Chapter No. 9
"Let's Read Feeds – A News Reader"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.9 "Let's Read Feeds – A News Reader"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Christopher Valles is a Software Engineer from Barcelona, Spain, currently based in London, UK. He started developing when he was seven using a Vtech kid laptop that was strangely shipped with a simple version of the BASIC programming language. Since then, he has explored more than 16 different programming languages ranging from Assembler to PHP, Python, and GO.

Chris also stepped into the sysadmin role and has been managing systems since he started working in this industry. He has taken care of servers right from simple webservers to infrastructures on the Cloud and internal Mac infrastructures. He is an Apple Certified Support Professional and Apple Certified Technical Coordinator.

His desire to learn and experiment has driven him to explore other fields, such as machine learning and robotics. He currently owns close to five robots and has built more than 20 over the past years. If you don't find him on the computer, he is probably spending time in the kitchen cooking delicious recipes.

The sectors where Chris has worked ranges from adult content websites and payment processors to social networks and the gaming industry. Presently, he's working as a Software Engineer at Hailo Networks, Ltd.

For More Information:
Zend Framework 2
Application Development

Nowadays, the programming scene is full of frameworks, and they move really fast, providing more and more functionalities. When we have to choose a framework, every developer has different requirements. But usually in the business environment we want to choose something that assures somehow a continuity on the framework, stability, and a good community behind.

Zend Framework is well known for being a really stable framework chosen by a lot of companies from really big to small ones. They have a huge community behind and a lot of people contribute in a way or another.

The first version of ZF was released in 2007, and since then they continue updating the project. Right now the code is old and tied to solutions that made sense in the past; the technology has advanced and new techniques and approaches have appeared. In this situation it makes perfect sense to try to incorporate them to the framework, but integrating these with the old code is just a nightmare because too many things have to be changed. In order to get the most from the new techniques, the solution was to do a new framework from scratch and implement the new concepts from the core and build everything around it.

That's what happened in September 2012, the first release of ZF2 was launched after a lot of work, discussions, and lines of code, and now we have it available to build the next generation of online services.

This new version was created from scratch using the best approaches for every problem and is really fresh and flexible. All the components have been revised and a lot of them rewritten, they also solved well-known problems from ZF1.

In this book we will review all the changes made in ZF2. The book is written with a hands-on approach, so you will learn the concepts while programming and building something that is actually usable. We will create a social network from scratch using 90 percent of all the components available in ZF2. So, we will cover a huge part of it, and definitely all you will need in the future is to write your applications.

What This Book Covers

Chapter 1, Let's Build a Social Network, explains how we can create or build a social network. It also covers how we can architect a social network.

Chapter 2, Setting Up the Environment, is important because if you do not understand it you will not be able to do anything! Let's see how to use Vagrant and virtual machines.

For More Information:
Chapter 3, Scratching the Surface of Zend Framework 2, is essential to get the knowledge of the basic components of the framework and also to help understand concepts introduced later on.

Chapter 4, The First Request/Response – Building the User Wall, is essential in a social network. It is where the users can see the content, and where we can see how the API-centric approach works.

Chapter 5, Handling Text Content – Posting Text, is the simplest action a user can do. We will learn how to create forms and validate them.

Chapter 6, Working with Images – Publishing Pictures, is something people love to do on social networks. We will learn how to handle images and file uploads.

Chapter 7, Dealing with URLs – Posting Links, is essential to share things we found on the Internet. This is the perfect excuse to see how to scrap contents of other websites and create custom validators.

Chapter 8, Dealing with Spam – Akismet to the Rescue, explains how to protect ourselves from the spammers in the Internet.

Chapter 9, Let's Read Feeds – A News Reader, is something we use almost every day. Why don't we integrate it and learn how to deal with feeds on our social network?

Chapter 10, Sign Up, is the basic thing people have to do to start using a service that will store data tailored to them. And, as usual, we will have to send the typical confirmation e-mail. How do we accomplish that task?

Chapter 11, Log In, is something we are used to doing and our users will need to do. We will see how to be sure that our users are who they say and authenticate them.

Chapter 12, Sending E-mails, is a link everyone has used at least once. We will cover in-depth the e-mail sending mechanism and we will see how to organize the views for e-mails.

Chapter 13, OAuth 2.0 Protocol Securing our API, is essential if we don't want people to use it in the Web for which it was not designed. As we will expose the API to the world, we should learn how to protect it.

For More Information:
Let's Read Feeds – A News Reader

It's time to add a new, big functionality on our social network. Right now, users can post a variety of content and comment on each other's content. Of course, they cannot register themselves or log in to the page, but this problem will be fixed in the next few chapters. From the point of view of the product, we want users to spend more time on our social network, and that's why we are going to add a feed reader.

By the end of this chapter, you will know how to use a few components of Zend\Feed, which will be used to parse the RSS feeds of the websites to which the user wants to subscribe to. Then we will use the CLI version of the controllers to make a script that will fetch the articles of the feeds every X minutes. Of course, this will be made to run by a cronjob. At the frontend, we will see how to use Zend\Navigation in two ways: the default easy way, and the programmatic way. We will also work with the usual suspects, databases, table gateways, and so on. But we will review that really fast because the focus now is on different components.

