Making Use of Forms and Fieldsets

So far all we have done is read data from the database. In a real-life application, this won't get us very far, as we'll often need to support the full range of full `Create`, `Read`, `Update` and `Delete` operations (CRUD). Typically, new data will arrive via web form submissions.

Form components

The zend-form (https://zendframework.github.io/zend-form/) and zend-inputfilter (https://zendframework.github.io/zend-inputfilter/) components provide us with the ability to create fully-featured forms and their validation rules. zend-form consumes zend-inputfilter internally, so let's take a look at the elements of zend-form that we will use for our application.

Fieldsets

`Zend\Form\Fieldset` models a reusable set of elements. You will use a `Fieldset` to create the various HTML inputs needed to map to your server-side entities. It is considered good practice to have one `Fieldset` for every entity in your application.

The `Fieldset` component, however, is not a form, meaning you will not be able to use a `Fieldset` without attaching it to the `Zend\Form\Form` instance. The advantage here is that you have one set of elements that you can re-use for as many forms as you like.

Forms

`Zend\Form\Form` is a container for all elements of your HTML `<form>`. You are able to add both single elements or fieldsets (modeled as `Zend\Form\Fieldset` instances).

Creating your first Fieldset

Explaining how zend-form works is best done by giving you real code to work with. So let's jump right into it and create all the forms we need to finish our `Blog` module. We start by creating a `Fieldset` that contains all the input elements that we need to work with our blog data:

- You will need one hidden input for the `id` property, which is only needed for editting and deleting data.
- You will need one text input for the `title` property.
- You will need one textarea for the `text` property.

Create the file `module/Blog/src/Form/PostFieldset.php` with the following contents:

```
<?php
namespace Blog\Form;

use Zend\Form\Fieldset;

class PostFieldset extends Fieldset
{
    public function init()
    {
        $this->add([
            'type' => 'hidden',
            'name' => 'id',
        ]);

        $this->add([
            'type' => 'text',
            'name' => 'title',
            'options' => [
                'label' => 'Post Title',
            ],
        ]);

        $this->add([
            'type' => 'textarea',
            'name' => 'text',
            'options' => [
                'label' => 'Post Text',
            ],
        ]);
    }
}
```

This new class creates an extension of `Zend\Form\Fieldset` that, in an `init()` method (more on this later), adds elements for each aspect of our blog post. We can now re-use this fieldset in as many forms as we want. Let's create our first form.

Creating the PostForm

Now that we have our `PostFieldset` in place, we can use it inside a

`Form`. The form will use the `PostFieldset`, and also include a submit button so that the user can submit the data.

Create the file `module/Blog/src/Form/PostForm.php` with the following contents:

```php
<?php
namespace Blog\Form;

use Zend\Form\Form;

class PostForm extends Form
{
    public function init()
    {
        $this->add([
            'name' => 'post',
            'type' => PostFieldset::class,
        ]);

        $this->add([
            'type' => 'submit',
            'name' => 'submit',
            'attributes' => [
                'value' => 'Insert new Post',
            ],
        ]);
    }
}
```

And that's our form. Nothing special here, we add our `PostFieldset` to the form, we add a submit button to the form, and nothing more.

Adding a new Post

Now that we have the `PostForm` written, it's time to use it. But there are a few more tasks left:

- We need to create a new controller `WriteController` which accepts the following instances via its constructor:
- a `PostCommandInterface` instance
- a `PostForm` instance
- We need to create an `addAction()` method in the new `WriteController` to handle displaying the form and processing it.
- We need to create a new route, `blog/add`, that routes to the `WriteController` and its `addAction()` method.
- We need to create a new view script to display the form.

Creating the WriteController

While we could re-use our existing controller, it has a different responsibility: it will be *writing* new blog posts. As such, it will need to emit *commands*, and thus use the `PostCommandInterface` that we have defined previously.

To do that, it needs to accept and process user input, which we have modeled in our `PostForm` in a previous section of this chapter.

Let's create this new class now. Open a new file, `module/Blog/src/Controller/WriteController.php`, and add the following contents:

```php
<?php
namespace Blog\Controller;

use Blog\Form\PostForm;
use Blog\Model\Post;
use Blog\Model\PostCommandInterface;
use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class WriteController extends AbstractActionController
{
    /**
     * @var PostCommandInterface
     */
    private $command;

    /**
     * @var PostForm
     */
    private $form;

    /**
     * @param PostCommandInterface $command
     * @param PostForm $form
     */
    public function __construct(PostCommandInterface $command, PostForm $form)
    {
        $this->command = $command;
        $this->form = $form;
    }

    public function addAction()
    {
    }
}
```

