Community Experience Distilled

Zend Framework 2 Application Development

Explore the Zend Framework 2 and create your own superb social network

Christopher Valles

[PACKT] open source* community experience distilled PUBLISHING

Zend Framework 2 Application Development

Explore the Zend Framework 2 and create your own superb social network

Christopher Valles

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Zend Framework 2 Application Development

First published: October 2013

Production Reference: 1211013

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-210-0

`www.packtpub.com`

Cover Image by Tim Green (`thisistimgreen@gmail.com`) via Flickr InMail

Credits

Author
Christopher Valles

Reviewers
Richard Ainger
Gabriel García Fernández
Doug Johnson
Brent Shaffer

Acquisition Editor
James Jones

Lead Technical Editor
Madhuja Chaudhari

Technical Editors
Shashank Desai
Krishnaveni Haridas
Ankita Thakur

Project Coordinator
Amey Sawant

Copy Editors
Mradula Hegde
Sayanee Mukherjee
Kirti Pai
Adithi Shetty
Laxmi Subramaniam

Proofreaders
Chrystal Ding
Clyde Jenkins

Indexer
Monica Ajmera Mehta

Graphics
Sheetal Aute

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

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.

Acknowledgments

First of all, I want to thank my lovely girlfriend for supporting me along the way of this journey. I know for sure that this would have not been possible without her, and I will be always grateful to her.

I also thank my friends, Gabriel Garcia and Tasos Bitsios, who where there answering my questions and sharing with me their thoughts about the book, chapters, and the code. Last but not least, I want to thank all the reviewers who helped me with their comments, specially Brent Shaffer for his patience and help on the implementation of OAuth 2.0 and for letting me use his library in this book.

I want to mention some people who, over the years, have contributed somehow to the person I am today and have encouraged me to follow my dreams, keep exploring new stuff, pursue new challenges, and supported me with all my crazy ideas. My parents Vicente Vallés Serrat and Manuela Ramos González, my brother Bryan Vallés Ramos, my first boss Òscar Martínez Ciuró, and Stuart Helen Overton-Smith for getting on the plane with me, teaching me how to fly, and helping me accomplish my dream of being a pilot. A special mention goes for my cats, Schrödinger and Bones, for giving me company and staying with me at 4 a.m. while I was writing.

Finally, I also want to thank all the people who made efforts and invested time researching the polyphasic sleep cycles and shared their knowledge for free on the Internet. To write this book, I switched to everyman sleep cycle that allowed me to stay awake 19 hours per day for the last six months.

About the Reviewers

Richard Ainger is a Software Engineer and all-round computer enthusiast residing in Australia.

Since becoming passionate about computers at an early age, Richard has explored a variety of programming languages including PHP, Java, C, and MIPS ASM. Richard has also branched outside of the more traditional desktop development and smartphone environments and has been an active developer of homebrew software for the PSP gaming system.

More recently, Richard has been enjoying developing software solutions for **The Cyber Institute** (**TCI**), a leader in the e-learning space within Australia. TCI is responsible for creating learning systems and content for some of the biggest names in Australia.

Every spare moment, Richard devotes extending his knowledge into new areas of software development trying to keep up with the constantly evolving field of software and web development.

I would like to thank the talented people I work with daily at TCI from whom I have learned a lot. I would also like to thank Christopher Valles and the people at Packt Publishing for giving me the opportunity to review and provide feedback during the production of this book.

Gabriel García Fernández is a Computer Engineer based in Barcelona working as Project Manager and CTO at Urban Ventures S.L. He has been working as a Web Developer for many years before becoming a tech manager.

He has an entrepreneurial mind, proactive, with a strong focus on team management and employees' motivation. He is very adaptable, self-motivated, and easily develops empathy with people and is always ready to learn. He is passionate about Agile methodologies and web development.

He spends his free time with his girlfriend, his family, going out with friends, or reading (literature is one of his passions).

Doug Johnson first discovered a knack for programming at Dartmouth College while tracking down a programming error in an orbital simulation that led to a citation in Physics. He has programmed applications in many languages, including C, C#, Delphi, PHP, and JavaScript and taught programming and scripting at the college level.

As the founder and owner of an accounting software consultancy, he developed customized solutions for business problems in the SMB market, primarily in medicine and manufacturing. Working with several companies, he has written health-care- and electronic-health-record software and served for several years on the IHE standards committee for eye care, co-writing one of the profiles.