A cronjob is a tool in Unix-like systems, which allows us to schedule the execution of any program we want.

For More Information:
Overview
On the application, the functionality will look similar to the following screenshot:

Let's review all the sections as follows:

1. We will refactor the menu we have on the header to use a simple Zend\Navigation.
2. This will be the menu that the user will use to select the feed they want to read. We will program a Zend\Navigator component to populate the pages and detect the element that is active.
3. This is a small form, which will be used to add new subscriptions.
4. This button will allow the users to remove the subscription to a feed and also delete the data we store on the database and the related articles.
5. This is the main section of the functionality, and this is where we are going to show a list of the articles to be read.
API development

We need to do two things on the API side of the project. The first one is related to the API we expose; we need to add methods to insert, retrieve, and delete subscriptions. Another big task we have to do is related to the articles we need to fetch from the RSS feeds. We will create a CLI script to fetch and store them on the database. As the first big task is similar to what we did in the previous chapters, we will revise it quickly and focus on the CLI script.

Requirements

The requirement on the API side will be adding a new endpoint to manage the feed subscriptions of a user. Of course, we will need a couple of tables and a few configurations on the database to store the information. For the CLI script, we need to add a new CLI route on the configuration, and then create the script that will fetch the data and store it on the database, outputting the information on the console as a feedback to the user.

The new endpoint will be /api/feeds/:username[/:id]. Now, let's see how to use each HTTP method on this new endpoint:

<table>
<thead>
<tr>
<th>HTTP method</th>
<th>Controller method</th>
<th>Parameters</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>get()</td>
<td>None</td>
<td>The get() method will return an HTTP error 405, because we will not retrieve information of a subscription directly. The getList() method will return a list of all the feeds to which a user is subscribed, and also the related list of posts nested inside each feed information.</td>
</tr>
<tr>
<td></td>
<td>getList()</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>POST</td>
<td>create()</td>
<td>data</td>
<td>This is the method used to add a new subscription for a user. This method will add a new entry on the table of feeds, but will not retrieve the articles of the feed at creation time.</td>
</tr>
<tr>
<td>PUT</td>
<td>update()</td>
<td>ID</td>
<td>This method is not allowed</td>
</tr>
<tr>
<td>HTTP method</td>
<td>Controller method</td>
<td>Parameters</td>
<td>Functionality</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>DELETE</td>
<td>delete()</td>
<td>ID</td>
<td>This method will be used to remove a subscription. Because an ID is mandatory, we will need to pass the id of the feed we want to remove and this will also trigger a removal of the related articles.</td>
</tr>
</tbody>
</table>

**Working with the database**

To store the information of the feeds and articles on the database, we need to use two tables. The first one called `user_feeds` will store the subscriptions of a specific user and the general information of that feed. The second table called `user_feed_articles` will store the articles of specific feeds and the information related to each article. The following is the structure of each table:

```sql
CREATE TABLE `user_feeds` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `user_id` int(11) unsigned DEFAULT NULL,  
  `url` varchar(2048) DEFAULT NULL,  
  `title` varchar(512) DEFAULT NULL,  
  `icon` varchar(2048) DEFAULT NULL,  
  `created_at` timestamp NULL DEFAULT NULL,  
  `updated_at` timestamp NULL DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `idx_user_id` (`user_id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `user_feed_articles` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `feed_id` int(11) unsigned DEFAULT NULL,  
  `title` varchar(512) DEFAULT NULL,  
  `content` text,  
  `url` varchar(2048) DEFAULT NULL,  
  `author` varchar(255) DEFAULT NULL,  
  `created_at` timestamp NULL DEFAULT NULL,  
  `updated_at` timestamp NULL DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `idx_feed_id` (`feed_id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

As you can see, the data stored in the tables is self-explanatory and straightforward.
Expanding the module structure

Ok, now it's time to take a look at the structure of the folders. As usual, we will create a new module called Feeds for this functionality. Remember to add it to the `application.config.php` file. The folder structure is as follows:

```
module
  Feeds
    config
      module.config.php
    src
      Feeds
        Controller
          CliController.php
          IndexController.php
        Model
          UserFeedArticlesTable.php
          UserFeedsTable.php
      Module.php
```

The module.config.php file

This file will contain the route that we will expose on the API and also a new type of route called console route. This new type of route will allow us to map the parameters used to call the application from the command line to actual controllers and actions that will take care of executing the required code. As this controls the parameters of the command-line call, we can also specify which one we will use, if any of them are optional, and so on.

