10)

- Fixes an issue when requesting an unknown controller method.

CHAPTER 9

What's new in version 2.0.2 (2019-01-23)

- Fixes an issue when running in Swoole mode behind an NGINX HTTPS proxy server.
- Fixes an issue with the way the controller namespace was obtained from the file path.
- Updates the documentation.

CHAPTER 10

What's new in version 2.0.1 (2019-01-20)

- Fixes an issue with the Bootstrap event on Windows.
- Modifies the Controller Manager in order to receive an instance from a Controller Factory directly.
- Updates the documentation.
- Updates the API documentation.

CHAPTER 11

What's new in version 2.0.0 (2019-01-15)

- Adds support for running the framework on Swoole.
- Uses PSR-7 compliant HTTP messages (Zend Diactoros).
- Uses PSR-15 compliant middleware and pipelines (Zend Stratigility).
- Uses the nikic/fast-route routing library.
- Uses the Pimple Container as a service manager.
- Uses an event-driven architecture (Zend EventManager).
- Uses Plates as the default template manager.
- Adds Twig as a possible template manager.
- Updates the Smarty template manager.
- Adds TailwindCSS to the default templates.
- Updates Bootstrap CSS Framework in alternative templates.
- Updates the Doctrine Framework (domain logic).
- Adds the Atlas ORM Framework (persistence logic).

CHAPTER 12

LightMVC License

12.1 Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   You must give any other recipients of the Work or Derivative Works a copy of this License; and You must cause any modified files to carry prominent notices stating that You changed the files; and You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide

additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

CHAPTER 13

Installation

13.1 Prerequisites

- PHP 7.2

13.2 Installation using Composer

To add the **LightMVC Framework** package to your PHP project, you can simply run the following command on your computer's CLI:

```
$ composer require lightmvc/ascmvc
```

You can also use the **LightMVC Skeleton Application** by issuing these commands:

```
$ git clone https://github.com/lightmvc/lightmvcskel
$ cd lightmvcskel
$ composer install
```

Note: The LightMVC Skeleton Application can also be downloaded as an archive file from the LightMVC Download Page.

CHAPTER 14

Configuration

The framework's configuration is set within a configuration file named `config/config.php` inside the project's root directory. It is possible to override the configurations found within this file by creating a file named `config.local.php` within the same `config` directory.

All configuration options must be stored within an index of the `$baseConfig` array.

The main preconfigured indexes of this array are:

- `BASEDIR`, which contains the full path to the project's root directory,
- `URLBASEADDR`, which contains the Web URL of the project,
- `appFolder`, which contains the name of the project's root directory,
- `env`, which contains an environment setting ('production' or 'development'),
- `routes`, which contains an array of FastRouter routes to be used,
- `templateManager`, which contains the name of the Template Manager that is to be used ('Plates', 'Twig' or 'Smarty'),
- `templateDir` under the `templates` index, which contains the name of the folder where the templates are stored,
- `async_process_bin`, which contains the path to the PHP script that will be used to fork processes in order to run event sourcing commands,
- `async_commands`, which contains an array of fully-qualified class names to run asynchronously,
- `events`, which contains an array of parameters in order to configure the LightMVC event sourcing controller-based aggregates,
- `session`, which contains an array of parameters in order to configure the LightMVC asynchronous PHP session.

Note: The Twig and Smarty template managers require additional indexes under the `templates` index. These are: `compileDir`, `configDir` and `cacheDir`.

Also, there are four optional preconfigured indexes in the `$baseConfig` array:

- `middleware`, which contains an array of PSR-15 compliant middleware to be used,
- `eventlog`, which contains an array of parameters in order to configure the LightMVC event sourcing Logger,
- `doctrine`, which contains an array of parameters in order to configure one or more Doctrine connections.
- `atlas`, which contains an array of parameters in order to configure one or more Atlas connections.

Here is an example of a `config/config.php` file:

```
<?php

$baseConfig['env'] = 'production'; // 'development' or 'production'

$baseConfig['appName'] = 'The LightMVC Framework Skeleton Application';

// Required configuration
require 'events.config.php';

require 'commands.config.php';

require 'routes.config.php';

require 'view.config.php';

// Optional configuration
include 'middleware.config.php';

include 'session.config.php';
```

14.1 Event Sourcing Configuration

When configuring event sourcing aggregates, one must configure both the event bus and, optionally, the command bus if one is to use asynchronous commands.

For example, the `config/events.config.php` file might look like the following:

```
<?php

$baseConfig['events'] = [
    // PSR-14 compliant Event Bus.
    'psr14_event_dispatcher' => \Ascmvc\EventSourcing\EventDispatcher::class,
    // Different read and write connections allow for simplified (!) CQRS. :)
    'read_conn_name' => 'dem1',
    'write_conn_name' => 'dem1',
];

$baseConfig['eventlog'] = [
    'enabled' => true,
    'doctrine' => [
        'log_conn_name' => 'dem1',
        'entity_name' => \Application\Log\Entity\EventLog::class,
    ],
    // Leave empty to log everything, including the kitchen sink. :)
    // If you you start whitelisting events, it will blacklist everything else by␣
↪default.
```

(continues on next page)

(continued from previous page)

```
    'log_event_type' => [
        'whitelist' => [
            \Ascmvc\EventSourcing\Event\WriteAggregateCompletedEvent::class,
        ],
        'blacklist' => [
            //\Ascmvc\EventSourcing\Event\AggregateEvent::class,
        ],
    ],
];
```

Note: For more information on configuring the application's event sourcing aggregates and the application's event log, please see the *Event Sourcing* section.

And, if using asynchronous commands, the `config/commands.config.php` file might look like the following:

```
<?php

// The PHP script to use when forking processes.
$baseConfig['async_process_bin'] = $baseConfig['BASEDIR']
. DIRECTORY_SEPARATOR
. 'bin'
. DIRECTORY_SEPARATOR
. 'process.php';

// List of commands to run asynchronously.
$baseConfig['async_commands'] = [
    \Application\Commands\ReadProductsCommand::class,
    \Application\Commands\WriteProductsCommand::class,
];
```

Note: When running the application as a Swoole coroutine, listed commands will not be forked, since new processes cannot be created from within a Swoole coroutine.

14.2 Routing Configuration

The `config/routes.config.php` file might look like the following:

```
$baseConfig['routes'] = [
    0 => [
        'GET',
        '/',
        'index',
    ],
];
```

Note: For more information on configuring the application's routes, please see the *Routing* section.

14.3 Session Configuration

And, the `config/session.config.php` file might look like this:

```
$baseConfig['session'] = [
    'enabled' => true,
    'psr6_cache_pool' => \Ascmvc\Session\Cache\DoctrineCacheItemPool::class,
    'doctrine_cache_driver' => \Doctrine\Common\Cache\FilesystemCache::class,
    //'doctrine_cache_driver' => \Doctrine\Common\Cache\XcacheCache::class,
    //'doctrine_cache_driver' => \Doctrine\Common\Cache\RedisCache::class,
    //'doctrine_cache_driver' => \Doctrine\Common\Cache\MemcachedCache::class,
    //'doctrine_cache_driver' => \Doctrine\Common\Cache\MemcacheCache::class,
    'doctrine_filesystem_cache_directory' => BASEDIR . DIRECTORY_SEPARATOR . 'cache' .
↪ DIRECTORY_SEPARATOR,
    'doctrine_cache_server_params' => [
        'host' => '127.0.0.1',
        'port' => 6379, // redis
        //'port' => 11211 // memcached/memcache
    ],
    'session_name' => 'PHPSESSION',
    'session_path' => '/',
    'session_id_length' => 32,
    'session_id_type' => 1,
    'session_storage_prefix' => 'ascmvc',
    'session_expire' => 60 * 30, // 30 minutes
];
```

It is possible to replace the value of the `psr6_cache_pool` index with any PSR-6 compliant class. Also, any of the given Doctrine\Common\Cache classes can be used in order to store the session data when using the `\Ascmvc\Session\Cache\DoctrineCacheItemPool` class.