We'll now create a factory for this new controller; create a new file, `module/Blog/src/Factory/WriteControllerFactory.php`, with the following contents:

```php
<?php
namespace Blog\Factory;

use Blog\Controller\WriteController;
use Blog\Form\PostForm;
use Blog\Model\PostCommandInterface;
use Interop\Container\ContainerInterface;
use Zend\ServiceManager\Factory\FactoryInterface;

class WriteControllerFactory implements FactoryInterface
{
    /**
     * @param ContainerInterface $container
     * @param string $requestedName
     * @param null|array $options
     * @return WriteController
     */
    public function __invoke(ContainerInterface $container, $requestedName, array $options = null)
    {
        $formManager = $container->get('FormElementManager');
        return new WriteController(
            $container->get(PostCommandInterface::class),
            $formManager->get(PostForm::class)
        );
    }
}
```

The above factory introduces something new: the `FormElementManager`. This is a plugin manager implementation that is specifically for forms. We don't necessarily need to register our forms with it, as it will check to see if a requested instance is a form when attempting to pull one from it. However, it does provide a couple nice features:

- If the form or fieldset or element retrieved implements an `init()` method, it invokes that method after instantiation. This is useful, as that way we're initializing after we have all our dependencies injected, such as input filters. Our form and fieldset define this method!

- It ensures that the various plugin managers related to input validation are shared with the instance, a feature we'll be using later.

Finally, we need to configure the new factory; in `module/Blog/config/module.config.php`, add an entry in the `controllers` configuration section:

```
'controllers' => [
    'factories' => [
        Controller\ListController::class => Factory\ListControllerFactory::class,
        // Add the following line:
        Controller\WriteController::class => Factory\WriteControllerFactory::class,
    ],
],
```

Now that we have the basics for our controller in place, we can create a route to it:

```php
<?php
// 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' => '\d+',
                            ],
                        ],
                    ],

                    // Add the following route:
                    'add' => [
                        'type' => Literal::class,
                        'options' => [
                            'route'    => '/add',
                            'defaults' => [
                                'controller' => Controller\WriteController::class,
                                'action'     => 'add',
                            ],
                        ],
                    ],
                ],
            ],
        ],
    ],
    'view_manager'    => [ /* ... */ ],
];
```

Finally, we'll create a dummy template:

```
<!-- Filename: module/Blog/view/blog/write/add.phtml -->
<h1>WriteController::addAction()</h1>
```

Check-in

If you try to access the new route `localhost:8080/blog/add` you're supposed to see the following 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\PostCommandInterface" to a factory; are you certain you provide
```

If this is not the case, be sure to follow the tutorial correctly and carefully check all your files.

The error is due to the fact that we have not yet defined an *implementation* of our `PostCommandInterface`, much less wired the implementation into our application!

Let's create a dummy implementation, as we did when we first started working with repositories. Create the file `module/Blog/src/Model/PostCommand.php` with the following contents:

```
<?php
namespace Blog\Model;

class PostCommand implements PostCommandInterface
{
    /**
     * {@inheritDoc}
     */
    public function insertPost(Post $post)
    {
    }

    /**
     * {@inheritDoc}
     */
    public function updatePost(Post $post)
    {
    }

    /**
     * {@inheritDoc}
     */
    public function deletePost(Post $post)
    {
    }
}
```

Now add service configuration in

`module/Blog/config/module.config.php`:

```
'service_manager' => [
    'aliases' => [
        /* ... */
        // Add the following line:
        Model\PostCommandInterface::class => Model\PostCommand::class,
    ],
    'factories' => [
        /* ... */
        // Add the following line:
        Model\PostCommand::class => InvokableFactory::class,
    ],
],
```

Reloading your application now will yield you the desired result.

Displaying the form

Now that we have new controller working, it's time to pass this form to the view and render it. Change your controller so that the form is passed to the view:

```
// In /module/Blog/src/Controller/WriteController.php:
public function addAction()
{
    return new ViewModel([
        'form' => $this->form,
    ]);
}
```

And then we need to modify our view to render the form:

```
<!-- Filename: module/Blog/view/blog/write/add.phtml -->
<h1>Add a blog post</h1>

<?php
$form = $this->form;
$form->setAttribute('action', $this->url());
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formCollection($form);
echo $this->form()->closeTag();
```

The above does the following:

- We set the `action` attribute of the form to the current URL.
- We "prepare" the form; this ensures any data or error messages bound to the form or its various elements are injected and ready to use for display purposes.
- We render an opening tag for the form we are using.
- We render the contents of the form, using the `formCollection()` view helper; this is a convenience method with some typically sane default markup. We'll be changing it momentarily.
- We render a closing tag for the form.

Form method

HTML forms can be sent using `POST` and `GET`. zend-form defaults to `POST`. If you want to switch to `GET`:

```
$form->setAttribute('method', 'GET');
```

Refreshing the browser you will now see your form properly displayed. It's not pretty, though, as the default markup does not follow semantics for Bootstrap (which is used in the skeleton application by default). Let's update it a bit to make it look better; we'll do that in the view script itself, as markup-related concerns belong in the view layer:

```
<!-- Filename: module/Blog/view/blog/write/add.phtml -->
<h1>Add a blog post</h1>

<?php
$form = $this->form;
$form->setAttribute('action', $this->url());

$fieldset = $form->get('post');

$title = $fieldset->get('title');
$title->setAttribute('class', 'form-control');
$title->setAttribute('placeholder', 'Post title');

$text = $fieldset->get('text');
$text->setAttribute('class', 'form-control');
$text->setAttribute('placeholder', 'Post content');

$submit = $form->get('submit');
$submit->setAttribute('class', 'btn btn-primary');

$form->prepare();

echo $this->form()->openTag($form);
?>

<fieldset>
<div class="form-group">
    <?= $this->formLabel($title) ?>
    <?= $this->formElement($title) ?>
    <?= $this->formElementErrors()->render($title, ['class' => 'help-block']) ?>
</div>

<div class="form-group">
    <?= $this->formLabel($text) ?>
    <?= $this->formElement($text) ?>
    <?= $this->formElementErrors()->render($text, ['class' => 'help-block']) ?>
</div>
</fieldset>

<?php
echo $this->formSubmit($submit);
echo $this->formHidden($fieldset->get('id'));
echo $this->form()->closeTag();
```

The above adds HTML attributes to a number of the elements we've defined, and uses more specific view helpers to allow us to render the exact markup we want for our form.

However, if we're submitting the form all we see is our form being displayed again. And this is due to the simple fact that we didn't add any logic to the controller yet.

General form-handling logic for controllers

Writing a controller that handles a form workflow follows the same basic pattern regardless of form and entities:

1. You need to check if the HTTP request method is via `POST`, meaning if the form has been sent.
2. If the form has been sent, you need to:
3. pass the submitted data to your `Form` instance
4. validate the `Form` instance
5. If the form passes validation, you will:
6. persist the form data
7. redirect the user to either the detail page of the entered data, or to an overview page
8. In all other cases, you need to display the form, potentially with error messages.

Modify your `WriteController:addAction()` to read as follows:

```
public function addAction()
{
    $request   = $this->getRequest();
    $viewModel = new ViewModel(['form' => $this->form]);

    if (! $request->isPost()) {
        return $viewModel;
    }

    $this->form->setData($request->getPost());

    if (! $this->form->isValid()) {
        return $viewModel;
    }

    $data = $this->form->getData()['post'];
    $post = new Post($data['title'], $data['text']);

    try {
        $post = $this->command->insertPost($post);
    } catch (\Exception $ex) {
        // An exception occurred; we may want to log this later and/or
        // report it to the user. For now, we'll just re-throw.
        throw $ex;
    }

    return $this->redirect()->toRoute(
        'blog/detail',
        ['id' => $post->getId()]
    );
}
```

Stepping through the code:

- We retrieve the current request.
- We create a default view model containing the form.
- If we do not have a `POST` request, we return the default view model.
- We populate the form with data from the request.
- If the form is not valid, we return the default view model; at

this point, the form will also contain error messages.
- We create a `Post` instance from the validated data.
- We attempt to insert the post.
- On success, we redirect to the post's detail page.

Child route names

When using the various `url()` helpers provided in zend-mvc and zend-view, you need to provide the name of a route. When using child routes, the route name is of the form `<parent>/<child>` — i.e., the parent name and child name are separated with a slash.

Submitting the form right now will return into the following error

```
Fatal error: Call to a member function getId() on null in
{projectPath}/module/Blog/src/Controller/WriteController.php
on line {lineNumber}
```

This is because our stub `PostCommand` class does not return a new `Post` instance, violating the contract!

Let's create a new implementation to work against zend-db. Create the file `module/Blog/src/Model/ZendDbSqlCommand.php` with the following contents:

```php
<?php
namespace Blog\Model;

use RuntimeException;
use Zend\Db\Adapter\AdapterInterface;
use Zend\Db\Adapter\Driver\ResultInterface;
use Zend\Db\Sql\Delete;
use Zend\Db\Sql\Insert;
use Zend\Db\Sql\Sql;
use Zend\Db\Sql\Update;

class ZendDbSqlCommand implements PostCommandInterface
{
    /**
     * @var AdapterInterface
     */
    private $db;

    /**
     * @param AdapterInterface $db
     */
    public function __construct(AdapterInterface $db)
    {
        $this->db = $db;
    }

    /**
     * {@inheritDoc}
     */
    public function insertPost(Post $post)
    {
        $insert = new Insert('posts');
        $insert->values([
            'title' => $post->getTitle(),
            'text' => $post->getText(),
        ]);