```
'router' => array(
    'routes' => array(
        'news' => array(
            'type' => 'Zend\Mvc\Router\Http\Segment',
            'options' => array(
                'route' => '/api/feeds/:username[/:id]',
                'constraints' => array(
                    'id' => '\d+'
                ),
                'defaults' => array(
                    ...
                )
            ),
        )
    )
)
```

For More Information:

Let's Read Feeds - A News Reader

'controller' => 'Feeds\Controller\Index',
},
},
},
),
),
),
),
),
),
),
),
,)

This is the normal route we expose on the API level. As you can see right now, the username is mandatory, and we also have an ID that is optional. The ID will refer to the feed ID when we delete the subscriptions. In the next chapter, we will make the username parameter disappear because we will use the information of the Oauth 2.0 mechanism to identify who's making the request; but right now, we need to specify the username each time we make a request.

'console' => array(
  'router' => array(
    'routes' => array(
      'feeds-process' => array(
        'options' => array(
          'route' => 'feeds process [--verbose|-v]',
          'defaults' => array(
            'controller' => 'Feeds\Controller\Cli',
            'action'     => 'processFeeds'
          )
        )
      )
    )
  )
),
)

This is the new type of route we were talking about earlier. As you can see, the structure is fairly similar to the default routes. The only difference is that the route parameter specifies the parameters you have to pass to the index.php file when called from the command line in order to execute the specific controller and the configured action. As you can also see, there is a parameter between the square brackets, which means that the parameter is optional. This is exactly the same as with the normal routes, but in this case, we also have a pipe symbol and a shorter version of the same parameter. This allows us to give longer and shorter versions of each parameter. Now, you go to the public folder using the command line and execute the following code:

php index.php feeds process -v

For More Information:  
The request will be sent to the **CliController.php** file and will be fulfilled. That's the line with which we will configure the cronjob:

```php
'controllers' => array(
    'invokables' => array(
        'Feeds\Controller\Index' => 'Feeds\Controller\IndexController',
        'Feeds\Controller\Cli' => 'Feeds\Controller\CliController',
    ),
),
```

This is the last block of code of this file, and we are only listing the available controllers on this module and mapping them to the actual file.

**The Module.php file**

In this case, we do not need to add or modify any of the default code that usually comes with this file. If you review the code on the file, you will see that we have two methods: `getConfig()` and `getAutoloaderConfig()`, and they are the same as the ones we saw in the previous `Module.php` files of other modules.

**Adding the UserFeedsTable.php file**

At this point, we already saw a few table gateways, and I'm confident that you are totally capable of creating them on your own. In this case, we will just highlight the `updateTimestamp()` method that takes care of updating the updated_at column with the current timestamp. This column will be used by the CLI script to track the last time we fetched the articles and avoid duplicating them on the database. As usual, we have other methods: `create()` that adds a new row on the table, `getByUserId()` that fetches all the rows based on the user_id attribute, and the `setDbAdapter()` method, which we have on all the table gateways.

```php
public function updateTimestamp($feedId)
{
    return $this->update(array(
        'updated_at' => new Expression('NOW()')
    ), array(
        'id' => $feedId
    ));
}
```

For More Information:

As you can see, we issue a call to the `update()` method passing an array with the changes we want to make; in this case, by updating the `updated_at` column with the `NOW()` expression. The second parameter we pass to the method is the `WHERE` clause, because we don't want to mess up and update all the rows.

### Adding the UserFeedArticleTable.php file

We are not going to review this table gateway in detail because it is really simple, and as we said before, you should be capable of doing it yourself. In this case, we have a couple of methods: the usual `create()` method to insert the data on the table and the `getByFeedId()` method to retrieve all the articles of a specific feed. As usual, the `setDbAdapter()` method is also present in this class.

### The contents of the IndexController.php file

In this controller, we will take care of retrieving the information of an RSS to insert it on the database, and this would imply the usage of `Zend\Feed`.

As usual, at the beginning of the class, we add the namespace and the dependencies we need on this controller. In this case, we also add the following line to declare that we want to use the `Zend\Feed\Reader\Reader` component of ZF2:

```php
use Zend\Feed\Reader\Reader;
```

As we make a query to retrieve the URL of the RSS feed and the fav icon of the page to use as an icon on our menu, we also need a `Zend\Http\Client`:

```php
use Zend\Http\Client;
```

Additionally, we also use the following components:

```php
use Zend\Dom\Query;
use Zend\Validator\Db\NoRecordExists;
```

As is common on the other controllers, we declare a few properties at the top of the class to hold a copy of the table gateways and avoid creating a few of them using the following code:

```php
protected $userFeedsTable;
protected $userFeedArticlesTable;
protected $usersTable;
```
The methods that are not implemented on these controllers are the same as the ones we saw in the previous table: `get()` and `update()`. They will contain the following line to return a 405 HTTP code to the clients:

```php
$this->methodNotAllowed();
```

Let’s now review each method one by one to see what they do and how they do it.

The first one we will review is `getList()`, which basically returns a list of all the feeds with the associated articles nested on the information. In the first few lines of the code, you can see that we extract the information of the user based on the mandatory `username` parameter on the route. As we mentioned before, this will be amended in the future when we implement the API authentication. But for now, the following quick and dirty solution will do the job:

```php
$username = $this->params()->fromRoute('username');
$usersTable = $this->getTable('UsersTable');
$user = $usersTable->getByUsername($username);
$userFeedsTable = $this->getTable('UserFeedsTable');
$userFeedArticlesTable = $this->getTable('UserFeedArticlesTable');