For more information on sessions, please see the *Sessions* section.

14.4 View Configuration

Moreover, it might be useful to configure common view elements in the local configuration files in order to keep things simple and use these elements within the application's templates, as these configuration elements will be available to any of the application's controllers.

Here is an example of setting up common view elements within a `config/view.config.php` file:

```
$baseConfig['templateManager'] = 'Plates';

$baseConfig['templates'] = [
    'templateDir' => $baseConfig['BASEDIR'] . DIRECTORY_SEPARATOR . 'templates' .␣
↪DIRECTORY_SEPARATOR . 'plates' . DIRECTORY_SEPARATOR,
    'compileDir' => $baseConfig['BASEDIR'] . DIRECTORY_SEPARATOR . 'templates_c' .␣
↪DIRECTORY_SEPARATOR,
    'configDir' => $baseConfig['BASEDIR'] . DIRECTORY_SEPARATOR . 'config' .␣
↪DIRECTORY_SEPARATOR,
    'cacheDir' => $baseConfig['BASEDIR'] . DIRECTORY_SEPARATOR . 'cache' . DIRECTORY_
↪SEPARATOR,
];

$baseConfig['view'] = [
```

(continues on next page)

(continued from previous page)

```
    'urlbaseaddr' => $baseConfig['URLBASEADDR'],
    'logo' => $baseConfig['URLBASEADDR'] . 'img/logo.png',
    'favicon' => $baseConfig['URLBASEADDR'] . 'favicon.ico',
    'appname' => $baseConfig['appName'],
    'title' => "Skeleton Application",
    'author' => 'Andrew Caya',
    'description' => 'Small CRUD application',
    'css' =>
        [
            $baseConfig['URLBASEADDR'] . 'css/bootstrap.min.css',
            $baseConfig['URLBASEADDR'] . 'css/dashboard.css',
            $baseConfig['URLBASEADDR'] . 'css/bootstrap.custom.css',
            $baseConfig['URLBASEADDR'] . 'css/dashboard.css',

        ],
    'js' =>
        [
            $baseConfig['URLBASEADDR'] . 'js/jquery-3.3.1.min.js',
            $baseConfig['URLBASEADDR'] . 'js/bootstrap.min.js',

        ],
    'bodyjs' => 0,
    'links' =>
        [
            'Home' => $baseConfig['URLBASEADDR'] . 'index',

        ],
    'navmenu' =>
        [
            'Home' => $baseConfig['URLBASEADDR'] . 'index',

        ],

];
```

For more information on configuring the application's view, please see the *Views* section.

14.5 Middleware Configuration

The `config/middleware.config.php` file might look like the following:

```
$baseConfig['middleware'] = [
    '/foo' => function ($req, $handler) {
        $response = new \Zend\Diactoros\Response();
        $response->getBody()->write('FOO!');

        return $response;
    },
    function ($req, $handler) {
        if (! in_array($req->getUri()->getPath(), ['/bar'], true)) {
            return $handler->handle($req);
        }

        $response = new \Zend\Diactoros\Response();
```

(continues on next page)

(continued from previous page)

```
        $response->getBody()->write('Hello world!');

        return $response;
    },
    '/baz' => [
        \Application\Middleware\SessionMiddleware::class,
        \Application\Middleware\ExampleMiddleware::class,
    ],
];
```

Note: The *Middleware* section contains all the needed information in order to set up PSR-15 compliant middleware.

14.6 Model Configuration

Finally, you can configure Doctrine within a `config/config.local.php` file, as follows:

```
$baseConfig['doctrine']['DBAL']['dcm1'] = [
    'driver'   => 'pdo_mysql',
    'host'     => 'localhost',
    'user'     => 'USERNAME',
    'password' => 'PASSWORD',
    'dbname'   => 'DATABASE',
];

// AND/OR

$baseConfig['doctrine']['ORM']['dem1'] = [
    'driver'   => 'pdo_mysql',
    'host'     => 'localhost',
    'user'     => 'USERNAME',
    'password' => 'PASSWORD',
    'dbname'   => 'DATABASE',
];

$baseConfig['atlas']['ORM']['aem1'] = [
    'driver'   => 'mysql',
    'host'     => 'localhost',
    'user'     => 'USERNAME',
    'password' => 'PASSWORD',
    'dbname'   => 'DATABASE',
];
```

Then, it would be possible to get the connection to the database by asking the Service Manager for it, from within a controller factory for example, in this way:

```
$dcm1 = $serviceManager['dcm1'];

// AND/OR

$dem1 = $serviceManager['dem1'];

// AND/OR
```

(continues on next page)

(continued from previous page)

```
$aem1 = $serviceManager['aem1'];
```

Note: Atlas and Doctrine DBAL and ORM objects are lazy-loaded, which avoids creating instances of these classes if they remain unused.

For more information on configuring the application's model, please see the *Models* section.

CHAPTER 15

Autoloading

The framework's autoload is managed by **Composer**. By default, the LightMVC Framework uses PSR-4 compliant autoloading. To add new namespaces within a LightMVC application, it is necessary to declare these namespace mappings within the application's `composer.json` file. For example, here are the namespaces of the **LightMVC Skeleton Application**:

```
"autoload": {
  "psr-4": {
    "Application\\Models\\": "models/Application/Models",
    "Application\\Middleware\\": "middleware/Application/Middleware",
    "Application\\Controllers\\": "controllers/Application/Controllers",
    "Application\\Services\\": "controllers/Application/Services",
    "Specialmodule\\Controllers\\": "controllers/Specialmodule/Controllers"
  }
},
```

Therefore, adding new namespaces is simply a question of adding new entries in this part of the file and running the following command from a CLI:

```
$ composer update
```

Note: A PSR-4 autoloader class is available within the framework if you wish to use configuration files instead of Composer's autoloading capabilities.

CHAPTER 16

HTTP Messages

The LightMVC Framework's HTTP message objects are the `\Zend\Diactoros\ServerRequest` and the `\Zend\Diactoros\Response` objects. These are PSR-7 compliant classes and are compatible with PSR-15 compliant middleware.

16.1 ServerRequest Object

In order to get a better understanding of the `ServerRequest` object, please see the ZF documentation on server-side applications.

16.2 Response Object

The `Response` object makes it possible to add headers and provide content to the application's final response to the client. Here is a simple example in order to do so:

```
$response = new Zend\Diactoros\Response();

// Write to the response body:
$response->getBody()->write("Hello");

// Multiple calls to write() append:
$response->getBody()->write(" World"); // now "Hello World"

// Add headers
// Note: headers can be added to the response after the data has been written to the␣
↪body
$response = $response
    ->withHeader('Content-Type', 'text/plain')
    ->withAddedHeader('X-Custom-Header', 'example');
```

For further reading on the `Response` object, please see the ZF documentation.

CHAPTER 17

Services

This framework's main services are:

- Event Manager (`\Ascmvc\Mvc\AscmvcEventManager`),
- Service Manager (`\Pimple\Container`).

17.1 Event Manager

The `\Ascmvc\Mvc\AscmvcEventManager` event manager is an extension of the `\Zend\EventManager\EventManager`. It is available through the application object's `getEventManager()` method. It is configured **WITH** a `\Zend\EventManager\SharedEventManager`. It is possible to get the shared manager by calling the main event manager's `getSharedManager()` method. This same shared manager will also be readily available within each controller aggregate by getting it from the controller's PSR-14 event dispatcher (event bus) like so:

```
// From within a controller's action method for example.
$sharedEventManager = $this->eventDispatcher->getSharedManager();
```

By doing so, it becomes possible to dispatch custom events not only to other parts of the current aggregate, but to also dispatch custom events to other aggregates outside of the current controller aggregate. Thus, Aspect-Oriented Programming becomes a clear possibility and allows for separation of concerns and code modularity.