He is currently developing tax software for Intuit, and he also develops websites and web applications using Drupal, Zend Framework, and AngularJS, primarily in the not-for-profit space.

He likes to cook and is developing a website called CookLoose that aims to teach how to go from being a non-cook to a competent cook.

I'd like to thank my wife who builds systems into our household that allows me to live comfortably while not paying any real attention to how she does it. She has designed a kitchen that anyone can work in. She doesn't understand a thing about what I do for a living but is still willing to listen to me going on and on about the latest, coolest thing I've discovered as long as I make her coffee in the morning.

Brent Shaffer is a Computer Scientist and musician living in Salt Lake City, Utah. He works for Adobe Systems, Inc. in the Web Services team for the Adobe Marketing Cloud platform. The Web Services team oversees all API Framework responsibilities for the Adobe Marketing Cloud, such as third-party integrations, authentication with OAuth 2.0, rate limiting, throttling, and traffic monitoring.

Brent also works remotely, performing with his band More Hazards More Heroes, a folk duo from Nashville, TN. His band has performed across the country their music has been featured in GoPro commercials and Hollywood films.

Brent is a longtime PHP and Symfony Framework advocate, and has worked closely with the community, writing plugins and official documentation for both the Symfony 1.x and 2.x projects.

Table of Contents

Preface

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.

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.

What you need for this book

For this book, you will need a computer, of course. In terms of software required, you don't have to be worried; all the components we use are open source, and everything is provided on the virtual machine we will set up in *Chapter 2*, *Setting Up the Environment*. The only thing you will need to do manually is install the last version of VirtualBox available for your operating system; the rest will be done almost automatically by the `Vagrant` scripts we provide.

Who this book is for

This book is for developers that want to start using Zend Framework 2 and want to build applications of an important size. We will cover a different approach on application architecture so this will also be helpful for you.

You don't need to have any experience with the framework or with Zend Framework 1, but if you have, it will be easier for you; but remember, it's not required. A good and solid knowledge of PHP 5.3 and an object-oriented paradigm is required because the whole framework is based on that and is strongly object oriented.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "On implementing this functionality, we will see how the new `Zend\Form` element handles the upload of files and how to process them."

A block of code is set as follows:

```
config.vm.box = "ubuntu-12.04"
config.vm.box_url = "http://files.vagrantup.com/precise64.box"
config.vm.network :private_network, ip: "192.168.56.2"
```