$feedsFromDb = $userFeedsTable->getByUserId($user->id);
$feeds = array();
foreach ($feedsFromDb as $f) {
    $feeds[$f->id] = $f;
    $feeds[$f->id]['articles'] = $userFeedArticlesTable->getByFeedId($f->id)->toArray();
}

return new JsonModel($feeds);
```

As you can see, we retrieved the feeds from the table based on the user ID and then we proceeded to extract the articles of each feed from the database. If you take a closer look, you can see that we are creating an associative array using the ID of the feed as a key. This is a convenient way to return the information for the frontend and be able to quickly extract the articles we need to show, based on the feed ID. Of course, if a feed doesn't have any articles, we will store an empty array on the `articles` key.

In the following section, we are going to review the `delete()` method, which is very simple:

```php
$username = $this->params()->fromRoute('username');
$usersTable = $this->getTable('UsersTable');
$user = $usersTable->getByUsername($username);
```

For More Information:

$userFeedsTable = $this->getTable('UserFeedsTable');
$userFeedArticlesTable = $this->getTable('UserFeedArticlesTable');

$userFeedArticlesTable->delete(array('feed_id' => $id));
return new JsonModel(array(
    'result' => $userFeedsTable->delete(array(
        'id' => $id,
        'user_id' => $user->id
    ))
));

The procedure is essentially the same, the only difference with the method we reviewed before is that we issue a `delete()` call on the table gateway, passing the `feed_id` attribute. The first call removes the articles from `user_feed_articles`, and the second `delete()` call removes the subscription from `user_feeds`. As you can see, we also used the `user_id` attribute on the second `delete()` call. This is a small protection to avoid someone deleting a subscription of another user. Of course, if we were doing this in a professional way, we would want to first check if the user has the subscription we are trying to delete before we actually delete the data.

Now let's jump to the last method we are going to explain in this controller. The `create()` method takes care of extracting the information given by the user from the website and storing the subscription. We can just accept an RSS URL and store it. But in this case, we also need to extract the favicon of the original website to use it at the frontend. So, the user needs to provide the URL of the website instead of the URL of the RSS; we will take care of discovering the RSS URL by using the following code:

$username = $this->params()->fromRoute('username');
$usersTable = $this->getTable('UsersTable');
$user = $usersTable->getByUsername($username);

This is the first block and as we saw before, it just retrieves the user from the database to have access to the user data.

$userFeedsTable = $this->getTable('UserFeedsTable');
$rssLinkXpath = '//link[@type="application/rss+xml"]';
$faviconXpath = '//link[@rel="shortcut icon"]';

In the second block, of the method, we retrieve an instance of the table gateway, and then store two XPath expressions that will help us to retrieve the HTML tags that contain the RSS URL and the favicon URL.

$client = new Client($data['url']);
$client->setEncType(Client::ENC_URLENCODED);
$client->setMethod(Zend\Http\Request::METHOD_GET);
$response = $client->send();

For More Information:
The third section of code prepares the HTTP client that we will use to retrieve the HTML and issues the request.

Now, we arrive on a conditional block that checks if we got the HTML from the website or not. In case the request fails, we throw an exception.

```php
if ($response->isSuccess()) {
    ...
} else {
    throw new Exception("Website not found", 404);
}
```

XPath is a query language that allows us to select nodes inside an XML document.

Keep in mind that here we are taking a shortcut with an exception. In a proper application, you need to add this to a queue and retry in a few minutes or just retry a few times before throwing the exception. But this is out of the scope of this book.

```php
$html = $response->getBody();
$html = mb_convert_encoding($html, 'HTML-ENTITIES', "UTF-8");

$dom = new Query($html);
$rssUrl = $dom->execute($rssLinkXpath);

if (!count($rssUrl)) {
    return new JsonModel(array(
        'result' => false,
        'message' => 'Rss link not found in the url provided'
    ));
}
$rssUrl = $rssUrl->current()->getAttribute('href');

$/faviconUrl = $dom->execute($faviconXpath);
if (count($faviconUrl)) {
    $faviconUrl = $faviconUrl->current()->getAttribute('href');
} else {
    $faviconUrl = null;
}
```

This is the code you will find inside the if block that we saw before, which of course, is only executed if we are able to get the contents of the page.
The first two lines just retrieve the contents and take care of converting the encoding, ensuring that there are no issues when using the HTML inside the Zend\Dom\Query object that comes later.

The first thing we try is to retrieve the HTML tag that contains the RSS link. In case we don't find it, we just throw an exception. After that, we store the href value on a variable and carry on trying to get the favicon URL. In this case, the favicon is not a critical part. So if we don't get it, we just ignore it.

The final part of this method is the following block. Here, we will load the RSS URL using a Zend\Feed\Reader\Reader component, and use that component to extract the information from the RSS-like title and so on:

```php
$rss = Reader::import($rssUrl);

return new JsonModel(array(
    'result' => $userFeedsTable->create(
        $user->id,
        $rssUrl,
        $rss->getTitle(),
        $faviconUrl
    )
));
```

As a final note on this controller we also have the methodNotAllowed() and getTable() methods that are similar to the ones used on the other controllers. As you see, we use exactly the same code over and over again for the methodNotAllowed() method which is a good candidate to be promoted as a parent class and get rid of it on all its children.

## Creating the CliController.php file

This is the new type of controller we are going to see in this book. Actually, it is not a new type, it is just a controller extending the AbstractActionController class, which will be called using the command-line Interface. This means that all the controllers extending the AbstractActionController class can be called through the CLI, but it is your job to detect if the request is coming from the normal channels as a HTTP request or a CLI request, and then act in one way or the other on each case. You can check the type of the request object to determine if it is a CLI request or a normal request. In the first case, the object will be an instance of Zend\Console\Request, and in the latter, it will be an instance of Zend\Http\PhpEnvironment\Request.
This script will take care of retrieving the new articles found on each RSS feed on the database. To accomplish that, we will use the updated_at field to avoid duplications on the articles. Also, keep in mind that this script is supposed to be called by a cronjob every X minute to refresh the articles in an automatic way.

As we are going to read the RSS feed from this controller, we need to declare the usage of the Feed components of ZF2 as follows:

```php
use Zend\Feed\Reader\Reader;
```

We need to work with the new tables and this means that we need to declare the properties that will hold the table and the `getTable()` method, which will create the instances.

```php
protected $userFeedsTable;
protected $userFeedArticlesTable;
```

After all of this, we arrive on the `processFeedsAction()` method we called on the console route, which is the one in charge of processing the requests. Let's see what we have inside it.

```php
$request = $this->getRequest();
$verbose = $request->getParam('verbose')
|| $request->getParam('v');
```

These are the first two lines. The objective is to check if the `verbose` parameter was specified while calling the script from the command line. If you remember, we specified two versions of the `verbose` parameter, the long and the short one, and that's why we need to retrieve both here.

```php
$userFeedsTable = $this->getTable('UserFeedsTable');
$userFeedArticlesTable = $this->getTable('UserFeedArticlesTable');
$feeds = $userFeedsTable->select();
```

The following block of code gets instances of the tables we need to use, and utilizes the first one to fetch all the feeds on the database and loop over them to execute the following code:

```php
foreach ($feeds as $feed) {
    if ($verbose) {
        printf("Processing feed: %s\n", $feed['url']);
    }
```
$lastUpdate = strtotime($feed['updated_at']);
$rss = Reader::import($feed['url']);

// Loop over each channel item/entry and store relevant data
for each
foreach ($rss as $item) {
    $timestamp = $item->getDateCreated()->getTimestamp();
    if ($timestamp > $lastUpdate) {
        if ($verbose) {
            printf("Processing item: %s\n", $item
                ->getTitle());
        }
        $author = $item->getAuthor();
        if (is_array($author)) {
            $author = $author['name'];
        }

        $userFeedArticlesTable->create(
            $feed['id'],
            $item->getTitle(),
            $item->getContent(),
            $item->getLink(),
            $author
        );
    }
}

if ($verbose) {
    printf("Updating timestamp\n");
}

$userFeedsTable->updateTimestamp($feed['id']);

if ($verbose) {
    printf("Finished feed processing\n\n");
}
The first part just outputs the information to the console in case the `verbose` parameter is specified. Then, we get the last update from the database and convert it to a timestamp to be able to compare it. Right after that, we import the RSS feed using the `Zend\Feed\Reader\Reader` component and loop over it to access all the articles.

For each article on the feed, we extract the timestamp and compare it against the value of the last update we have on the database. In case the article was published after the last update, we proceed printing more debug information, and then insert the data on the table, using the information we get from the item object.

After looping through all the articles, we print more debug information and then update the timestamp for this feed on the database, finishing with more debug strings.

If something goes wrong while processing, the script will stop. So, if you are going to use this on a production-ready application, you need to take care of all the exceptions that can occur.

**The ApiErrorListener.php file**

Because now the controllers can be called by the normal channel or CLI, we need to handle the errors in a different way for each case. If it is an error produced while fulfilling an HTTP request, we need to return it using the method we have in place; however, if an error occurs while running the CLI script, we need to output it on the console. To fix this, we have to do a small change in the `ApiErrorListener.php` file, and replace the first line of the `onRender()` method. Instead of just checking if the response is OK, we are also going to test if the request is a CLI request.