Note: Each controller has access to a segregated event dispatcher (event bus), as the controller is considered to be the Aggregate Root of its event sourcing aggregate.

For more information on configuring the controller's event dispatcher, please see the *Event Sourcing Configuration* section.

The main `AscmvcEventManager` is designed to be able to trigger `\Ascmvc\Mvc\AscmvcEvent` events for the entire application. The `\Ascmvc\Mvc\AscmvcEvent` class is an extension of the `Zend\EventManager\Event` class. Here is a list of the framework's main MVC events:

```
/**#@+
 * Mvc events triggered by the Event Manager
 */
const EVENT_BOOTSTRAP      = 'bootstrap';
const EVENT_ROUTE          = 'route';
const EVENT_DISPATCH       = 'dispatch';
const EVENT_RENDER         = 'render';
const EVENT_FINISH         = 'finish';
/**#@-*/
```

These events correspond to listener interfaces that are implemented by default in each and every controller. Thus, from within any controller, it is possible to tap into a specific MVC event, or to downright interrupt the application's flow by returning a `\Zend\Diactoros\Response`, from within these listener methods.

Here is a short description of each main event:

- EVENT_BOOTSTRAP (onBootstrap): this event is triggered right after the booting and initialization phases of the application. Using the onBootstrap method within a controller class makes it possible to run code immediately after the middleware pipeline. And, if you attach a listener to this event with a high priority, you can run code before the execution of any middleware, any controller or any service.
- EVENT_ROUTE (onRoute): this event is triggered after bootstrapping is done and the router class has been instantiated, but before the router actually tries to resolve the request URI to a handler.
- EVENT_DISPATCH (onDispatch): this event is triggered after the router has instantiated a controller manager with a requested handler, but before the controller manager actually hands control over to the requested controller.
- EVENT_RENDER (onRender): this event is triggered once the controller has returned its output, but before the output is parsed by the template managers.
- EVENT_FINISH (onFinish): this event is triggered once rendering is done and/or a response object is available (event short-circuit). This event allows to manipulate the response object before returning the response to the client.

Note: You should avoid as much as possible to use the onBootstrap() method within the controller classes, as this would not scale very well if there is a large number of controllers.

Note: If you run the framework using **Swoole**, you should avoid using a high priority `AscmvcEvent::EVENT_FINISH` listeners to manipulate the response, because this event will be ignored by Swoole. To achieve the same result, one should use a very low priority listener on the `AscmvcEvent::EVENT_RENDER` event instead.

Here is an example of a controller that is tapping into the `AscmvcEvent::EVENT_BOOTSTRAP` event in order to short-circuit the application's execution and return an early response:

```
<?php

namespace Application\Controllers;

use Ascmvc\Mvc\AscmvcEvent;
use Ascmvc\Mvc\Controller;
use Zend\Diactoros\Response;

class FakeController extends Controller
```

(continues on next page)

(continued from previous page)

```
{
    public static function onBootstrap(AscmvcEvent $event)
    {
        $response = new Response();
        $response->getBody()->write('Hello World!');
        return $response;
    }

// [...]
```

In order to attach a new listener to one of the main MVC events, you can simply do it this way:

```
$this->event->getApplication()->getEventManager()->attach(AscmvcEvent::EVENT_RENDER,␣
↪function ($event) use ($serviceManager) {
    // do something here
}, 3);
```

Note: The last parameter is a priority indicator. The higher the indicator, the higher the priority of the listener. Any listener can be given a priority of three (3) or more in order to run **BEFORE** any of the preconfigured listeners.

To learn more about the LightMVC events and and corresponding listeners, please see the **LightMVC Framework**'s API documentation.

For more information on available methods of the `\Zend\EventManager\EventManager`, please see the ZF documentation, and the ZF API documentation.

17.2 Service Manager

The LightMVC Service Manager is an instance of the `\Pimple\Container` class. It is a simple implementation of a Registry and allows for easy storage and retrieval of objects and data. The Pimple container object implements the `\ArrayAccess` interface and thus, can be accessed as if it was an array.

Storing a service is as simple as:

```
// Store SomeService instance
$serviceManager['someService'] = function ($serviceManager) {
    return new SomeService();
};
```

And, retrieving the same service would be done as follows:

```
// Retrieve SomeService instance
$someService = $serviceManager['someService'];
```

It is possible to store a service within the container as a lazy-loading one. To do so, you must use the container's `factory()` method:

```
// Store SomeService instance
$serviceManager['someService'] = $serviceManager->factory(function ($serviceManager) {
    // Retrieve the database connection and inject it within the SomeService␣
↪constructor
    return new SomeService($serviceManager['em1']);
});
```

To learn more about **Pimple**, please visit the Pimple Website.

CHAPTER 18

Routing

The framework uses the nikic/fast-route library - `FastRoute` - as its main routing service.

All configuration must be given in the `$baseConfig` array, under the `routes` index.

Note: For more information on configuring the application's routes, please see the *Routing Configuration* section.

18.1 Routes

Defining a route is as simple as adding an integer-referenced array to the `routes` array contained within the `$baseConfig` array. This new integer-referenced array must contain three elements, in the following order: 1- an HTTP verb, 2- a URL, and 3- the name of the controller.

1. The HTTP verb can be one of `GET`, `POST`, `PUT`, `PATCH`, or `DELETE`. An array of multiple HTTP verbs can be given.

Note: If a page is requested with another HTTP verb than the ones that are defined in the corresponding route, the application will return a '405 - Method Not Allowed' header.

2. The URL can contain named placeholders. These are defined by using curly braces. By default, any placeholder named `action` will be mapped to the name of a controller's handler method. The default handler method of any controller is the `indexAction` action. By default, **LightMVC** is a 'single-action controller' framework.

The contents of any placeholder will be available within the $vars variable inside the controller's handler method, under an index with the name of the corresponding HTTP verb, and a sub-index with the name of the placeholder. Finally, any part of the URL can be defined as being optional by using square brackets.

Note: It is possible to use a regex in order to only allow some characters in the URL.

3. The name of the controller must be in lowercase and must map to an existing controller in order to avoid a runtime exception that will be thrown by the controller manager when it tries to get the requested handler (controller method).

Note: The controller name can contain a forward slash ('/') in order to reference to a module name. For example, `special/index` would map to the `IndexController` within the `Special\Controller\` namespace. When no forward slash is given, the default namespace is `Application\Controller\`.

Here is an example of a more advanced route configuration:

```
$baseConfig['routes'] = [
    2 => [
        ['GET', 'POST'],
        '/products[/{action}[/{id:[0-9]+}]]',
        'product',
    ],
];
```

In this example, only `GET` and `POST` requests will be allowed on any URL beginning with `/products`. The URL can also contain an `action` name, which will map to a request handler method within the controller, and an `id` parameter, which will contain at least one digit, ranging from 0 through 9. The contents of the `action` and `id` placeholders will be available in the $vars variable within the controller's request handler method: `$vars['get']['action']` and `$vars['get']['id']`.

For further reading on the `FastRoute` object, please see the FastRoute Code Repository.

18.2 Caching Routes

When running a **LightMVC** application in `production` mode (please see the *Configuration* section for more details), routes will be cached. It is therefore important to delete the `cache/routes.cache` file in order to refresh the cache if need be.

CHAPTER 19

Controllers

The framework's controllers are extensions of the `Ascmvc\Mvc\Controller` or the `Ascmvc\EventSourcing\AggregateRootController` classes, which both implement the `Ascmvc\AscmvcEventManagerListenerInterface` interface. The `AggregateRootController` also implements the `Ascmvc\EventSourcing\AggregateEventListenerInterface`. Within the LightMVC Framework, controllers are considered to be the Aggregate Root (main command) of the each and every event sourcing aggregate.

