Models and the ServiceManager

In the previous chapter we've learned how to create a "Hello World" Application using zend-mvc. This is a good start, but the application itself doesn't really do anything. In this chapter we will introduce you into the concept of models, and with this, introduce zend-servicemanager.

What is a Model?

A model encapsulates application logic. This often entails *entity* or *value* objects representing specific *things* in our model, and *repositories* for retrieving and updating these objects.

For what we're trying to accomplish with our `Blog` module, this means that we need functionality for retrieving and saving blog posts. The posts themselves are our entities, and the repository will be what we retrieve them from and save them with. The model will get its data from some source; when writing the model, we don't really care about what the source actually is. The model will be written against an interface that we define and that future data providers must implement.

Writing the PostRepository

When writing a repository, it is a common best-practice to define an interface first. Interfaces are a good way to ensure that other programmers can easily build their own implementations. In other

words, they can write classes with identical function names, but which internally do completely different things, while producing the same expected results.

In our case, we want to create a `PostRepository`. This means first we are going to define a `PostRepositoryInterface`. The task of our repository is to provide us with data from our blog posts. For now, we are going to focus on the read-only side of things: we will define a method that will give us all posts, and another method that will give us a single post.

Let's start by creating the interface at `module/Blog/src/Model/PostRepositoryInterface.php`

```
namespace Blog\Model;

interface PostRepositoryInterface
{
    /**
     * Return a set of all blog posts that we can iterate over.
     *
     * Each entry should be a Post instance.
     *
     * @return Post[]
     */
    public function findAllPosts();

    /**
     * Return a single blog post.
     *
     * @param  int $id Identifier of the post to return.
     * @return Post
     */
    public function findPost($id);
}
```

The first method, `findAllPosts()`, will return return all posts, and the second method, `findPost($id)`, will return the post matching

the given identifier `$id`. What's new in here is the fact that we actually define a return value that doesn't exist yet. We will define this class at a later point; for now, we will create the `PostRepository` class.

Create the class `PostRepository` at `module/Blog/src/Model/PostRepository.php`; be sure to implement the `PostRepositoryInterface` and its required method (we will fill these in later). You then should have a class that looks like the following:

```
namespace Blog\Model;

class PostRepository implements PostRepositoryInterface
{
    /**
     * {@inheritDoc}
     */
    public function findAllPosts()
    {
        // TODO: Implement findAllPosts() method.
    }

    /**
     * {@inheritDoc}
     */
    public function findPost($id)
    {
        // TODO: Implement findPost() method.
    }
}
```

Create an entity

Since our `PostRepository` will return `Post` instances, we must create that class, too. Let's create `module/Blog/src/Model/Post.php`:

```
namespace Blog\Model;

class Post
{
    /**
     * @var int
     */
    private $id;

    /**
     * @var string
     */
    private $text;

    /**
     * @var string
     */
    private $title;

    /**
     * @param string $title
     * @param string $text
     * @param int|null $id
     */
    public function __construct($title, $text, $id = null)
    {
        $this->title = $title;
        $this->text = $text;
        $this->id = $id;
    }

    /**
     * @return int|null
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @return string
```

```
     */
    public function getText()
    {
        return $this->text;
    }

    /**
     * @return string
     */
    public function getTitle()
    {
        return $this->title;
    }
}
```

Notice that we only created getter methods; this is because each instance should be unchangeable, allowing us to cache instances in the repository as necessary.