```php
if ($e->getRequest() instanceof \Zend\Console\Request || $e->getResponse()->isOk())
```

For More Information:
Frontend
At the frontend, we are going to create a new module called Feeds, modify the API module, and also the layout.phtml file inside the Common module. The folder structure is as follows:

```
  module
    Feeds
      config
        module.config.php
      src
        Feeds
          Controller
            IndexController.php
          Entity
            Article.php
            Feed.php
          Forms
            SubscribeForm.php
            UnsubscribeForm.php
      view
        feeds
          index
            index.phtml
          forms
            subscribe-form.phtml
            unsubscribe-form.phtml
        partials
          menu-feed-container.phtml
            Module.php
```

The API module
As we did before, we need to extend the ApiClient.php class and add the required methods to interact with the new endpoint we just created on the API side of the app.

The following three changes will allow us to interface the functionality provided at the frontend with the corresponding actions in the API.
Modifying the ApiClient.php file

The file will be modified to add three new methods: getFeeds(), addFeedSubscription(), and removeFeedSubscription(). The definition of the getFeeds() method is as follows:

```php
public static function getFeeds($username)
{
    $url = self::$endpointHost . sprintf(self::$endpointFeeds, $username);
    return self::doRequest($url);
}
```

The first method will just issue a GET request at the endpoint using the $username parameter on the URL. This will return the list of feeds of that user. The definition of the addFeedSubscription() method is as follows:

```php
public static function addFeedSubscription($username, $postData)
{
    $url = self::$endpointHost . sprintf(self::$endpointFeeds, $username);
    return self::doRequest($url, $postData, Request::METHOD_POST);
}
```

This is the second method. The only difference with the first one is that now we send an array of data to the API containing the URL of the website to which we want to subscribe and force a specific type of request; in this case, it's POST. The definition of the removeFeedSubscription() method is as follows:

```php
public static function removeFeedSubscription($username, $feedId)
{
    $url = self::$endpointHost . sprintf(self::$endpointSpecificFeed, $username, $feedId);
    return self::doRequest($url, null, Request::METHOD_DELETE);
}
```
And this is the last method we need to add. Now the URL used to issue the request will contain the username and also the ID of the feed we want to remove. On the request side, we force the DELETE method because (if you remember) we are dealing with the RESTful web service.

The Feeds module

The Feeds module is the new module we are going to add. Of course, this module will contain all the required data to handle a new feed reader. As usual, when we create a new module, we have to inform ZF2 about its existence. In order to accomplish this, we only have to add the name of the module on the application.config.php file in an appropriate configuration section.

The contents of module.config.php

Let's have a look at the contents of the module.config.php file. As you can imagine, we are going to expose three new routes: the first, to list the contents; the second, to handle the subscription request; the third, to handle the unsubscribed request. The routes look as follows:

```php
'router' => array(
    'routes' => array(
        'feeds' => array(
            'type' => 'Zend\Mvc\Router\Http\Segment',
            'options' => array(
                'route' => '/:username/feeds[:feed_id]
[/page/:page]',
                'constraints' => array(
                    'username' => '\w+',
                    'feed_id' => '\d+',
                ),
                'defaults' => array(
                    'controller' => 'Feeds\Controller\Index',
                    'action' => 'index'
                ),
            ),
        ),
        'feeds-subscribe' => array(
            'type' => 'Zend\Mvc\Router\Http\Segment',
            'options' => array(
                'route' => '/:username/feeds/subscribe',
                'constraints' => array(
            )
        ),
    ),
),
```

For More Information:
'username' => '\w+',
},
'defaults' => array(
  'controller' => 'Feeds\Controller\Index',
  'action' => 'subscribe'
),
},
},
'feeds-unsubscribe' => array(
  'type' => 'Zend\Mvc\Router\Http\Segment',
  'options' => array(
    'route' => ':username/feeds/unsubscribe',
    'constraints' => array(
      'username' => '\w+',
    ),
    'defaults' => array(
      'controller' => 'Feeds\Controller\Index',
      'action' => 'unsubscribe'
    ),
  ),
},
},
},
},
},

Right after that, we need to specify the available controllers and the view manager configuration, which is as follows:

'controllers' => array(
  'invokables' => array(
    'Feeds\Controller\Index' =>
      'Feeds\Controller\IndexController'
  ),
},
),
'view_manager' => array(
  'template_path_stack' => array(
    __DIR__ . '/../view',
  ),
},
)
The IndexController.php file

The controller will also contain three methods, one for each route we declared. The components we need to use are as follows:

```php
use Zend\Mvc\Controller\AbstractActionController;
use Zend\Stdlib\Hydrator\ClassMethods;
use Zend\Navigation\Navigation;
use Zend\Navigation\Page\AbstractPage;
use Zend\Paginator\Paginator;
use Zend\Paginator\Adapter\ArrayAdapter;
use Api\Client\ApiClient;
use Users\Entity\User;
use Feeds\Entity\Feed;
use Feeds\Forms\SubscribeForm;
use Feeds\Forms\UnsubscribeForm;
```

As you can see, we have a mix of the components we created and the components from ZF2. If you notice carefully, you will realize that the last two are related to the Navigation object. We are going to learn how to add pages to it programmatically and also how to use the default behavior.

Now, let's dissect the `indexAction()` method that will show the feeds with the feed menu and the articles as follows:

```php
$viewData = array();

$flashMessenger = $this->flashMessenger();

$username = $this->params()->fromRoute('username');
$this->layout()->username = $username;
$currentFeedId = $this->params()->fromRoute('feed_id');