Note: For more information on configuring an application's event sourcing aggregates and the application's event log, please see the *Event Sourcing Configuration* section.

Note: For more information on the framework's event sourcing aggregates in general, please see the *Event Sourcing* section.

19.1 Controller Methods

Every controller has the following basic concrete definition:

```
class Controller extends AbstractController implements␣
↪AscmvcEventManagerListenerInterface
{
    public function __construct(array $baseConfig)
    {
        $this->baseConfig = $baseConfig;

        $this->view = $this->baseConfig['view'];
    }

    public static function onBootstrap(AscmvcEvent $event)
```

(continues on next page)

(continued from previous page)

```
    {
    }

    public function onDispatch(AscmvcEvent $event)
    {
    }

    public function onRender(AscmvcEvent $event)
    {
    }

    public function onFinish(AscmvcEvent $event)
    {
    }

    public function indexAction($vars = null)
    {
    }
}
```

Thus, every controller has an `indexAction` request handler by default, and every controller has the ability to tap into any of the framework's major events, except the `AscmvcEvent::EVENT_ROUTE` event. Upon instantiation of the required controller by the controller manager (dispatcher), a minimal version of the application's `$baseConfig` array will be injected into the controller. Upon execution of the controller's request handler method, all the global server variables are injected into the handler through the `$vars` variable.

Note: One should avoid as much as possible to use the onBootstrap() method within the controller classes, as this would not scale very well if there is a large number of controllers.

For more information on the event manager and the main MVC events, please see the *Event Manager* section.

When extending the `Ascmvc\EventSourcing\AggregateRootController` class instead of the `Ascmvc\Mvc\Controller` class, a controller has the following additional concrete definition:

```
class AggregateRootController extends Controller
{
    /**
     * Contains the name of the Aggregate Root.
     *
     * @var string
     */
    protected $aggregateRootName;

    /**
     * Contains a list of listeners for this aggregate, where the key is the name of␣
↪the event
     * and the value is the FQCN of the class that is to become a listener of the␣
↪specified event.
     *
     * @var array
     */
    protected $aggregateListenerNames = [];

    public function __construct(array $baseConfig, EventDispatcher $eventDispatcher)
    {
```

(continues on next page)

(continued from previous page)

```
        parent::__construct($baseConfig, $eventDispatcher);

        $this->aggregateRootName = static::class;

        $aggregateIdentifiers[] = $this->aggregateRootName;

        if (isset($baseConfig['eventlog']) && $baseConfig['eventlog']['enabled'] ===␣
↪true) {
            $aggregateIdentifiers[] = EventLogger::class;
        }

        $eventDispatcher->setIdentifiers($aggregateIdentifiers);

        if (!empty($this->aggregateListenerNames)) {
            foreach ($this->aggregateListenerNames as $key => $listenerName) {
                if (is_string($key) && is_string($listenerName)) {
                    $eventDispatcher->attach(
                        $key,
                        $listenerName::getInstance($eventDispatcher)
                    );
                }
            }
        }

        $sharedEventManager = $eventDispatcher->getSharedManager();

        if (!is_null($sharedEventManager)) {
            $sharedEventManager->attach(
                $this->aggregateRootName,
                '*',
                [$this, 'onAggregateEvent']
            );
        }
    }

    /**
     * Runs before the controller's default action.
     *
     * @param null $vars
     *
     * @return mixed|void
     */
    public function preIndexAction($vars = null)
    {
    }
}
```

Essentially, these additional facilities allow for automatic configuration of the Aggregate Root, by setting the name of the Aggregate Root, by setting the Event Dispatcher's identifiers accordingly, by attaching all listeners found in the `$aggregateListenerNames` property to the specified events, and by attaching the Aggregate Root controller as a listener to all events dispatched within its own event sourcing aggregate. If event logging is enabled, it will also add the Event Logger's Aggregate Root name as an aggregate identifier in order to dispatch all of the current aggregate's events to this other aggregate. Adding more identifiers might also prove useful in case one needs to also dispatch all events from one aggregate to another.

Additionally, the 'pre' action methods allow for the dispatching of events before the actual call to the main action method. The naming convention for 'pre' methods is to capitalize the first letter of the name of the action method

and to add the prefix 'pre' in front of the name. Thus, the `indexAction()` method would have a 'pre' action method with the name `preIndexAction()`. The 'pre' method has access to the same environment variables as the controller's main request handler method, through the injection of the `$vars` variable.

Note: For further reading on the framework's event sourcing aggregates in general, please see the *Event Sourcing* section.

19.2 Controller Factories

Any controller can implement the `Ascmvc\AscmvcControllerFactoryInterface` interface and become a factory that can store a factory of itself in the service manager (**Pimple** container) and/or return an instance of itself to the controller manager, after completing some specific logic.

This is useful if you need to set up some specific service or resource before injecting it into an instance of the controller, or if you need to customize the way the event sourcing aggregates and listeners are configured and set up in general by overriding the `Ascmvc\EventSourcing\AggregateRootController` class' automatic configuration.

For a working example, please see the section on the *LightMVC Skeleton Application*.

For information on how to deal with other types of factories, please see the *Service Manager* section.

CHAPTER 20

Event Sourcing

The framework's event sourcing library allows you to set up an event sourcing infrastructure, that uses CQRS or not, based on controllers as the aggregate root (main command and main event listener) of each and every aggregate. The principle of it is that all of the software's underlying commands and interactions with other systems (internal or external) will be called through the dispatching of events. These events are named 'aggregate events' and can be any custom event that the controller can dispatch through an event dispatcher, also known as the event bus. The event dispatcher will then notify any listener that is attached to this event.

Typically, an event sourcing aggregate will have Read Model and Policy listeners. These event listeners will then call aggregate commands by passing any values received through the aggregate event. In order to make things more simple, these values are normally wrapped inside an immutable value object that allows for a unified interface to deal with these values throughout the entire aggregate life cycle.

Since all of the aggregate's underlying commands are called through events, logging the aggregate's activities is made much more simple. Auditing and monitoring are added benefits that come with event sourcing.

To read further on event sourcing, please see the Martin Fowler's definition and explanations on this subject.

Note: For more information on configuring an application's event sourcing aggregates and the application's event log, please see the *Event Sourcing Configuration* section.

20.1 Aggregates

An aggregate is defined as a collective of classes that work together in order to accomplish a specific task. When using aggregates with controllers as their aggregate root, this task, or common goal, is sending a response back for a specific request from the client.

Thus, the first step in order to start using aggregates is to define a new controller by extending the `Ascmvc\EventSourcing\AggregateRootController` class. This allows for automatic configuration of the aggregate root. Essentially, it will set the name of the Aggregate Root, set the Event Dispatcher's identifiers accordingly, attach all listeners found in the class' `$aggregateListenerNames` property, and attach the Aggregate Root controller as a listener to all events dispatched within the event sourcing aggregate. If event logging is enabled, it

will also add the Event Logger's Aggregate Root name as an aggregate identifier in order to dispatch all of the current aggregate's events to this other aggregate.

Note: For more information on this automatic configuration of the aggregate root, please see the section on *Controller Methods*.

It is also posible to define an aggregate manually, by extending the `Ascmvc\Mvc\Controller` class and by establishing the aggregate root from within a controller's `factory()` method when implementing the `\Ascmvc\AscmvcControllerFactoryInterface` interface, like so:

```
public static function factory(array &$baseConfig, EventDispatcher &$eventDispatcher,␣
↪Container &$serviceManager, &$viewObject)
{
    // Setting the identifiers of this Event Dispatcher (event bus).
    // Subscribing this controller (Aggregate Root) and the Event Sourcing Logger.
    $eventDispatcher->setIdentifiers(
        [
            SomeController::class,
        ]
    );

    // Do something else...

    $controller = new SomeController($baseConfig, $eventDispatcher);

    $sharedEventManager = $eventDispatcher->getSharedManager();

    // Attaching this controller's listener method to the shared event manager's
    // corresponding identifier (see above).
    $sharedEventManager->attach(
        SomeController::class,
        '*',
        [$controller, 'onAggregateEvent']
    );

    return $controller;
}
```

By setting the event dispatcher's identifier to the controller class' fully-qualified class name (FQCN) and by attaching the controller's listener method `onAggregateEvent` with the controller's name as the aggregate root's name (first parameter of the event dispatcher's `attach()` method) and with a wildcard symbol as the event' name (second parameter of the same `attach()` method), we are, in fact, making this controller a listener to all of the aggregates events. This will allow the controller to determine what is left to be done, before a response can be considered to be completely finished. In an asynchronous environment like Swoole, this allows for simultaneous execution of multiple parts of the aggregate, without having to wait for one part to finish before another one can be executed.