Any command-line input or output is written as follows:

```
cd /var/source-api
php composer.phar self-update
php composer.phar install
cd /var/source-client
php composer.phar self-update
php composer.phar install
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "This validator belongs to the **Email** field."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase to address it.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best

1
Let's Build a Social Network

In this chapter, we will see an overview of what we are going to build through this book, a social network. Also, we will cover the technical solution chosen to build it from scratch and thereby getting the most out of Zend Framework 2.

By the end of this chapter, you should have a good understanding of how the API-centric approach works and a clear picture of the project that we will build through this book.

Why ZF2?

Zend Framework 2, as you might know, is a framework that allows you to build applications using the components provided. The framework uses the best practices of the industry and the components are extensively tested and proven to be good. This means that you will build your application on top of a robust base.

The benefits of using a Zend framework against using your own framework is that you benefit from the knowledge of all the contributors of the project. Also, it will be easier to get more developers on board as it's a known framework with the documentation available online, and there are a lot of people who already know it or have worked with it.

What are we going to build?

The objective of this book is to show you as many components of **Zend Framework 2** (**ZF2**) as possible. In order to achieve this, we will build a basic social network that will allow the user to post text, pictures, links, comments, and so on. This will cover all the basics of the framework and also demonstrate the usage of the majority of the components in common real-life scenarios.

Let's see all the functionalities and sections this social network will offer to the users.

Building a user profile

The user wall will be the main section where users can share content. Users will be able to post text, images, and URLs.

By building this section, we will experiment with the first request and building blocks of ZF2, and in the meantime we are going to see the technical approach chosen for the project in action.

Posting text

Users will be able to post text on their wall and also on their friends' walls using a simple form. To accomplish this, we will have a look at how forms work and how to use filters and validators to ensure that the data is secure and correct.

Uploading pictures

Pictures are one of the forms of media shared extensively by users of social networks. In our case, we will give users the opportunity to do this. On implementing this functionality, we will see how the new `Zend\Form` element handles the upload of files, and how to process them.

Sharing links

URLs are the last thing users will be able to share on the profile. A description will appear automatically with the link. This functionality will allow us to discover how to crawl contents from remote websites, filter them, and store them.

Posting comments

People also enjoy commenting on the content of others people and we will give the user a way to do this on our social network. An interesting part of ZF2 will be used in this section to fight spam. We will see how to use third-party services using Akismet.

Building news reader

Building news reader is a big section on the project. From here, the user will be able to add, remove, read, and organize RSS feeds. We will use `Zend\Feed` to do basic RSS actions and provide the frontend with data.

Registering and logging in

A basic action that users should be able to do is register and login to the social network. We will provide them with the forms to do this. We are going to integrate the session handling through `Zend\Session` and the login functionality using `Zend\Authentication` on this project.

E-mails

Another functionality that social networks provide is keeping the user updated about what is happening while he/she is away. We will implement a notification system and also a way to recover the password if you lose it. On building this section of the project, we will see the `Zend\Mail` component in action and how to use it to send e-mails.

Public APIs

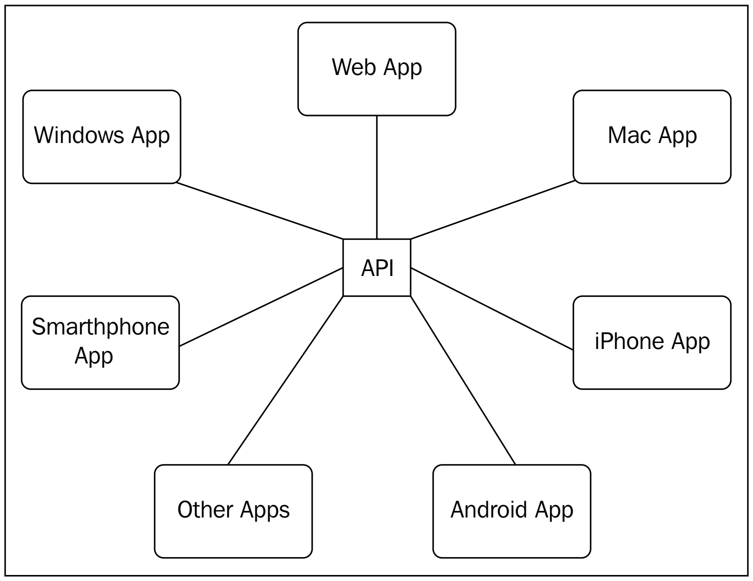
