Editing and Deleting Data - tutorials

In the previous chapter we've come to learn how we can use the zend-form and zend-db components for *creating* new data-sets. This chapter will focus on finalizing the CRUD functionality by introducing the concepts for *editing* and *deleting* data.

Binding Objects to Forms

The one fundamental difference between our "add post" and "edit post" forms is the existence of data. This means we need to find a way to get data from our repository into the form. Luckily, zend-form provides this via a **data-binding** feature.

In order to use this feature, you will need to retrieve a `Post` instance, and bind it to the form. To do this, we will need to:

- Add a dependency in our `WriteController` on our `PostRepositoryInterface`, from which we will retrieve our `Post`.
- Add a new method to our `WriteController`, `editAction()`, that will retrieve a `Post`, bind it to the form, and either display the form or process it.
- Update our `WriteControllerFactory` to inject the `PostRepositoryInterface`.

We'll begin by updating the `WriteController`:

- We will import the `PostRepositoryInterface`.

- We will add a property for storing the `PostRepositoryInterface` .
- We will update the constructor to accept the `PostRepositoryInterface` .
- We will add the `editAction()` implementation.

The final result will look like the following:

```
 <?php


namespace Blog\Controller;

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

class WriteController extends AbstractActionController
{

    private $command;


    private $form;


    private $repository;


    public function __construct(
        PostCommandInterface $command,
        PostForm $form,
        PostRepositoryInterface $repository
    ) {
        $this->command = $command;
        $this->form = $form;
```

```
        $this->repository = $repository;
    }

    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;
        }

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

        try {
            $post = $this->command->insertPost($post);
        } catch (\Exception $ex) {


            throw $ex;
        }

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

    public function editAction()
    {
        $id = $this->params()->fromRoute('id');
        if (! $id) {
            return $this->redirect()->toRoute('blog');
        }

        try {
```

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

        $this->form->bind($post);
        $viewModel = new ViewModel(['form' => $this->form]);

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

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

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

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

The primary differences between `addAction()` and `editAction()` are that the latter needs to first fetch a `Post`, and this post is *bound* to the form. By binding it, we ensure that the data is populated in the form for the initial display, and, once validated, the same instance is updated. This means that we can omit the call to `getData()` after validating the form.

Now we need to update our `WriteControllerFactory`. First, add a new import statement to it:

```
use Blog\Model\PostRepositoryInterface;
```

Next, update the body of the factory to read as follows:

```
public function __invoke(ContainerInterface $container, $requestedName,
array $options = null)
{
    $formManager = $container->get('FormElementManager');

    return new WriteController(
        $container->get(PostCommandInterface::class),
        $formManager->get(PostForm::class),
        $container->get(PostRepositoryInterface::class)
    );
}
```

The controller and model are now wired together, so it's time to turn to routing.

Adding the edit route

The edit route is identical to the `blog/detail` route we previously defined, with two exceptions:

- it will have a path prefix, `/edit`
- it will route to our `WriteController`

Update the 'blog' `child_routes` to add the new route:

```
use Zend\Router\Http\Segment;

return [
    'service_manager' => [  ],
    'controllers'     => [  ],
    'router'          => [
        'routes' => [
            'blog' => [


                'child_routes' => [


                    'edit' => [
```

```
                        'type' => Segment::class,
                        'options' => [
                            'route'    => '/edit/:id',
                            'defaults' => [
                                'controller' =>
Controller\WriteController::class,
                                'action'     => 'edit',
                            ],
                            'constraints' => [
                                'id' => '[1-9]\d*',
                            ],
                        ],
                    ],
                ],
            ],
        ],
    ],
    'view_manager'    => [  ],
];
```

Creating the edit template

Rendering the form remains essentially the same between the `add` and `edit` templates; the only difference between them is the form action. As such, we will create a new *partial* script for the form, update the `add` template to use it, and create a new `edit` template.

Create a new file, `module/Blog/view/blog/write/form.phtml`, with the following contents:

```
 <?php
$form = $this->form;
$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');
```

```
$submit->setValue($this->submitLabel);
$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();
```

Now, update the `add` template, `module/Blog/view/write/add.phtml` to read as follows:

```
 <h1>Add a blog post</h1>

<?php
$form = $this->form;
$form->setAttribute('action', $this->url());
echo $this->partial('blog/write/form', [
    'form' => $form,
    'submitLabel' => 'Insert new post',
]);
```

The above retrieves the form, sets the form action, provides a context-appropriate label for the submit button, and renders it with our new partial view script.