Each part of the aggregate is then responsible of accomplishing its own subordinated task in order to fulfill the common goal. The way each part of the aggregate can interact with the other parts is by dispatching events through the event dispatcher.

20.2 Event Dispatcher

The default LightMVC event dispatcher is an instance of the `\Ascmvc\EventSourcing\EventDispatcher` class. It is a PSR-14 compliant event dispatcher. Therefore, you can replace this event dispatcher with

any other PSR-14 compliant event dispatcher. Since the LightMVC event dispatcher is an extension of the `\Zend\EventManager\EventManager`, it is possible to use any of the known Zend event manager facilities.

Note: For more information on configuring an application's event sourcing aggregates, please see the *Event Sourcing Configuration* section.

To dispatch aggregate events, it is a question of instantiating an aggregate value object and an aggregate event, and then using the event dispatcher's `dispatch()` method to dispatch it to the attached listeners.

```
// The value object can be empty.
$aggregateValueObject = new AggregateImmutableValueObject();

// The aggregate even must receive an aggregate value object,
// the name of aggregate root, and the name of the event.
$event = new AggregateEvent(
    $aggregateValueObject,
    ProductsController::class,
    ProductsController::READ_REQUESTED
);

$this->eventDispatcher->dispatch($event);
```

Note: If listeners are callables that return a `\Generator` instance, the Event Dispatcher will run these listeners asynchronously, with the lowest possible priority. This is very useful for long-lasting tasks that need to be executed as quickly as possible.

Note: Default aggregate listeners that are configured automatically by the aggregate root controller, using the `$aggregateListenerNames` property, MUST be invokable objects.

The event dispatcher contains an instance of the `\Zend\EventManager\SharedEventManager` by default. This allows for the dispatching of events to other parts of the application, or for listening to events dispatched by other parts of the application.

For more information on the shared event manager, please see the *Event Manager* section.

20.3 Aggregate Events

The LightMVC `\Ascmvc\EventSourcing\Event\AggregateEvent` class is, ultimately, an extension of the `\Zend\EventManager\Event` class. The added facilities allow the dispatching code to define the name of the aggregate root, and to inject an aggregate value object to be shared with listeners. The framework defines two child event classes: `\Ascmvc\EventSourcing\Event\ReadAggregateCompletedEvent` and `\Ascmvc\EventSourcing\Event\WriteAggregateCompletedEvent`. These two classes are designed to make logging easier and to allow for dispatching to the Read Model and Policy listeners more convenient.

20.4 Aggregate Value Objects

An `\Ascmvc\EventSourcing\AggregateImmutableValueObject` object is an immutable value object that is designed to allow all parts of an aggregate to easily share any data through a common interface. An aggregate

value object can be empty. Since this class implements the Serializable interface, it is possible to serialize its data into a string format. Finally, it allows its data to be hydrated into an array with its `hydrateToArray()` method.

20.5 Aggregate Event Listeners

All LightMVC listeners implement the `\Ascmvc\EventSourcing\EventListenerInterface` interface. This interface defines one single listener method named `onEvent()`. This being said, one can define any custom listener method, but the LightMVC event sourcing implementation recommends using the default `onEvent()` listener method for all event listeners. The framework offers an implementation of this interface which is named `\Ascmvc\EventSourcing\EventListener`. This class allows the extending listener object to benefit from the automatic injection of the controller's Event Dispatcher.

Moreover, the framework defines an `\Ascmvc\EventSourcing\AggregateEventListenerInterface` interface, that has an `onAggregateEvent()` listener method. The implementing class is named `\Ascmvc\EventSourcing\AggregateEventListener` and allows for automatic configuration of the event listener's `$aggregateRootName` property, depending on the name of the aggregate that dispatched the event.

There are two main types of listeners in the LightMVC event sourcing implementation. The `\Ascmvc\EventSourcing\ReadModel` class and the `\Ascmvc\EventSourcing\Policy` class. The framework also offers variants of these two main types when dealing with aggregates: the `\Ascmvc\EventSourcing\AggregateReadModel` and the `\Ascmvc\EventSourcing\AggregatePolicy` classes.

20.6 Aggregate Read Models

The `\Ascmvc\EventSourcing\ReadModel` class, or the `\Ascmvc\EventSourcing\AggregateReadModel` variant for aggregates, are to be used to call a command that will read data from a given source. The Read Model is responsible of determining what is the data source and how to access it.

Here is an example of an invokable non-blocking Aggregate Read Model that calls an asynchronous read command through the `\Ascmvc\EventSourcing\CommandRunner` command bus:

```
use Application\Events\ReadProductsCompleted;
use Application\Models\Entity\Products;
use Application\Models\Traits\DoctrineTrait;
use Ascmvc\EventSourcing\AggregateImmutableValueObject;
use Ascmvc\EventSourcing\AggregateReadModel;
use Ascmvc\EventSourcing\CommandRunner;
use Ascmvc\EventSourcing\Event\AggregateEvent;
use Ascmvc\EventSourcing\Event\Event;
use Ascmvc\EventSourcing\EventDispatcher;

class ProductsReadModel extends AggregateReadModel
{
    const READ_COMPLETED = 'products_read_completed';

    use DoctrineTrait;

    protected $id;

    protected $products;

    protected $productsRepository;
```

(continues on next page)

(continued from previous page)

```
    protected $commandRunner;

    protected function __construct(EventDispatcher $eventDispatcher, Products
↪$products)
    {
        parent::__construct($eventDispatcher);

        $this->products = $products;
    }

    public static function getInstance(EventDispatcher $eventDispatcher)
    {
        $productsEntity = new Products();

        return new self($eventDispatcher, $productsEntity);
    }

    public function __invoke(AggregateEvent $event)
    {
        if (is_null($this->commandRunner)) {
            $this->onAggregateEvent($event);

            $app = $event->getApplication();

            $valuesArray = $event->getAggregateValueObject()->getProperties();

            $arguments = [];

            if (!empty($valuesArray)) {
                $values = $event->getAggregateValueObject()->serialize();

                $arguments = [
                    '--values' => $values,
                ];
            }

            $swoole = $app->isSwoole();

            $this->commandRunner = new CommandRunner($app, 'products:read',
↪$arguments, $swoole);
        }

        while ($this->commandRunner->start()) {
            yield true;
        }

        $processStdout = $this->commandRunner->getOutput();
        //$processStderr = $this->commandProcess->getError();

        $aggregateValueObject = new AggregateImmutableValueObject();

        if (!empty(trim($processStdout))) {
            $aggregateValueObject = $aggregateValueObject->unserialize(
↪$processStdout);
        }
```

(continues on next page)

(continued from previous page)

```
        $event = new ReadProductsCompleted(
            $aggregateValueObject,
            $event->getAggregateRootName(),
            ProductsReadModel::READ_COMPLETED
        );

        $this->eventDispatcher->dispatch($event);

        return;
    }

    public function onAggregateEvent(AggregateEvent $event)
    {
        parent::onAggregateEvent($event);
    }

    public function onEvent(Event $event)
    {
    }
}
```