Bringing Life into our PostRepository

Now that we have our entity in place, we can bring life into our `PostRepository` class. To keep the repository easy to understand, for now we will only return some hard-coded content from our `PostRepository` class directly. Create a property inside the `PostRepository` called `$data` and make this an array of our `Post` type. Edit `PostReepository` as follows:

```
namespace Blog\Model;

class PostRepository implements PostRepositoryInterface
{
    private $data = [
        1 => [
            'id'    => 1,
            'title' => 'Hello World #1',
            'text'  => 'This is our first blog post!',
        ],
        2 => [
            'id'    => 2,
            'title' => 'Hello World #2',
            'text'  => 'This is our second blog post!',
        ],
        3 => [
            'id'    => 3,
            'title' => 'Hello World #3',
            'text'  => 'This is our third blog post!',
        ],
        4 => [
            'id'    => 4,
            'title' => 'Hello World #4',
            'text'  => 'This is our fourth blog post!',
        ],
        5 => [
            'id'    => 5,
            'title' => 'Hello World #5',
            'text'  => 'This is our fifth blog post!',
        ],
    ];

    /**
     * {@inheritDoc}
     */
    public function findAllPosts()
    {
        // TODO: Implement findAllPosts() method.
    }

    /**
```

```
     * {@inheritDoc}
     */
    public function findPost($id)
    {
        // TODO: Implement findPost() method.
    }
}
```

Now that we have some data, let's modify our `find*()` functions to return the appropriate entities:

```
namespace Blog\Model;

use DomainException;

class PostRepository implements PostRepositoryInterface
{
    private $data = [
        1 => [
            'id'    => 1,
            'title' => 'Hello World #1',
            'text'  => 'This is our first blog post!',
        ],
        2 => [
            'id'    => 2,
            'title' => 'Hello World #2',
            'text'  => 'This is our second blog post!',
        ],
        3 => [
            'id'    => 3,
            'title' => 'Hello World #3',
            'text'  => 'This is our third blog post!',
        ],
        4 => [
            'id'    => 4,
            'title' => 'Hello World #4',
            'text'  => 'This is our fourth blog post!',
        ],
        5 => [
            'id'    => 5,
            'title' => 'Hello World #5',
            'text'  => 'This is our fifth blog post!',
        ],
    ];

    /**
     * {@inheritDoc}
     */
    public function findAllPosts()
    {
        return array_map(function ($post) {
            return new Post(
```

```
                $post['title'],
                $post['text'],
                $post['id']
            );
        }, $this->data);
    }

    /**
     * {@inheritDoc}
     */
    public function findPost($id)
    {
        if (! isset($this->data[$id])) {
            throw new DomainException(sprintf('Post by id "%s" not found', $id));
        }

        return new Post(
            $this->data[$id]['title'],
            $this->data[$id]['text'],
            $this->data[$id]['id']
        );
    }
}
```

Both methods now have appropriate return values. Please note that from a technical point of view, the current implementation is far from perfect. We will improve this repository in the future, but for now we have a working repository that is able to give us some data in a way that is defined by our `PostRepositoryInterface`.

Bringing the Service into the Controller

Now that we have our `PostRepository` written, we want to get access to this repository in our controllers. For this task, we will step foot into a new topic called "Dependency Injection" (DI).

When we're talking about dependency injection, we're talking about a way to get dependencies into our classes. The most

common form, "Constructor Injection", is used for all dependencies that are required by a class at all times.

In our case, we want to have our `ListController` somehow interact with our `PostRepository`. This means that the class `PostRepository` is a dependency of the class `ListController`; without the `PostRepository`, our `ListController` will not be able to function properly. To make sure that our `ListController` will always get the appropriate dependency, we will first define the dependency inside the `ListController` constructor. Modify `ListController` as follows:

```
namespace Blog\Controller;

use Blog\Model\PostRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;

class ListController extends AbstractActionController
{
    /**
     * @var PostRepositoryInterface
     */
    private $postRepository;

    public function __construct(PostRepositoryInterface $postRepository)
    {
        $this->postRepository = $postRepository;
    }
}
```

The constructor now has a required argument; we will not be able to create instances of this class anymore without providing a `PostRepositoryInterface` implementation. If you were to go back to your browser and reload your project with the url `localhost:8080/blog`, you'd see the following error message:

```
Catchable fatal error: Argument 1 passed to Blog\Controller\ListController::__construct()
must be an instance of Blog\Model\PostRepositoryInterface, none given,
called in {projectPath}/vendor/zendframework/src/Factory/InvokableFactory.php on line {lineNumber}
and defined in {projectPath}/module/Blog/src/Controller/ListController.php on line {lineNumber}
```