Next in line is the creation of the new template, `blog/write/edit`:

```
 <h1>Edit blog post</h1>

<?php
$form = $this->form;
$form->setAttribute('action', $this->url('blog/edit', [], true));
echo $this->partial('blog/write/form', [
    'form' => $form,
    'submitLabel' => 'Update post',
]);
```

The three differences between the `add` and `edit` templates are:

- The heading at the top of the page.
- The URI used for the form action.
- The label used for the submit button.

Because the URI requires the identifier, we need to ensure the identifier is passed. The way we've done this in the controllers is to pass the identifier as a parameter: `$this->url('blog/edit/', ['id' => $id])`. This would require that we pass the original `Post` instance or the identifier we pull from it to the view, however. zend-router allows another option, however: you can tell it to re-use currently matched parameters. This is done by setting the last parameter of the view-helper to `true`: `$this->url('blog/edit', [], true)`.

If you try and update the post, it'll be successful, but you'll notice that no edits were saved! Why? Because we have not yet implemented the functionality in our command class. Let's do that now.

Edit the file `module/Blog/src/Model/ZendDbSqlCommand.php`, and update the `updatePost()` method to read as follows:

```
 public function updatePost(Post $post)
{
```

```
    if (! $post->getId()) {
        throw RuntimeException('Cannot update post; missing
identifier');
    }

    $update = new Update('posts');
    $update->set([
            'title' => $post->getTitle(),
            'text' => $post->getText(),
    ]);
    $update->where(['id = ?' => $post->getId()]);

    $sql = new Sql($this->db);
    $statement = $sql->prepareStatementForSqlObject($update);
    $result = $statement->execute();

    if (! $result instanceof ResultInterface) {
        throw new RuntimeException(
            'Database error occurred during blog post update operation'
        );
    }

    return $post;
}
```

This looks very similar to the `insertPost()` implementation we did earlier. The primary difference is the usage of the `Update` class; instead of calling a `values()` method on it, we call:

- `set()`, to provide the values we are updating.
- `where()`, to provide criteria to determine which records (record singular, in our case) are updated.

Additionally, we test for the presence of an identifier before performing the operation, and, because we already have one, and the `Post` submitted to us contains all the edits we submitted to the database, we return it verbatim on success.

Implementing the delete functionality

Last but not least, it's time to delete some data. We start this process by implementing the `deletePost()` method in our `ZendDbSqlCommand` class:

```
public function deletePost(Post $post)
{
    if (! $post->getId()) {
        throw RuntimeException('Cannot update post; missing
identifier');
    }

    $delete = new Delete('posts');
    $delete->where(['id = ?' => $post->getId()]);

    $sql = new Sql($this->db);
    $statement = $sql->prepareStatementForSqlObject($delete);
    $result = $statement->execute();

    if (! $result instanceof ResultInterface) {
        return false;
    }

    return true;
}
```

The above uses `Zend\Db\Sql\Delete` to create the SQL necessary to delete the post with the given identifier, which we then execute.

Next, let's create a new controller, `Blog\Controller\DeleteController`, in a new file `module/Blog/src/Controller/DeleteController.php`, with the following contents:

```
<?php
namespace Blog\Controller;

use Blog\Model\Post;
use Blog\Model\PostCommandInterface;
use Blog\Model\PostRepositoryInterface;
use InvalidArgumentException;
use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class DeleteController extends AbstractActionController
```

```
{

    private $command;


    private $repository;


    public function __construct(
        PostCommandInterface $command,
        PostRepositoryInterface $repository
    ) {
        $this->command = $command;
        $this->repository = $repository;
    }

    public function deleteAction()
    {
        $id = $this->params()->fromRoute('id');
        if (! $id) {
            return $this->redirect()->toRoute('blog');
        }

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

        $request = $this->getRequest();
        if (! $request->isPost()) {
            return new ViewModel(['post' => $post]);
        }

        if ($id != $request->getPost('id')
            || 'Delete' !== $request->getPost('confirm', 'no')
        ) {
            return $this->redirect()->toRoute('blog');
        }

        $post = $this->command->deletePost($post);
        return $this->redirect()->toRoute('blog');
```

```
    }
}
```

Like the `WriteController`, it composes both our `PostRepositoryInterface` and `PostCommandInterface`. The former is used to ensure we are referencing a valid post instance, and the latter to perform the actual deletion.

When a user requests the page via the `GET` method, we will display a page containing details of the post, and a confirmation form. When submitted, we'll check to make sure they confirmed the deletion before issuing our delete command. If any conditions fail, or on a successful deletion, we redirect to our blog listing page.