Here is another example of a Read Model that calls a blocking read command, by passing to it all the necessary data, and the required database entity manager, in order for the command to successfully execute itself and retrieve data from a 'products' table in the database:

```
<?php

namespace Application\ReadModels;

use Application\Commands\ReadProductsCommand;
use Application\Models\Entity\Products;
use Application\Models\Traits\DoctrineTrait;
use Ascmvc\EventSourcing\Event\Event;
use Ascmvc\EventSourcing\EventDispatcher;
use Ascmvc\EventSourcing\ReadModel;

class ProductsReadModel extends ReadModel
{
    use DoctrineTrait;

    protected $id;

    protected $products;

    protected $productsRepository;

    protected function __construct(EventDispatcher $eventDispatcher, Products
↪$products)
    {
        parent::__construct($eventDispatcher);

        $this->products = $products;
    }

    public static function getInstance(EventDispatcher $eventDispatcher)
    {
```

(continues on next page)

(continued from previous page)

```
        $productsEntity = new Products();

        return new self($eventDispatcher, $productsEntity);
    }

    public function onEvent(Event $event)
    {
        // The read connection can be different from the write connection if␣
↪implementing full CQRS.
        $connName = $event->getApplication()->getBaseConfig()['events']['read_conn_
↪name'];

        $entityManager = $event->getApplication()->getServiceManager()[$connName];

        $productsCommand = new ReadProductsCommand(
            $event->getAggregateValueObject(),
            $entityManager,
            $this->eventDispatcher
        );

        if (!is_null($productsCommand)) {
            $productsCommand->execute();
        }

        return;
    }
}
```

If the listener is named inside the `Ascmvc\EventSourcing\AggregateRootController` class' `$aggregateListenerNames` array property, it will automatically be called upon when the specified event occurs.

If configuring the aggregate manually, one must, from within the controller's `factory()` method (or any other main `AscmvcEvent` method), attach the Read Model to the aggregate's event bus (event dispatcher) in this way:

```
// Controller's factory() method

// Manually attach an invokable listeners if needed
$someReadModel = SomeReadModel::getInstance($eventDispatcher);

$eventDispatcher->attach(
    ProductsController::READ_REQUESTED,
    $someReadModel
);
```

Thus, the Read Model will listen for any event with the name `ProductsController::READ_REQUESTED` from within this aggregate.

20.7 Aggregate Policies

The `\Ascmvc\EventSourcing\Policy` class, or the `\Ascmvc\EventSourcing\AggregatePolicy` class, are to be used to call a command that will write data to a given source. The Policy is responsible of determining what data to write, where to store it and how to access the storage.

Here is an example of an invokable non-blocking Aggregate Policy that calls an asynchronous write command through

the `\Ascmvc\EventSourcing\CommandRunner` command bus and then, dispatches a new event by including the output from the command:

```
use Application\Controllers\ProductsController;
use Application\Events\WriteProductsCompleted;
use Application\Models\Traits\DoctrineTrait;
use Ascmvc\EventSourcing\AggregateImmutableValueObject;
use Ascmvc\EventSourcing\AggregatePolicy;
use Ascmvc\EventSourcing\CommandRunner;
use Ascmvc\EventSourcing\Event\AggregateEvent;
use Ascmvc\EventSourcing\Event\Event;
use Ascmvc\EventSourcing\EventDispatcher;

class ProductsPolicy extends AggregatePolicy
{
    const CREATE_COMPLETED = 'products_create_completed';

    const UPDATE_COMPLETED = 'products_update_completed';

    const DELETE_COMPLETED = 'products_delete_completed';

    use DoctrineTrait;

    protected $properties;

    protected $products;

    protected $productsRepository;

    protected $commandRunner;

    public static function getInstance(EventDispatcher $eventDispatcher)
    {
        return new self($eventDispatcher);
    }

    public function __invoke(AggregateEvent $event)
    {
        if (is_null($this->commandRunner)) {
            $this->onAggregateEvent($event);

            $app = $event->getApplication();

            $name = $event->getName();

            $execute = '';

            if ($name === ProductsController::CREATE_REQUESTED) {
                $execute = 'create';
            } elseif ($name === ProductsController::UPDATE_REQUESTED) {
                $execute = 'update';
            } elseif ($name === ProductsController::DELETE_REQUESTED) {
                $execute = 'delete';
            }

            $valuesArray = $event->getAggregateValueObject()->getProperties();

            $arguments = [];
```

(continues on next page)

(continued from previous page)

```
            if (!empty($valuesArray)) {
                $values = $event->getAggregateValueObject()->serialize();

                $arguments = [
                    'execute' => $execute,
                    '--values' => $values,
                ];
            }

            $swoole = $app->isSwoole();

            $this->commandRunner = new CommandRunner($app, 'products:write',
↪$arguments, $swoole);
        }

        while ($this->commandRunner->start()) {
            yield true;
        }

        $processStdout = $this->commandRunner->getOutput();
        //$processStderr = $this->commandProcess->getError();

        if (!empty($processStdout)) {
            $processStdoutArray = unserialize($processStdout);

            if (isset($processStdoutArray['data'])) {
                $valueObjectProperties = $processStdoutArray['data'];
            }
        } else {
            $valueObjectProperties = [];
        }

        $name = $event->getName();

        $aggregateValueObject = new AggregateImmutableValueObject(
↪$valueObjectProperties);

        if ($name === ProductsController::CREATE_REQUESTED) {
            $event = new WriteProductsCompleted(
                $aggregateValueObject,
                $event->getAggregateRootName(),
                ProductsPolicy::CREATE_COMPLETED
            );
        } elseif ($name === ProductsController::UPDATE_REQUESTED) {
            $event = new WriteProductsCompleted(
                $aggregateValueObject,
                $event->getAggregateRootName(),
                ProductsPolicy::UPDATE_COMPLETED
            );
        } elseif ($name === ProductsController::DELETE_REQUESTED) {
            $event = new WriteProductsCompleted(
                $aggregateValueObject,
                $event->getAggregateRootName(),
                ProductsPolicy::DELETE_COMPLETED
            );
        }
```

(continues on next page)

(continued from previous page)

```
        $eventParams = $processStdoutArray['params'];

        $event->setParams($eventParams);

        $this->eventDispatcher->dispatch($event);

        return;
    }

    public function onAggregateEvent(AggregateEvent $event)
    {
        parent::onAggregateEvent($event);
    }

    public function onEvent(Event $event)
    {
    }
}
```

Here is another example of a Policy that calls a blocking write command, by passing to it all the necessary data, and the required database entity manager, in order for the command to successfully execute itself and store the data to a 'products' table in the database:

```
<?php

namespace Application\Policies;

use Application\Commands\WriteProductsCommand;
use Application\Models\Traits\DoctrineTrait;
use Ascmvc\EventSourcing\Event\Event;
use Ascmvc\EventSourcing\EventDispatcher;
use Ascmvc\EventSourcing\Policy;

class ProductsPolicy extends Policy
{
    use DoctrineTrait;

    protected $properties;

    protected $products;

    protected $productsRepository;

    public static function getInstance(EventDispatcher $eventDispatcher)
    {
        return new self($eventDispatcher);
    }

    public function onEvent(Event $event)
    {
        $connName = $event->getApplication()->getBaseConfig()['events']['write_conn_
↪name'];

        $entityManager = $event->getApplication()->getServiceManager()[$connName];
```

(continues on next page)

(continued from previous page)

```
        $argv['name'] = $event->getName();

        $productsCommand = new WriteProductsCommand(
            $event->getAggregateValueObject(),
            $entityManager,
            $this->eventDispatcher,
            $argv
        );

        $productsCommand->execute();

        return;
    }
}
```

If the listener is named inside the `Ascmvc\EventSourcing\AggregateRootController` class' `$aggregateListenerNames` array property, it will automatically be called upon when the specified event occurs.