And this error message is expected. It tells you exactly that our `ListController` expects to be passed an implementation of the `PostRepositoryInterface`. So how do we make sure that our `ListController` will receive such an implementation? To solve this, we need to tell the application how to create instances of the `Blog\Controller\ListController`. If you remember back to when we created the controller, we mapped it to the `InvokableFactory` in the module configuration:

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

use Zend\ServiceManager\Factory\InvokableFactory;

return [
    'controllers'  => [
        'factories' => [
            Controller\ListController::class => InvokableFactory::class,
        ],
    ],
    'router' => [ /** Router Config */ ]
    'view_manager' => [ /** ViewManager Config */ ],
);
```

The `InvokableFactory` instantiates the mapped class using no constructor arguments. Since our `ListController` now has a required argument, we need to change this. We will now create a custom factory for our `ListController`. First, update the configuration as follows:

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

// Remove the InvokableFactory import statement

return [
    'controllers' => [
        'factories' => [
            // Update the following line:
            Controller\ListController::class => Factory\ListControllerFactory::class,
        ],
    ],
    'router' => [ /** Router Config */ ]
    'view_manager' => [ /** ViewManager Config */ ],
);
```

The above changes the mapping for the `ListController` to use a new factory class we'll be creating, `Blog\Factory\ListControllerFactory`. If you refresh your browser you'll see a different error message:

```
An error occurred

An error occurred during execution; please try again later.

Additional information:

Zend\ServiceManager\Exception\ServiceNotFoundException

File:
{projectPath}/zendframework/zend-servicemanager/src/ServiceManager.php:{lineNumber}

Message:

Unable to resolve service "Blog\Controller\ListController" to a factory; are you
certain you provided it during configuration?
```

This exception message indicates that the service container could not resolve the service to a factory, and asks if we provided it

during configuration. We did, so the end result is that the factory must not exist. Let's write the factory now.

Writing a Factory Class

Factory classes for zend-servicemanager may implement either `Zend\ServiceManager\Factory\FactoryInterface`, or be callable classes (classes that implement the `__invoke()` method); `FactoryInterface` itself defines the `__invoke()` method. The first argument is the application container, and is required; if you implement the `FactoryInterface`, you must also define a second argument, `$requestedName`, which is the service name mapping to the factory, and an optional third argument, `$options`, which will be any options provided by the controller manager at instantiation. In most situations, the last argument can be ignored; however, you can create re-usable factories by implementing the second argument, so this is a good one to consider when writing your factories! For our purposes, this is a one-off factory, so we'll only use the first argument. Let's implement our factory class:

```
// In /module/Blog/src/Factory/ListControllerFactory.php:
namespace Blog\Factory;

use Blog\Controller\ListController;
use Blog\Model\PostRepositoryInterface;
use Interop\Container\ContainerInterface;
use Zend\ServiceManager\Factory\FactoryInterface;

class ListControllerFactory implements FactoryInterface
{
    /**
     * @param ContainerInterface $container
     * @param string $requestedName
     * @param null|array $options
     * @return ListController
     */
    public function __invoke(ContainerInterface $container, $requestedName, array $options = null)
    {
        return new ListController($container->get(PostRepositoryInterface::class));
    }
}
```

The factory receives an instance of the application container, which, in our case, is a `Zend\ServiceManager\ServiceManager` instance; these also conform to `Interop\Container\ContainerInterface`, allowing re-use in other dependency injection systems if desired. We pull a service matching the `PostRepositoryInterface` fully qualified class name and pass it directly to the controller's constructor.

There's no magic happening; it's just PHP code.

Refresh your browser and you will see this error message:

```
An error occurred

An error occurred during execution; please try again later.

Additional information:

Zend\ServiceManager\Exception\ServiceNotFoundException

File:
{projectPath}/vendor/zendframework/zend-servicemanager/src/ServiceManager.php:{lineNumber}

Message:

Unable to resolve service "Blog\Model\PostRepositoryInterface" to a factory; are
you certain you provided it during configuration?
```

Exactly what we expected. Within our factory, the service `Blog\Model\PostRepositoryInterface` is requested but the `ServiceManager` doesn't know about it yet. Therefore it isn't able to create an instance for the requested name.