The last functionality we will provide in our project is the ability to integrate our data and functionalities in other projects. In order to accomplish this, we will expose our API to the outside world using OAuth 2.0. Here, we will learn how to put an OAuth 2.0 authentication mechanism with ZF2 in place.

The approach – API-centric app

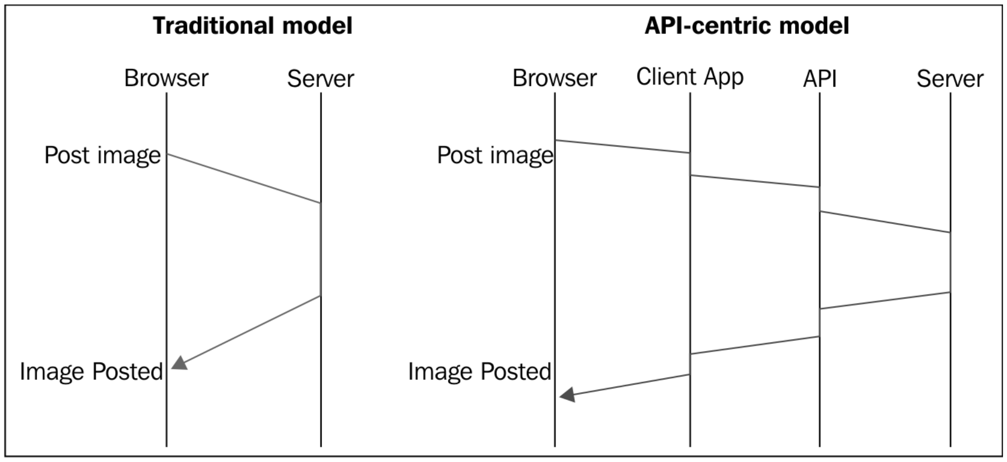
To build the social network, we will use an approach called API-centric. First of all, for those who are unfamiliar with the term API, it is an interface that we build to expose functionality. By doing this, we allow other applications to interact with us. For example, a news website can expose an API to allow people to retrieve articles from their archive by specifying the date of the article, the author, and so on.

An API-centric application is an application that executes all functionality that make calls to an internal API. For example, if a user on our social network is going to post a picture, our app will pass the image and details of the user to the API to execute the actual steps needed to store the image and publish it on the user profile.

The API-centric architecture looks like the following figure:



As you can see, API is the central point and everything else is built around it. The web app, the mac app, and so on are the clients that consume API. Now, let's compare the lifecycle of a request between an app-centric approach and a traditional model.



As you can see, in the traditional model you made requests directly to the server. The server in this case contains the logic of the client and also the business logic. In the API-centric model, the request is made from the browse to the client app; this application can be on the same server as the API or on a separate machine. Then, the client app will issue a request to the API. After this, the request will go back to the client app and finally to the user. In this case, we are separating the code of the client app from the code of the API. The client app acts as a proxy for the API that has the business logic. Note that the image doesn't represent the time spent on the request.

Since the explosive increase in the usage of smartphones, an increasing number of web applications have ended up with an application on the phone. Some of them just adapt the website to the phone screen size by removing or redesigning the interface, while others choose to build a native application that will run on the phone of the visitor.

The first benefit of this approach for our social network is that the core of the application is just an API and all the related clients will rely on it to use the functionality. This means we have a good separation of concerns, and we will have separated the business logic from the client logic. This will allow us to create a website to access the service and the possibility of building a native application for mobile phones or even a desktop program using the same API in future.

The second benefit is that the API is stateless; this means that the calls made to the API will not include anything about the session, and there is no session handling/management involved. This sounds wrong at the beginning but allows the developer to build a RESTful API that will not rely on the state of the user session or data stored on the session.

RESTful is the application of the REST architecture in web services. REST is an architectural style for designing networked applications. The idea behind it is to avoid complex mechanisms and use plain HTTP. A typical RESTful web service will use HTTP to do the four **CRUD** (**Create, Retrieve, Update, and Delete**) operations.

Another benefit is that the code can be tested further as you don't have to recreate the whole user session in order to test a functionality.

If we take a look at this approach from the server-side point of view, we can see some benefits; as we are separating the responsibilities of every component, we can assign different machines and resources to each of them. This way of organizing the servers allows us also to scale the components we need independent of the others.

One downside of the approach we can easily see is that if the API goes down, everything will go down and the clients will not be able to do anything.

Summary

In this chapter we saw the concept of the project that we are going to build throughout this book. We also described in detail each part of the social network, the functionalities that will be available to the user, and the components that we will learn on building each section.

We also introduced the technical approach that we will use to build the social network, how it works, the benefits and downsides of using it, and why we choose this option instead of a more conventional one.

