Understanding the Router

Our module is coming along nicely. However, we're not really doing all that much yet; to be precise, all we do is display *all* blog entries on one page. In this chapter, you will learn everything you need to know about the `Router` in order to route to controllers and actions for displaying a single blog post, adding a new blog post, editing an existing post, and deleting a post.

Different route types

Before we go into details on our application, let's take a look at the most often used route types.

Literal routes

As mentioned in a previous chapter, a literal route is one that exactly matches a specific string. Examples of URLs that can utilize literal routes include:

- `http://domain.com/blog`
- `http://domain.com/blog/add`
- `http://domain.com/about-me`
- `http://domain.com/my/very/deep/page`
- `http://domain.com/my/very/deep/page`

Configuration for a literal route requires you to provide the path to match, and the "defaults" to return on a match. The "defaults" are then returned as route match parameters; one use case for these is

to specify the controller to invoke and the action method on that controller to use. As an example:

```
'router' => [
    'routes' => [
        'about' => [
            'type' => \Zend\Router\Http\Literal::class,
            'options' => [
                'route'    => '/about-me',
                'defaults' => [
                    'controller' => 'AboutMeController',
                    'action'     => 'aboutme',
                ],
            ],
        ],
    ],
],
```

Segment routes

Segment routes allow you to define routes with variable parameters; a common use case is for specifying an identifier in the path. Examples of URLs that might require segment routes include:

- `http://domain.com/blog/1` (parameter "1" is dynamic)
- `http://domain.com/blog/details/1` (parameter "1" is dynamic)
- `http://domain.com/blog/edit/1` (parameter "1" is dynamic)
- `http://domain.com/blog/1/edit` (parameter "1" is dynamic)
- `http://domain.com/news/archive/2014` (parameter "2014" is dynamic)
- `http://domain.com/news/archive/2014/january` (parameter "2014" and "january" are dynamic)

Configuring a segment route is similar to that of a literal route. The primary differences are:

- The route will have one or more `:<varname>` segments, indicating items that will be dynamically filled. `<varname>` should be a string, and will be used to identify the variable to return when routing is successful.
- The route *may* also contain *optional* segments, which are items surrounded by square braces (`[]`), and which can contain any mix of literal and variable segments internally.
- The "defaults" can include the names of variable segments; in case that segment is missing, the default will be used. (They can also be completely independent; for instance, the "controller" rarely should be included as a segment!).
- You may also specify "constraints" for each variable segment; each constraint will be a regular expression that must pass for matching to be successful.

As an example, let's consider a route where we want to specify a variable "year" segment, and indicate that the segment must contain exactly four digits; when matched, we should use the `ArchiveController` and its `byYear` action:

```
'router' => [
    'routes' => [
        'archives' => [
            'type' => \Zend\Router\Http\Segment::class,
            'options' => [
                'route'    => '/news/archive[/:year]',
                'defaults' => [
                    'controller' => ArchiveController::class,
                    'action'     => 'byYear',
                    'year'       => date('Y'),
                ],
                'constraints' => [
                    'year' => '\d{4}',
                ],
            ],
        ],
    ],
],
```

This configuration defines a route for a URL such as `//example.com/news/archive/2014`. The route contains the variable segment `:year`, which has a regex constraint defined as `\d{4}`, indicating it will match if and only if it is exactly four digits. As such, the URL `//example.com/news/archive/123` will fail to match, but `//example.com/news/archive/1234` will.

The definition marks an optional segment, denoted by `[/:year]`. This has a couple of implications. First, it means that we can also match:

- `//example.com/news/archive`
- `//example.com/news/archive/`

In both cases, we'll also still receive a value for the `:year` segment, because we defined a default for it: the expression `date('Y')` (returning the current year).

Segment routes allow you to dynamically match paths, and provide

extensive capabilities for how you shape those paths, matching variable segments, and providing constraints for them.

Different routing concepts

When thinking about an entire application, you'll quickly realize that you may have many, many routes to define. When writing these routes you have two options:

- Spend less time writing routes that in turn are a little slow in matching.
- Write very explicit routes that match faster, but require more work to define.

Generic routes

A generic route is greedy, and will match as many URLs as possible. A common approach is to write a route that matches the controller and action:

```
'router' => [
    'routes' => [
        'default' => [
            'type' => \Zend\Router\Http\Segment::class,
            'options' => [
                'route'    => '/[:controller[/:action]]',
                'defaults' => [
                    'controller' => Application\Controller\IndexController::class,
                    'action'     => 'index',
                ],
                'constraints' => [
                    'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                    'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
                ],
            ],
        ],
    ],
],
```

Let's take a closer look as to what has been defined in this configuration. The `route` part now contains two optional parameters, `controller` and `action`. The `action` parameter is optional only when the `controller` parameter is present. Both have constraints that ensure they only allow strings that would be valid PHP class and method names.