Like the other controllers, we now need a factory. Create the file `module/Blog/src/Factory/DeleteControllerFactory.php` with the following contents:

```
<?php
namespace Blog\Factory;

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

class DeleteControllerFactory implements FactoryInterface
{

    public function __invoke(ContainerInterface $container,
$requestedName, array $options = null)
    {
        return new DeleteController(
            $container->get(PostCommandInterface::class),
            $container->get(PostRepositoryInterface::class)
        );
    }
}
```

We'll now wire this into the application, mapping the controller to its factory, and providing a new route. Open the file `module/Blog/config/module.config.php` and make the following edits.

First, map the controller to its factory:

```
 'controllers' => [
    'factories' => [
        Controller\ListController::class =>
Factory\ListControllerFactory::class,
        Controller\WriteController::class =>
Factory\WriteControllerFactory::class,

        Controller\DeleteController::class =>
Factory\DeleteControllerFactory::class,
    ],
],
```

Now add another child route to our "blog" route:

```
 'router' => [
    'routes' => [
        'blog' => [


            'child_routes' => [


                'delete' => [
                    'type' => Segment::class,
                    'options' => [
                        'route' => '/delete/:id',
                        'defaults' => [
                            'controller' =>
Controller\DeleteController::class,
                            'action'     => 'delete',
                        ],
                        'constraints' => [
                            'id' => '[1-9]\d*',
                        ],
                    ],
                ],
            ],
        ],
    ],
],
```

Finally, we'll create a new view script, `module/Blog/view/blog/delete/delete.phtml`, with the following contents:

```
 <h1>Delete post</h1>

<p>Are you sure you want to delete the following post?</p>

<ul class="list-group">
    <li class="list-group-item"><?=
$this->escapeHtml($this->post->getTitle()) ?></li>
</ul>

<form action="<?php $this->url('blog/delete', [], true) ?>"
method="post">
    <input type="hidden" name="id" value="<?=
$this->escapeHtmlAttr($this->post->getId()) ?>" />
    <input class="btn btn-default" type="submit" name="confirm"
value="Cancel" />
    <input class="btn btn-danger" type="submit" name="confirm"
value="Delete" />
</form>
```

This time around, we're not using zend-form; as it consists of just a hidden element and cancel/confirm buttons, there's no need to provide an OOP model for it.

From here, you can now visit one of the existing blog posts, e.g., `http://localhost:8080/blog/delete/1` to see the form. If you choose `Cancel`, you should be taken back to the list; if you choose `Delete`, it should delete the post and then take you back to the list, and you should see the post is no longer present.

Making the list more useful

Our blog post list currently lists everything about all of our blog posts; additionally, it doesn't link to them, which means we have to manually update the URL in our browser in order to test functionality. Let's update the list view to be more useful; we'll:

- List just the title of each blog post;
- linking the title to the post display;

- and providing links for editing and deleting the post.

- Add a button to allow users to add a new post.

In a real-world application, we'd probably use some sort of access controls to determine if the edit and delete links will be displayed; we'll leave that for another tutorial, however.

Open your `module/Blog/view/blog/list/index.phtml` file, and update it to read as follows:

```
 <h1>Blog Posts</h1>

<div class="list-group">
<?php foreach ($this->posts as $post): ?>
  <div class="list-group-item">
    <h4 class="list-group-item-heading">
      <a href="<?= $this->url('blog/detail', ['id' => $post->getId()])
?>">
        <?= $post->getTitle() ?>
      </a>
    </h4>

    <div class="btn-group" role="group" aria-label="Post actions">
      <a class="btn btn-xs btn-default" href="<?=
$this->url('blog/edit', ['id' => $post->getId()]) ?>">Edit</a>
      <a class="btn btn-xs btn-danger" href="<?=
$this->url('blog/delete', ['id' => $post->getId()]) ?>">Delete</a>
    </div>
  </div>
<?php endforeach ?>
</div>

<div class="btn-group" role="group" aria-label="Post actions">
  <a class="btn btn-primary" href="<?= $this->url('blog/add') ?>">Write
new post</a>
</div>
```

At this point, we have a far more functional blog, as we can move around between pages using links and buttons.

Summary

In this chapter we've learned how data binding within the zend-form component works, and used it to provide functionality for our update routine. We also learned how this allows us to de-couple our controllers from the details of how a form is structured, helping us keep implementation details out of our controller.

We also demonstrated the use of view partials, which allow us to split out duplication in our views and re-use them. In particular, we did this with our form, to prevent needlessly duplicating the form markup.

Finally, we looked at two more aspects of the `Zend\Db\Sql` subcomponent, and learned how to `Update` and `Delete` operations.

In the next chapter we'll summarize everything we've done. We'll talk about the design patterns we've used, and we'll cover several questions that likely arose during the course of this tutorial.