2 Setting Up the Environment

In this chapter, we will focus on the environment setup to be able to carry on with the next chapters. With this environment, you'll be able to put into practice all the examples provided in this book and try out everything.

By the end of this chapter, your computer will be ready to execute all the source code that we will develop in this book and will allow you to interact with the social network.

Using the provided virtual machine

Every computer is different, not only the hardware but also the software, operating system, user preferences, and so on. When you are the only developer in the team or you are developing solo, you usually only care about the differences between your machine and the server. This is completely different when you are in a team with more than one developer, because you share the code with people and you have to somehow ensure that your code executes exactly the same way as in the machine of another developer in the team.

A lot of techniques or approaches have been used to try to solve this issue and one of them is **virtual machines** (**VM**). A virtual machine is a simulation of a real machine using virtual components that mimics the computer architecture and allows you to run an operating system inside it. This also removes the issue of different operating systems across developers, such as Linux, Mac OS, or Windows.

This approach allows us to have the exact same copy of the machine and that means the same configuration, architecture, operating system, and software. Although we are using the same copy of the machine, anyone can make changes on the virtual machine, which would make our systems different again. To minimize this issue, we are going to use a tool called Vagrant.

Vagrant allows you to create and configure reproducible and portable virtual machines in an automated way. This tool relies on another piece of software called VirtualBox, which is basically the one that is going to give us the ability to virtualize our development environment and run the exact same copy.

Installing the required software

There are a few steps we should follow to get the environment up and running. For the installation instructions of the software, we will follow the guidelines on the websites, because if the software gets an update and the installation method changes, our instructions will be outdated. Following the ones found on the official website, we'll be assured that we have the latest instructions for the software that we are using.

The first step is to install VirtualBox. To do that, follow the instructions at `https://www.virtualbox.org/wiki/Downloads`.

Once VirtualBox is up and running, we can jump to the installing of Vagrant. We will again refer to the website for instructions on installation, which are available at `http://docs.vagrantup.com/v1/docs/getting-started/index.html#install_vagrant`.

The final piece of software we need to install is Git. You can find the installation instructions for your system at `http://git-scm.com/downloads`.

Getting a copy of the files

When you have VirtualBox, Vagrant, and Git installed on your system, we can jump to setup and configure the VM we are going to use. Let's see the following steps that are involved in the process:

1. Create a new folder to store the code from the book, this can be in your Documents folder, home folder, or even on the Desktop.
2. After that, open `terminal` if you're on Mac/Linux; open `git bash` if you're on Windows.
3. Navigate to the folder you just created using the `cd` command.
4. Get a copy of the Vagrant scripts by executing the following command:

   ```
   git clone https://github.com/christophervalles/ZF2-Vagrant.git
   ```

5. After getting a copy of the files, we need to execute a command inside the folder we just cloned. To do that, first enter that folder using the `cd` command. In my case, it looks like the following command:

   ```
   cd ZF2-Vagrant
   ```

6. We need to initialize the sub modules by executing the following command:
   ```
   git submodule update --init --recursive
   ```
7. After initializing the submodules, open the contents of the folder with your favorite text editor.
8. We need to adapt the configuration values of `Vagrantfile` to suit your system. The instructions can be found in the following section.

Vagrantfile

This is an overview of all the sections inside `Vagrantfile`, which will help you understand every piece of configuration code and how to tweak it to suit your system.

The first section looks like the following code snippet:

```
config.vm.synced_folder "~/source-api", "/var/source-api"
config.vm.synced_folder "~/source-client", "/var/source-client"
```

These two lines allow us to share a folder between our host computer and the VirtualBox. This means that we will edit the source code on the host machine and it will be executed on the VM. The line is composed of the command itself and two parameters: the first one indicating the path to the folder we want to share on our host machine and the second one the path of the shared folder inside the VM.

We need to adapt the first parameter to point the VM to the right folder on our host machine. In my case, the API code is located at `~/source-api` and the client code will be at `~/source-client`. If you stored your code on the desktop or somewhere else, adjust these two lines to point to the right folders on your host machine.

The next section on the configuration file looks like the following code snippet:

```
config.vm.box = "ubuntu-12.04"
config.vm.box_url = "http://files.vagrantup.com/precise64.box"
config.vm.network :private_network, ip: "192.168.56.2"
```

The first line specifies the name we want for the virtual machine; in our case, we don't need to change it.

The second line is a URL to download the base image; this is used by Vagrant to download the base operating system in case you don't have it. This parameter should not be changed.

The third line is the one you would probably want to tweak. In this line, we are defining the IP that the VM will have on our subnetwork. If you already have multiple VMs in the same subnetwork, you must adapt the given value to assign a new IP.

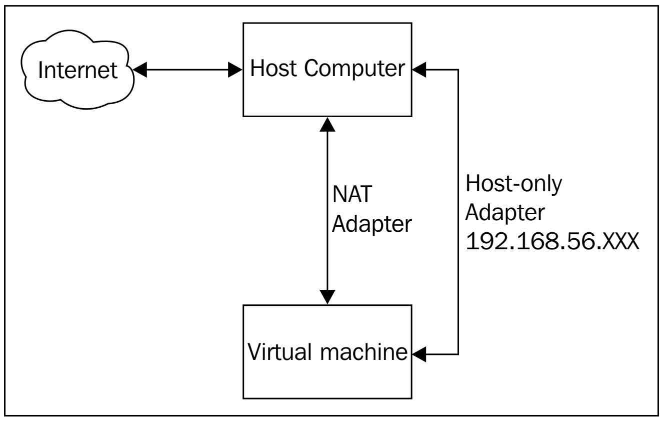
```
config.vm.provider :virtualbox do |vb|
  vb.customize ["modifyvm", :id, "--memory", "1024"]
end
```

These lines of code manage the amount of RAM memory we assign from our host to the VM. The default value is 1024 MB, but for our purpose, this can be changed to 512 MB or less; however, if you have plenty of RAM memory, just leave this value as it is.

VirtualBox has a few options for network adapters; the most common ones are Bridge, NAT, and host-only. In our setup, we will use NAT provided by default with Vagrant and host-only.

The NAT interface will connect the virtual machine to the Internet, using our computer as a router.

The host-only interface will connect our computer and the virtual machine using a subnetwork; this will allow us to have a static IP for the virtual machine without having to rely on our router and without consuming an IP of our main network.



Now we are going to take a look at the next section of code. The first line is to specify the folder where we have the recipes stored and the rest of the code is to add the recipes we want in the order we need them to be executed. In your case, you don't need to change anything.

Vagrant is using a piece of software called Chef to make the changes on the target computer, in our case a virtual machine. We use recipes to describe to Chef how we want certain configurations on the computer. Also, the recipes explain to Chef how the software gets installed on the system.