Registering Services

Registering other services follows the same pattern as registering a controller. We will modify our `module.config.php` and add a new key called `service_manager`; the configuration of this key is the same as that for the `controllers` key. We will add two entries, one for `aliases` and one for `factories`, as follows:

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

// Re-add the following import:
use Zend\ServiceManager\Factory\InvokableFactory;

return [
    // Add this section:
    'service_manager' => [
        'aliases' => [
            Model\PostRepositoryInterface::class => Model\PostRepository::class,
        ],
        'factories' => [
            Model\PostRepository::class => InvokableFactory::class,
        ],
    ],
    'controllers'  => [ /** Controller Config */ ],
    'router'       => [ /** Router Config */ ],
    'view_manager' => [ /** View Manager Config */ ],
];
```

This aliases `PostRepositoryInterface` to our `PostRepository` implementation, and then creates a factory for the `PostRepository` class by mapping it to the `InvokableFactory` (like we originally did for the `ListController`); we can do this as our `PostRepository` implementation has no dependencies of its own.

Try refreshing your browser. You should see no more error messages, but rather exactly the page that we have created in the previous chapter of the tutorial.

Using the repository in our controller

Let's now use the `PostRepository` within our `ListController`. For this we will need to overwrite the default `indexAction()` and return a view with the results from the `PostRepository`. Modify `ListController` as follows:

```
// In module/Blog/src/Controller/ListController.php:
namespace Blog\Controller;

use Blog\Model\PostRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;
// Add the following import statement:
use Zend\View\Model\ViewModel;

class ListController extends AbstractActionController
{
    /**
     * @var PostRepositoryInterface
     */
    private $postRepository;

    public function __construct(PostRepositoryInterface $postRepository)
    {
        $this->postRepository = $postRepository;
    }

    // Add the following method:
    public function indexAction()
    {
        return new ViewModel([
            'posts' => $this->postRepository->findAllPosts(),
        ]);
    }
}
```

First, please note that our controller imported another class, `Zend\View\Model\ViewModel`; this is what controllers will usually return within zend-mvc applications. `ViewModel` instances allow you to provide variables to render within your template, as well as indicate which template to use. In this case we have assigned a variable called `$posts` with the value of whatever the repository method `findAllPosts()` returns (an array of `Post` instances). Refreshing the browser won't change anything yet because we haven't updated our template to display the data.

ViewModels are not required

You do not actually need to return an instance of `ViewModel`; when you return a normal PHP array, zend-mvc internally converts it into a `ViewModel`. The following are equivalent:

```
// Explicit ViewModel:
return new ViewModel(['foo' => 'bar']);

// Implicit ViewModel:
return ['foo' => 'bar'];
```

Accessing View Variables

When pushing variables to the view, they are accessible in two ways: either using object notation (`$this->posts`) or implicitly as script-level variables `$posts`). The two approaches are equivalent; however, calling `$posts` results in a little round-trip through the renderer's `__get()` method. We often recommend using `$this` notation to visually differentiate between variables passed to the view, and those created within the script itself.

Let's modify our view to display a table of all blog posts that our repository returns:

```
<!-- Filename: module/Blog/view/blog/list/index.phtml -->
<h1>Blog</h1>

<?php foreach ($this->posts as $post): ?>
<article>
  <h1 id="post<?= $post->getId() ?>"><?= $post->getTitle() ?></h1>

  <p><?= $post->getText() ?></p>
</article>
<?php endforeach ?>
```

In the view script, we iterate over the posts passed to the view model. Since every single entry of our array is of type `Blog\Model\Post`, we can use its getter methods and render it.

After saving this file, refresh your browser, and you should now see a list of blog entries!

Summary

In this chapter, we learned:

- An approach to building the models for an application.
- A little bit about dependency injection.
- How to use zend-servicemanager to implement dependency injection in zend-mvc applications.
- How to pass variables to view scripts from the controller.

In the next chapter, we will take a first look at the things we should do when we want to get data from a database.