$response = ApiClient::getWall($username);
if ($response !== false) {
    $hydrator = new ClassMethods();
    $user = $hydrator->hydrate($response, new User());
} else {
    $this->getResponse()->setStatusCode(404);
    return;
}
```

For More Information:
$subscribeForm = new SubscribeForm();
$unsubscribeForm = new UnsubscribeForm();
$subscribeForm->setAttribute(
    'action',
    $this->url()->fromRoute(
        'feeds-subscribe', array('username' => $username)
    )
);
$unsubscribeForm->setAttribute(
    'action',
    $this->url()->fromRoute(
        'feeds-unsubscribe', array('username' => $username)
    )
);

These are the first lines. We will get some data from the route, such as the username and the current feed ID if we are looking at some articles. After that, we get the information of the user, create an instance of the subscribed and unsubscribed form, and finally, configure them, setting the actions that will handle their requests.

$hydrator = new ClassMethods();
$response = ApiClient::getFeeds($username);
$feeds = array();
foreach ($response as $r) {
    $feeds[$r['id']] = $hydrator->hydrate($r, new Feed());
}
if ($currentFeedId === null && !empty($feeds)) {
    $currentFeedId = reset($feeds)->getId();
}

The following block of code takes care of getting the feeds of the user from the API and hydrating the Feed entities. After that, we check if we are already reading articles from a feed. If not, we assign the first feed as default, in case the user has several subscriptions.

$feedsMenu = new Navigation();
$router = $this->getEvent()->getRouter();
$routeMatch = $this->getEvent()->getRouteMatch()->setParam(
    'feed_id',
    $currentFeedId
);
foreach ($feeds as $f) {
Let's Read Feeds – A News Reader

```php
$feedsMenu->addPage(
    AbstractPage::factory(array(
        'title' => $f->getTitle(),
        'icon' => $f->getIcon(),
        'route' => 'feeds',
        'routeMatch' => $routeMatch,
        'router' => $router,
        'params' => array(
            'username' => $username,
            'feed_id' => $f->getId()
        )
    )
);}
```

This is the block of code that will create the feed menu. We have used the Navigation component from ZF2 and added pages to it based on the feeds a user has. The AbstractPage factory will create pages of the MVC type, which means that they are tied to routes or pairs of controllers/actions. This also gives us a benefit; the component will detect the element of the menu that is actively looking at the URL of the request by itself. As we have created the pages manually, we need to help the component a little by specifying the router where all the routes are stored and the RouteMatch object that will match the URL we are fulfilling. The RouteMatch object will be used to test each page against it and decide which one is active. Finally, while creating a page, we can add more parameters than the ones needed, and then retrieve them. This is what we do with the icon parameter.

```
$currentFeed = $currentFeedId != null? $feeds[$currentFeedId] : null;

if ($currentFeed != null) {
    $paginator = new Paginator(
        new ArrayAdapter($currentFeed->getArticles())
    );
    $paginator->setItemCountPerPage(5);
    $paginator->setCurrentPageNumber(
        $this->params()->fromRoute('page')
    );
    $viewData['paginator'] = $paginator;
    $viewData['feedId'] = $currentFeedId;
}
```

For More Information:
The following block of code takes care of preparing a `Zend\Paginator\Paginator` component to show the posts with a paginator. As you can see in the code, we have used the `ArrayAdapter` object, which allows us to pass an array of content to be paginated. After that, we configure the paginator and assign it to the view as follows:

```php
$unsubscribeForm->get('feed_id')->setValue($currentFeedId);
$viewData['subscribeForm'] = $subscribeForm;
$viewData['unsubscribeForm'] = $unsubscribeForm;
$viewData['feed'] = $currentFeedId;
$viewData['username'] = $username;
$viewData['feedsMenu'] = $feedsMenu;
$viewData['profileData'] = $user;

return $viewData;
```

This is the last block of code of this method. In here, we set the value of the hidden field on the unsubscribe form that will contain the id of the feed the user wants to conceal. After that, we just pass the information we need to the view.

In the following code, we are going to review the `subscribeAction()` and `unsubscribeAction()` methods. We will put them together because they are essentially the same and there is a difference in only one line. They are good candidates for being refactors and I leave you with the task of mixing them and removing their duplications.

```php
$username = $this->params()->fromRoute('username');
$request = $this->getRequest();

if ($request->isPost()) {
    $data = $request->getPost()->toArray();

    $response = ApiClient::addFeedSubscription(
        $username, array('url' => $data['url'])
    );

    if ($response['result'] == true) {
        $this->flashMessenger()->addMessage(
            'Subscribed successfully!'`
        );
    } else {
        return $this->getResponse()->setStatusCode(500);
    }
}
```

For More Information:
As you can see now, the content is really simple. After getting the request, we check if it's a POST request. After that, we get the data from the form, and use the ApiClient interface to send the appropriate request to the API. Based on the response obtained, we show an error message or we redirect the user to the list of feeds where they can see the new one.

The line that is different between them, of course, is the one that utilizes ApiClient. In the first case, we issue a call to addFeedSubscription() and in the second case, we issue a call to removeFeedSubscription(), as given in the following code:

```php
$response = ApiClient::removeFeedSubscription(
    $username,
    $data['feed_id']
);
```

There are two things that can be improved in the preceding code. The first one is the validation. I don't know if you noticed, but we are not checking if the form is valid or not. I leave that as an exercise to you because you should now be confident enough to implement that part on this code. Another thing we can improve is the error handling; for brevity purpose, we leave the error handling behind and take shortcuts for handling them. But as we say all the time, if you are creating a production-ready application, you need to take care of that.

### Entities and forms
This new functionality contains a couple of forms with their views, and a couple of entities to store the new data. We already saw in the previous chapters how to create them and the contents inside them, so we are going to skip the explanation now in order to focus on the new things we need to learn. It is very easy to see the contents. You just need to browse the code and read the contents of the files. I'm sure they will look familiar to you.

### The menu-feed-container.phtml file
Do you remember the feed menu we populated before? Now we want to customize the way it looks to add our own classes and structure. In order to accomplish that, when we render the menu, we can specify a partial that will take care of rendering the menu itself, and that's what we are going to do now using the following code:
When we deal with HTML, we usually remove some attributes to be able to fit the code in here. So for the real version, check the code. For the explanation purpose, the preceding version is more than enough.

While using a partial with a Navigation component, the container is exposed to us as a variable. This is the way we have to access the pages of the menu. In this case, we iterate over the pages to create each link. As you remember, we use a RouteMatch object to decide whether a page is active or not and we now benefit from that using the isActive() method. Based on that, we add or do not add a class to the link of the menu.

Also, we mentioned that you can add more parameters to the pages and retrieve them later. We added the icon, and now we are able to retrieve it here and use the information stored inside it to add the favicon of the page as an image on the menu.

Finally, you can always use the methods provided by default, for example, getTitle() in this case, to get the name of the menu item.

The index.phtml file
As before, the view is big and ugly and this is why we are just going to focus on one small section of the view, but that doesn't mean that you can ignore the contents of the file. So, jump to the following code and see the file by yourself:

```php
<?php echo $this->navigation($feedsMenu)->menu()->setPartial('partials/menu-feed-container.phtml') ?>
```

This is how we can use the Navigation component we created on the controller to render it using the custom partial we created. We are using the navigation helper to pass the container of the menu, and then configuring it to use the partial.
The Common module

In the Common module, we are going to make a small change that will benefit from the automated side of ZF2. We will add a new item on the header menu; but instead of doing it manually, we are going to use the default configuration of a Navigation component. In order to do that, we just need to specify the contents of the menu in a configuration file, and then use the navigation helper to render a menu with the default settings.

The global.php file

The global.php file stored in the config folder of the project is where we are going to store the contents of the menu. It is very easy, because we add the items as an array as given in the following code:

```php
return array(
    'navigation' => array(
        'default' => array(
            array(
                'label' => 'Home',
                'route' => 'wall',
            ),
            array(
                'label' => 'Feeds',
                'route' => 'feeds',
            ),
        ),
    ),
    'service_manager' => array(
        'factories' => array(
            'navigation' =>
                'Zend\Navigation\Service\DefaultNavigationFactory',
        ),
    ),
);
```

In the navigation section, we have the default menu containing only labels and routes. This will force the Navigation components to use MVC pages and as you are imagining, we will benefit from the RouteMatch attribute that the component will generate.

The second part of the config file is the service manager's configuration to point out what we mean with navigation, and point it to the right file on the framework.

For More Information:
The layout.phtml file

This is the last file we need to modify to make everything work. We are going to render the menu we just created on the config file. It's very easy because we only need to add the lines of code where we had the old menu:

```php
<?php $this->navigation('navigation')->findBy('route', 'wall')->setParams(array('username' => $this->username)); ?>
<?php $this->navigation('navigation')->findBy('route', 'feeds')->setParams(array('username' => $this->username)); ?>
<?php echo $this->navigation('navigation')->menu()->setUlClass('nav')->renderMenu(); ?>
```

The first two lines are taking care of specifying the username value that has to be used on the route while creating the links.

After that, in the last line, instead of passing a container to the navigation helper, we are passing a string that will be resolved to the DefaultNavigationFactory object, which we specified on the service manager's configuration before.

There are also two things that we need to notice here. The first one is that we can customize the class used for the ul tag that the helper puts in place while rendering the menu. This is accomplished using the setUlClass() method. The second thing is that we are using the same helper as the one used for the feeds and, if you remember, we specified a custom partial to render the menu. To avoid the usage of that partial here, we just need to force a call to the renderMenu() method, which will render it using the default ul/li structure.

Summary

We just added a new big functionality that will keep our users on the social network for more often. Congratulations!

In this chapter, we saw how to use components, for example, Zend\Feed in order to parse RSS feeds and forget about the XML structure. We explored the usage of the framework to create command-line interface scripts and still have access to all the components we created and the ones provided by ZF2. Finally, we also discovered how to create a menu using the Zend\Navigation component in the simplest way and also in the programmatic way with a customized partial.

In the next chapter, we will give the users the option of registering on the social network, but still they will not be able to log in. This will be covered right after the registration chapter.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.