The big advantage of this approach is the immense time you save when developing your application; one route, and then all you need to do is create controllers, add action methods to them, and they are immediately available.

The downsides are in the details.

In order for this to work, you will need to use aliases when defining your controllers, so that you can alias shorter names that omit namespaces to the fully qualified controller class names; this sets up the potential for collisions between different application

modules which might define the same controller class names.

Second, matching nested optional segments, each with regular expression constraints, adds performance overhead to routing.

Third, such a route does not match any additional segments, constraining your controllers to omit dynamic route segments and instead rely on query string arguments for route parameters — which in turn leaves parameter validation to your controllers.

Finally, there is no guarantee that a valid match will result in a valid controller and action. As an example, if somebody requested `//example.com/strange/nonExistent`, and no controller maps to `strange`, or the controller has no `nonExistentAction()` method, the application will use more cycles to discover and report the error condition than it would if routing had simply failed to match. This is both a performance and a security consideration, as an attacker could use this fact to launch a Denial of Service.

Basic routing

By now, you should be convinced that generic routes, while nice for prototyping, should likely be avoided. That means defining explicit routes.

Your initial approach might be to create one route for every permutation:

```
'router' => [
    'routes' => [
        'news' => [
            'type' => \Zend\Router\Http\Literal::class,
            'options' => [
                'route'    => '/news',
                'defaults' => [
                    'controller' => NewsController::class,
                    'action'     => 'showAll',
                ],
            ],
        ],
        'news-archive' => [
            'type' => \Zend\Router\Http\Segment::class,
            'options' => [
                'route'    => '/news/archive[/:year]',
                'defaults' => [
                    'controller' => NewsController::class,
                    'action'     => 'archive',
                ],
                'constraints' => [
                    'year' => '\d{4}',
                ],
            ],
        ],
        'news-single' => [
            'type' => \Zend\Router\Http\Segment::class,
            'options' => [
                'route'    => '/news/:id',
                'defaults' => [
                    'controller' => NewsController::class,
                    'action'     => 'detail',
                ],
                'constraints' => [
                    'id' => '\d+',
                ],
            ],
        ],
    ],
],
```

Routing is done as a stack, meaning last in, first out (LIFO). The

trick is to define your most general routes first, and your most specific routes last. In the example above, our most general route is a literal match against the path `/news`. We then have two additional routes that are more specific, one matching `/news/archive` (with an optional segment for the year), and another one matching `/news/:id`. These exhibit a fair bit of repetition:

- In order to prevent naming collisions between routes, each route name is prefixed with `news-`.
- Each routing string contains `/news`.
- Each defines the same default controller.

Clearly, this can get tedious. Additionally, if you have many routes with repitition such as this, you need to pay special attention to the stack and possible route overlaps, as well as performance (if the stack becomes large).

Child routes

To solve the problems detailed in the last section, zend-router allows defining "child routes". Child routes inherit inherit all `options` from their respective parents; this means that if an option, such as the controller default, doesn't change, you do not need to redefine it.

Additionally, child routes match *relative* to the parent route. This provides several optimizations:

- You do not need to duplicate common path segments.
- Routing will ignore the child routes *unless the parent matches*, which can provide enormous performance benefits during routing.

Let's take a look at a child routes configuration using the same

example as above:

```
'router' => [
    'routes' => [
        'news' => [
            // First we define the basic options for the parent route:
            'type' => \Zend\Router\Http\Literal::class,
            'options' => [
                'route'    => '/news',
                'defaults' => [
                    'controller' => NewsController::class,
                    'action'     => 'showAll',
                ],
            ],

            // The following allows "/news" to match on its own if no child
            // routes match:
            'may_terminate' => true,

            // Child routes begin:
            'child_routes' => [
                'archive' => [
                    'type' => \Zend\Router\Http\Segment::class,
                    'options' => [
                        'route'    => '/archive[/:year]',
                        'defaults' => [
                            'action' => 'archive',
                        ],
                        'constraints' => [
                            'year' => '\d{4}',
                        ],
                    ],
                ],
                'single' => [
                    'type' => \Zend\Router\Http\Segment::class,
                    'options' => [
                        'route'    => '/:id',
                        'defaults' => [
                            'action' => 'detail',
                        ],
                        'constraints' => [
                            'id' => '\d+',
                        ],
```

```
                ],
            ],
        ],
    ],
],
```

At its most basic, we define a parent route as normal, and then add an additional key, `child_routes`, which is normal routing configuration for additional routes to match if the parent route matches.

The `may_terminate` configuration key is used to determine if the parent route is allowed to match on its own; in other words, if no child routes match, is the parent route a valid route match? The flag is `false` by default; setting it to `true` allows the parent to match on its own.

The `child_routes` themselves look like standard routing at the top-level, and follow the same rules; they themselves can have child routes, too! The thing to remember is that any routing strings defined *are relative to the parent*. As such, the above definition allows matching any of the following:

- `/news`
- `/news/archive`
- `/news/archive/2014`
- `/news/42`

(If `may_terminate` was set to `false`, the first path above, `/news`, *would not match*.)