If configuring the aggregate manually, one must, from within the controller's `factory()` method (or any other main `AscmvcEvent` method), attach the Read Model to the aggregate's event bus (event dispatcher) in this way:

```
// Controller's factory() method

$productsPolicy = ProductsPolicy::getInstance($eventDispatcher);

// If there are many listeners to attach, one may use a
// Listener Aggregate that implements the␣
↪\Zend\EventManager\ListenerAggregateInterface
// instead of attaching them one by one.
$eventDispatcher->attach(
    ProductsController::CREATE_REQUESTED,
    $somePolicy
);

$eventDispatcher->attach(
    ProductsController::UPDATE_REQUESTED,
    $somePolicy
);

$eventDispatcher->attach(
    ProductsController::DELETE_REQUESTED,
    $somePolicy
);
```

Note: To learn more about the `\Zend\EventManager\ListenerAggregateInterface` interface, please see the ZF documentation on Aggregate Listeners.

Thus, the Policy will listen for any of the above mentioned events from within this aggregate.

20.8 Aggregate Commands

Aggregate commands can be of two types: blocking or non-blocking. When using non-blocking commands, one should use an instance of the `\Ascmvc\EventSourcing\CommandRunner` class to run an `\Ascmvc\EventSourcing\AsyncCommand` command, which extends the `Symfony\Component\Console\Command\Command` class. To use an asynchronous command, one must give it a name and give the command a body. Here is an example of an async command:

```
use Application\Models\Entity\Products;
use Application\Models\Repository\ProductsRepository;
use Ascmvc\AbstractApp;
use Ascmvc\EventSourcing\AsyncCommand;
use Doctrine\ORM\Mapping\ClassMetadata;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class ReadProductsCommand extends AsyncCommand
{
    protected static $defaultName = 'products:read';

    public function __construct(AbstractApp $webapp)
    {
        // you *must* call the parent constructor
        parent::__construct($webapp);
    }

    protected function configure()
    {
        $this
            ->setName('products:read')
            ->setDescription("Query Doctrine for 'Products' entities.");
        $this
            // configure options
            ->addOption('values', null, InputOption::VALUE_REQUIRED, 'Specify a␣
↪serialized value object array to use.');
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $connName = $this->getWebapp()->getBaseConfig()['events']['read_conn_name'];

        $entityManager = $this->getWebapp()->getServiceManager()[$connName];

        $serializedAggregateValueObjectProperties = $input->getOption('values');

        if (!empty($serializedAggregateValueObjectProperties)) {
            $args = unserialize($serializedAggregateValueObjectProperties);
        } else {
            $args = [];
        }

        $productsRepository = new ProductsRepository(
            $entityManager,
            new ClassMetadata(Products::class)
        );
```

(continues on next page)

(continued from previous page)

```
        try {
            if (isset($args['id'])) {
                $result = $productsRepository->find($args['id']);

                if (!is_null($result)) {
                    $results[] = $productsRepository->hydrateArray($result);
                } else {
                    $results = [];
                }
            } else {
                $results = $productsRepository->findAll();
            }
        } catch (\Exception $e) {
            return 1;
        }

        if (!empty($results)) {
            $outputValues = serialize($results);
        } else {
            $outputValues = '';
        }

        $output->writeln($outputValues);
    }
}
```

Once the command is named and defined, one must add it to the `async_commands` index of the `$baseConfig` array. This way, the command bus (`\Ascmvc\EventSourcing\CommandRunner`) will know how to find the command. The command will then be executed according to the requirements of the PHP environment within which the command is called. When in a non-Swoole environment, the command will be forked using `Symfony\Component\Process\Process` and **ReactPHP**. Otherwise, the command bus will call the command from within Swoole's coroutine.

If one only wishes to call a simple blocking command, the `\Ascmvc\EventSourcing\Command` offers a very simple blueprint that defines common functionality to be used by all commands. Command classes should extend this base class and should represent an imperative that takes place within an aggregate. If one is to say "write this data about our products to the database", one should extend the `\Ascmvc\EventSourcing\Command` class and name the class `WriteProductsCommand` within the namespace of the aggregate. Once the command has finished executing itself, it should dispatch a new aggregate event in order to notify listeners that the command is finished. Here is an example of what a `WriteProductsCommand` class could look like:

```
<?php

namespace Application\Commands;

use Application\Controllers\ProductsController;
use Application\Events\WriteProductsCompleted;
use Application\Models\Entity\Products;
use Application\Models\Repository\ProductsRepository;
use Ascmvc\EventSourcing\AggregateImmutableValueObject;
use Doctrine\ORM\Mapping\ClassMetadata;

class WriteProductsCommand extends ProductsCommand
{
    public function execute()
    {
```

(continues on next page)

(continued from previous page)

```
        $name = $this->argv['name'];

        $args = $this->aggregateValueObject->getProperties();

        $productsRepository = new ProductsRepository(
            $this->entityManager,
            new ClassMetadata(Products::class)
        );

        $values = [];

        try {
            if ($name === ProductsController::CREATE_REQUESTED) {
                $productsRepository->save($args);
            } elseif ($name === ProductsController::UPDATE_REQUESTED) {
                $products = $this->entityManager->find(Products::class, $args['id']);

                $values['pre'] = [
                    'id' => $products->getId(),
                    'name' => $products->getName(),
                    'price' => $products->getPrice(),
                    'description' => $products->getDescription(),
                    'image' => $products->getImage(),
                ];

                $productsRepository->save($args, $products);
            } elseif ($name === ProductsController::DELETE_REQUESTED) {
                if (isset($args['id'])) {
                    $products = $this->entityManager->find(Products::class, $args['id
↪']);
                    $productsRepository->delete($products);
                }
            }

            $params = ['saved' => 1];

            $values['post'] = $args;

            $aggregateValueObject = new AggregateImmutableValueObject($values);

            if ($name === ProductsController::CREATE_REQUESTED) {
                $event = new WriteProductsCompleted(
                    $aggregateValueObject,
                    ProductsController::class,
                    ProductsController::CREATE_COMPLETED
                );
            } elseif ($name === ProductsController::UPDATE_REQUESTED) {
                $event = new WriteProductsCompleted(
                    $aggregateValueObject,
                    ProductsController::class,
                    ProductsController::UPDATE_COMPLETED
                );
            } elseif ($name === ProductsController::DELETE_REQUESTED) {
                $event = new WriteProductsCompleted(
                    $aggregateValueObject,
                    ProductsController::class,
                    ProductsController::DELETE_COMPLETED
```

(continues on next page)

(continued from previous page)

```
                );
            }

            $event->setParams($params);
        } catch (\Exception $e) {
            $event->setParam('error', 1);
        }

        $this->eventDispatcher->dispatch($event);
    }
}
```

This new class will then be ready to be called by a `\Ascmvc\EventSourcing\Policy` listener once the corresponding event will be dispatched by another object, whether it is the main command (controller action method) or a subordinate command.

20.9 Event Logger

LightMVC Framework's event sourcing implementation comes with `\Ascmvc\EventSourcing\EventLogger` that will log any event based on two criteria: 1- any aggregate that has added the `EventLogger` class name to its event bus identifiers, and 2- any whitelisted (or not blacklisted) event class type. Concerning this second criterium, the logger will log all events if no classes were whitelisted or blacklisted. If one class is whitelisted or blacklisted, the logger will blacklist by default.

Also, it is possible to log events to a different database if a Doctrine ORM connection name is defined for it in the application's configuration.

Note: For more information on configuring an application's event log, please see the *Event Sourcing Configuration* section.

For a working example of Event Sourcing and CQRS with LightMVC, please use our skeleton application as it is explained in the section on the *LightMVC Skeleton Application*.

CHAPTER 21

Views

By default, the framework uses the **Plates** template manager. **Twig** and **Smarty** are also available. In order to change the template manager, one only has to change the parameters in the `config/view.config.php` file.

For more information on configuring the template managers and view elements, please see the *View Configuration* section.