        $sql = new Sql($this->db);
        $statement = $sql->prepareStatementForSqlObject($insert);
        $result = $statement->execute();
        if (! $result instanceof ResultInterface) {
            throw new RuntimeException(
                'Database error occurred during blog post insert operation'
            );
        }

        $id = $result->getGeneratedValue();

        return new Post(
            $post->getTitle(),
            $post->getText(),
            $result->getGeneratedValue()
        );
    }

    /**
     * {@inheritDoc}
     */
    public function updatePost(Post $post)
    {
    }

    /**
     * {@inheritDoc}
     */
    public function deletePost(Post $post)
    {
    }
}
```

In the `insertPost()` method, we do the following:

- We create a `Zend\Db\Sql\Insert` instance, providing it the table name.
- We add values to the `Insert` instance.
- We create a `Zend\Db\Sql\Sql` instance with the database adapter, and prepare a statement from our `Insert` instance.
- We execute the statement and check for a valid result.
- We marshal a return value.

Now that we have this in place, we'll create a factory for it; create the file `module/Blog/src/Factory/ZendDbSqlCommandFactory.php` with the following contents:

```
<?php
namespace Blog\Factory;

use Interop\Container\ContainerInterface;
use Blog\Model\ZendDbSqlCommand;
use Zend\Db\Adapter\AdapterInterface;
use Zend\ServiceManager\Factory\FactoryInterface;

class ZendDbSqlCommandFactory implements FactoryInterface
{
    public function __invoke(ContainerInterface $container, $requestedName, array $options = null)
    {
        return new ZendDbSqlCommand($container->get(AdapterInterface::class));
    }
}
```

And finally, we'll wire it up in the configuration; update the `service_manager` section of `module/Blog/config/module.config.php` to read as follows:

```
'service_manager' => [
    'aliases' => [
        Model\PostRepositoryInterface::class => Model\ZendDbSqlRepository::class,
        // Update the following alias:
        Model\PostCommandInterface::class => Model\ZendDbSqlCommand::class,
    ],
    'factories' => [
        Model\PostRepository::class => InvokableFactory::class,
        Model\ZendDbSqlRepository::class => Factory\ZendDbSqlRepositoryFactory::class,
        Model\PostCommand::class => InvokableFactory::class,
        // Add the following line:
        Model\ZendDbSqlCommand::class => Factory\ZendDbSqlCommandFactory::class,
    ],
],
```

Submitting your form again, it should process the form and redirect you to the detail page for the new entry!

Let's see if we an improve this a bit.

Using zend-hydrator with zend-form

In our controller currently, we have the following:

```
$data = $this->form->getData()['post'];
$post = new Post($data['title'], $data['text']);
```

What if we could automate that, so we didn't need to worry about:

- Whether or not we're using a fieldset
- What the form fields are named

Fortunately, zend-form features integration with zend-hydrator. This will allow us to return a `Post` instance when we retrieve the validated values!

Let's udpate our fieldset to provide a hydrator and a prototype object.

First, add two import statements to the top of the class file:

```
// In module/Blog/src/Form/PostFieldset.php:
use Blog\Model\Post;
use Zend\Hydrator\Reflection as ReflectionHydrator;
```

Next, update the `init()` method to add the following two lines:

```
// In /module/Blog/src/Form/PostFieldset.php:

public function init()
{
    $this->setHydrator(new ReflectionHydrator());
    $this->setObject(new Post('', ''));

    /* ... */
}
```

When you grab the data from this fiedlset, it will be returned as a `Post` instance.

However, we grab data *from the form*; how can we simplify that interaction?

Since we only have the one fieldset, we'll set it as the form's *base fieldset*. This hints to the form that when we retrieve data from it, it should return the values from the specified fieldset instead; since our fieldset returns the `Post` instance, we'll have exactly what we need.

Modify your `PostForm` class as follows:

```
// In /module/Blog/src/Form/PostForm.php:

public function init()
{
    $this->add([
        'name' => 'post',
        'type' => PostFieldset::class,
        'options' => [
            'use_as_base_fieldset' => true,
        ],
    ]);

    /* ... */
```

Let's update our `WriteController`; modify the `addAction()` method to replace the following two lines:

```
$data = $this->form->getData()['post'];
$post = new Post($data['title'], $data['text']);
```

to:

```
$post = $this->form->getData();
```

Everything should continue to work. The changes done serve the purpose of de-coupling the details of how the form is structured from the controller, allowing us to work directly with our entities at all times!

Conclusion

In this chapter, we've learned the fundamentals of using zend-form, including adding fieldsets and elements, rendering the form, validating input, and wiring forms and fieldsets to use entities.

In the next chapter we will finalize the CRUD functionality by creating the update and delete routines for the blog module.