You'll note that the child routes defined above do not specify a `controller` default. Child routes *inherit options* from the parent, however, which means that, effectively, each of these will use the

same controller as the parent!

The advantages to using child routes include:

- Explicit routes mean fewer error conditions with regards to matching controllers and action methods.
- Performance; the router ignores child routes unless the parent matches.
- De-duplication; the parent route contains the common path prefix and common options.
- Organization; you can see at a glance all route definitions that start with a common path segment.

The primary disadvantage is the verbosity of configuration.

A practical example for our blog module

Now that we know how to configure routes, let's first create a route to display only a single blog entry based on internal identifier. Given that ID is a variable parameter, we need a segment route. Furthermore, we know that the route will also match against the same `/blog` path prefix, so we can define it as a child route of our existing route. Let's update our configuration:

```
// In module/Blog/config/module.config.php:
namespace Blog;

use Zend\Router\Http\Literal;
use Zend\Router\Http\Segment;
use Zend\ServiceManager\Factory\InvokableFactory;

return [
    'service_manager' => [ /* ... */ ],
    'controllers'     => [ /* ... */ ],
    'router'          => [
        'routes' => [
            'blog' => [
                'type' => Literal::class,
                'options' => [
                    'route'    => '/blog',
                    'defaults' => [
                        'controller' => Controller\ListController::class,
                        'action'     => 'index',
                    ],
                ],
                'may_terminate' => true,
                'child_routes'  => [
                    'detail' => [
                        'type' => Segment::class,
                        'options' => [
                            'route'    => '/:id',
                            'defaults' => [
                                'action' => 'detail',
                            ],
                            'constraints' => [
                                'id' => '[1-9]\d*',
                            ],
                        ],
                    ],
                ],
            ],
        ],
    ],
    'view_manager'    => [ /* ... */ ],
];
```

With this we have set up a new route that we use to display a single blog entry. The route defines a parameter, `id`, which needs to be a sequence of 1 or more positive digits, not beginning with 0.

The route will call the same `controller` as the parent route, but using the `detailAction()` method instead. Go to your browser and request the URL `http://localhost:8080/blog/2`; you'll see the following error message:

```
A 404 error occurred

Page not found.

The requested controller was unable to dispatch the request.

Controller:
Blog\Controller\ListController

No Exception available
```

This is due to the fact that the controller tries to access the `detailAction()`, which does not yet exist. We'll create this action now; go to your `ListController` and add the following action, which will return an empty view model

```
// In module/Blog/src/Controller/ListController.php:

/* .. */

class ListController extends AbstractActionController
{
    /* ... */

    public function detailAction()
    {
        return new ViewModel();
    }
}
```

Refresh your browser, which should result in the familiar message that a template was unable to be rendered.

Let's create this template now and assume that we will get a `Post` instance passed to the template to see the details of our blog. Create a new view file under `module/Blog/view/blog/list/detail.phtml`:

```
<h1>Post Details</h1>

<dl>
    <dt>Post Title</dt>
    <dd><?= $this->escapeHtml($this->post->getTitle()) ?></dd>

    <dt>Post Text</dt>
    <dd><?= $this->escapeHtml($this->post->getText()) ?></dd>
</dl>
```

The above template is expecting a `$post` variable referencing a `Post` instance in the view model. We'll now update the `ListController` to provide that:

```
public function detailAction()
{
    $id = $this->params()->fromRoute('id');

    return new ViewModel([
        'post' => $this->postRepository->findPost($id),
    ]);
}
```

If you refresh your application now, you'll see the details for our `Post` are displayed. However, there is one problem with what we have done: while we have our repository set up to throw an `InvalidArgumentException` when no post is found matching a given identifier, we do not check for it in our controller.

Go to your browser and open the URL `http://localhost:8080/blog/99`; you will see the following error message:

```
An error occurred

An error occurred during execution; please try again later.

Additional information:
InvalidArgumentException

File:
{projectPath}/module/Blog/src/Model/ZendDbSqlRepository.php:{lineNumber}

Message:
Blog post with identifier "99" not found.
```

This is kind of ugly, so our `ListController` should be prepared to do something whenever an `InvalidArgumentException` is thrown by the `PostService`. Let's have the controller redirect to the blog post overview.

First, add a new import to the `ListController` class file:

```
use InvalidArgumentException;
```

Now add the following try-catch statement to the `detailAction()` method:

```
public function detailAction()
{
    $id = $this->params()->fromRoute('id');

    try {
        $post = $this->postRepository->findPost($id);
    } catch (\InvalidArgumentException $ex) {
        return $this->redirect()->toRoute('blog');
    }

    return new ViewModel([
        'post' => $post,
    ]);
}
```

Now whenever a user requests an invalid identifier, you'll be redirected to the route `blog`, which is our list of blog posts!