By default, a LightMVC application should hold two folders for the view scripts: a `templates` folder and a `templates_c` folder.

The `templates_c` folder is used by the **Twig** and **Smarty** template managers in order to store compiled versions of the templates. This template cache will only be active if the application is in `production` mode (see the *Configuration*). **Plates** does not use a cache by default.

For more information on **Plates**, please see the Plates website.

For more information on **Twig**, please see the Twig website.

For more information on **Smarty**, please see the Smarty website.

CHAPTER 22

Models

The framework offers two backend frameworks by default: **Doctrine** and **Atlas**. **Doctrine** is very useful when dealing with Domain Logic, and **Atlas** is very useful when dealing with Persistence Logic.

To learn how to configure these backend services in the LightMVC Framework, please see the *Model Configuration* section.

For more information on **Doctrine**, please see the Doctrine website.

For more information on **Atlas**, please see the Atlas website.

Code examples of how to use these backend frameworks are given in the *LightMVC Skeleton Application* application section of this documentation.

CHAPTER 23

Middleware

The LightMVC Framework uses **Zend Stratigility** for its middleware implementation. This implementation is therefore PSR-15 compliant.

Configuring middleware is very straightforward in the LightMVC Framework. In a `config/middleware.config.php` file, one might configure some middleware as per the following:

```
$baseConfig['middleware'] = [
    '/foo' => function ($req, $handler) {
        $response = new \Zend\Diactoros\Response();
        $response->getBody()->write('FOO!');

        return $response;
    },
    function ($req, $handler) {
        if (! in_array($req->getUri()->getPath(), ['/bar'], true)) {
            return $handler->handle($req);
        }

        $response = new \Zend\Diactoros\Response();
        $response->getBody()->write('Hello world!');

        return $response;
    },
    '/baz' => [
        \Application\Middleware\SessionMiddleware::class,
        \Application\Middleware\ExampleMiddleware::class,
    ],
];
```

Any callable or any class that implements the `\Psr\Http\Server\MiddlewareInterface` interface can be used as valid middleware.

```
use \Psr\Http\Server\MiddlewareInterface;

```

(continues on next page)

(continued from previous page)

```
class ExampleMiddleware implements MiddlewareInterface
{
    public function process(ServerRequestInterface $request, RequestHandlerInterface
↪$handler): ResponseInterface
    {
    }
}
```

Middleware can be configured with or without paths. When indicating a path as the name of the array index of the middleware, the middleware will only run if the path matches the request URI. If the middleware's array index is an integer, the middleware will run on every request. Finally, it is possible to stack middleware within the same array index. In this latter case, the LightMVC Framework will lazy-load this FIFO stack of middleware and will run it in the given order.

Note: Normally, the middleware pipeline should always return a valid PSR-7 compliant response object. Otherwise, the pipeline would throw an exception. In the case of the LightMVC Framework, the pipeline can quietly fail in order to allow the MVC components to handle the request.

By default, the middleware pipeline is attached as a listener to the `AscmvcEvent::EVENT_BOOTSTRAP` event and can be overridden by a high priority listener on this event.

For more information on **Zend Stratigility**, please see the Zend Stratigility website.

You can also see the *Middleware Configuration* section.

CHAPTER 24

Sessions

As mentioned in the *Session Configuration* section, setting up asynchronous PHP sessions with LightMVC is a matter of adding the appropriate configuration in the `config/session.config.php` file. But, one might need to use an asynchronous session in a customized way. To do so, it is necessary to instantiate a session `Config` object, constructor injecting into it any custom configuration array that might be deemed necessary, and obtaining an instance of the `SessionManager` object by requesting it through the `getSessionManager()` static method. Once this is done, it is a question of invoking the `SessionManager`'s `start()` method to get the session started. Here is a working example:

```
<?php

$config['session'] = [
    'enabled' => true,
    'psr6_cache_pool' => \Ascmvc\Session\Cache\DoctrineCacheItemPool::class,
    'doctrine_cache_driver' => \Doctrine\Common\Cache\FilesystemCache::class,
    'doctrine_filesystem_cache_directory' => __DIR__ . DIRECTORY_SEPARATOR . 'cache' .
↪ DIRECTORY_SEPARATOR,
    'session_name' => 'MYSESSION',
    'session_path' => '/',
    'session_id_length' => 32,
    'session_id_type' => 1,
    'session_storage_prefix' => 'myprefix',
    'session_expire' => 60 * 60, // 60 minutes
];

$config = new \Ascmvc\Session\Config($config['session']);
$sessionManager = \Ascmvc\Session\SessionManager::getSessionManager(null, null,
↪$config);

try {
    $sessionManager->start();
} catch (\Throwable $e){
    echo $e->getMessage();
}
```

(continues on next page)

(continued from previous page)

```
// Do something here! :)

// Manually persist the session data.
// This step is optional, as the session object will persist itself automatically
// before the PHP script exits and falls out of scope of the PHP runtime.
$sessionManager->persist();
```

CHAPTER 25

LightMVC Skeleton Application

A **LightMVC Framework Skeleton Application** is available as a working example of the framework itself.

25.1 Installation

To start using the **Skeleton Application**, simply clone it from Github:

```
$ git clone https://github.com/lightmvc/lightmvcskel
```

Note: Please make sure that the `public` folder is accessible to the Web server and that the `cache`, `logs` and `templates_c` folders are writable.

Once cloned, enter the directory and install the dependencies using **Composer**:

```
$ cd lightmvcskel
$ composer install
```

In order to use all the included features, one must create and populate the database. A sample database is included in the `data` folder.

Furthermore, sample virtual host configuration files for the **Apache** and **nginx** Web servers are included in this same folder.

25.2 File Structure

The application's file structure is quite straightforward:

- `bin`: contains scripts that will be used to fork and run commands.
- `cache`: contains files created by the application while running in `production` mode.

- `config`: contains all of the application's configuration files.
- `controllers`: contains all of the application's controllers, grouped by module (namespace).
- `data`: contains any data file (database backups, configuration files for external services, etc.).
- `lib`: contains any libraries that are not available through Composer.
- `logs`: contains all of the application's log files.
- `middleware`: contains all of the application's middleware classes.
- `models`: contains all of the application's model classes (entities, repositories, backend traits, etc.).
- `public`: contains the front controller file, the Swoole front controller file, and all other static files (CSS, JS, images, fonts, favicons, etc.).
- `templates`: contains all of the application's template files, grouped by template manager.
- `templates_c`: contains all of the application's compiled template files when running in `production` mode (except when using `Plates`).
- `vendor`: contains all of the application's installed dependencies through **Composer**.

25.3 Running on Swoole

The **LightMVC Framework Skeleton Application** can run on Swoole in order to make it lightning fast. In order to do so, you must make sure to install Swoole. From the CLI, as the root user, type the following:

```
$ pecl install swoole
```

After answering a few questions, Swoole will be compiled and installed. Then, as the root user, run the following command (on Linux/Unix/Mac):

```
$ echo "extension=swoole.so" >> /etc/php.ini
```

Note: If running **Swoole** on **Windows**, please add the extension manually in **PHP**'s `php.ini` file. The `php.ini` file might be located elsewhere on your system. For example, on **Ubuntu** 18.04, when running **PHP** 7.2, you will find this file in `/etc/php/7.2/apache2`. You can discover the location of this file by entering the command `php --ini` on the command line. It must also be mentioned that some systems have multiple INI files (CLI vs Web). Please modify all those that apply.

Then, from within the root directory of the project, you can run the following command:

```
$ COMPOSER_PROCESS_TIMEOUT=0 composer run-swoole
```

Note: By default, Swoole will listen on the `localhost` loopback, on port 9501. If you wish to change this, please modify the `run-swoole` command inside the `composer.json` file accordingly.

Have a lot of fun! :)

CHAPTER 26

Indices and tables

- genindex
- search

Index