```
chef.cookbooks_path = "./cookbooks"

chef.add_recipe "apt"
chef.add_recipe "mysql::server"
chef.add_recipe "nginx"
chef.add_recipe "php-fpm"
chef.add_recipe "php::module_apc"
chef.add_recipe "php::module_curl"
chef.add_recipe "php::module_gd"
chef.add_recipe "php::module_mysql"
chef.add_recipe "custom::db"
chef.add_recipe "custom::hosts"
chef.add_recipe "custom::vhosts"
```

The last piece of configuration is a JSON structure that will set up a few configuration variables for Chef. These variables will be accessed from the recipes and will allow us to inject some data from Vagrantfile to customize the recipes and the final output of the Chef execution. This can be illustrated from the following code snippet:

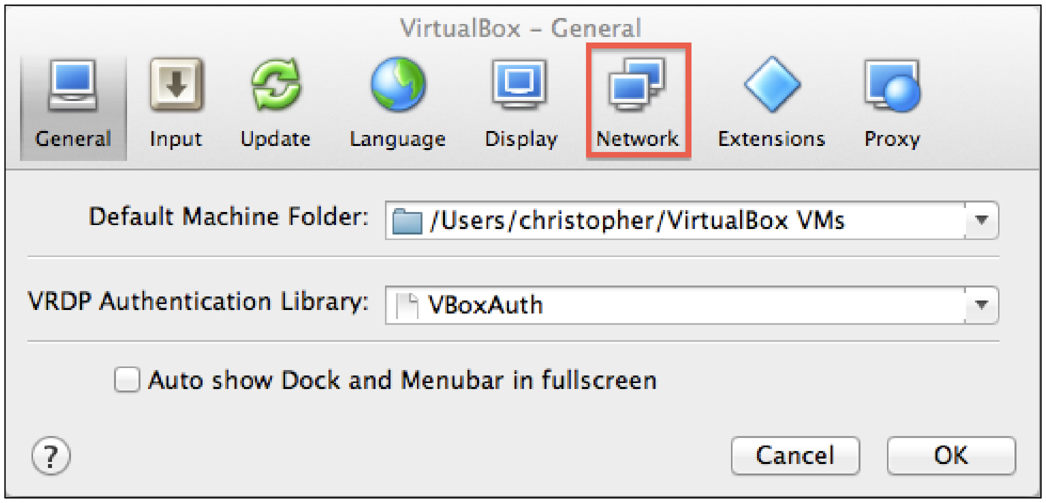
```
chef.json = {
  :mysql => {
    :server_root_password => "root",
    :server_repl_password => "repl",
    :server_debian_password => "debian"
  },
  :nginx => {
    :default_site_enabled => false
  }
}
```

As you can see, we are setting up a password for MYSQL users and also disabling the default virtualhost `nginx` file. Nothing needs to be changed on these two blocks of configuration for our setup.

VirtualBox subnet

In order to set up the host-only adapter, we should first set up a subnet between our computer and the virtual machine. The following is the step-by-step procedure, which shows how everything can be done:

1. Open the **VirtualBox** preference panel and click on **Network**. Your panel may differ from the one shown in the following screenshot, but you should be able to find the network section pretty easily:

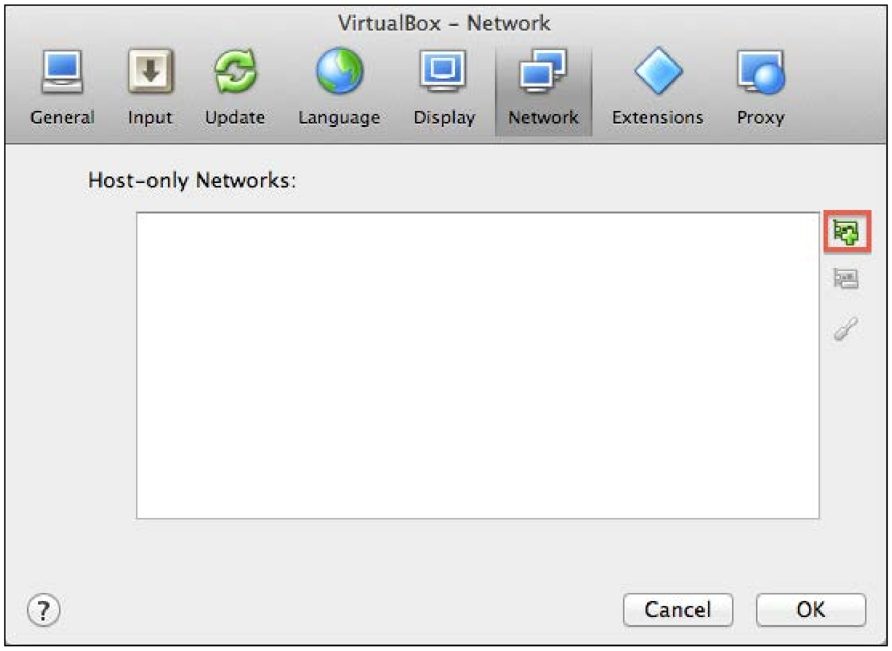


Default Machine Folder: /Users/christopher/VirtualBox VMs

VRDP Authentication Library: VBoxAuth

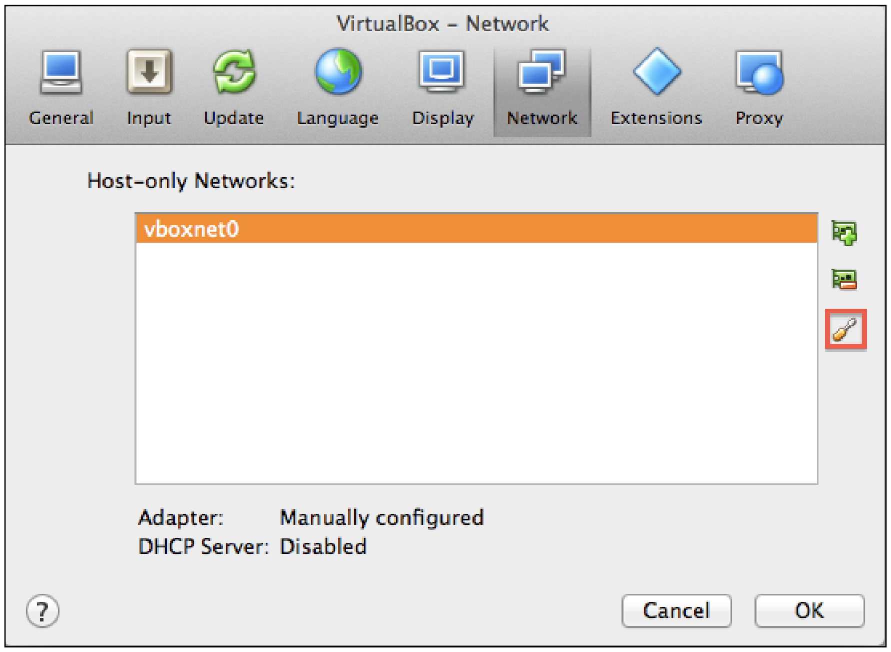
☐ Auto show Dock and Menubar in fullscreen

2. Click on the **Add** button to add a new subnet if the list is empty. Otherwise, jump to the next step.



Host-only Networks:

3. Click on the **Edit** button to review if the settings are as we need them. Notice that the name of the network can differ, but that will not affect the system.

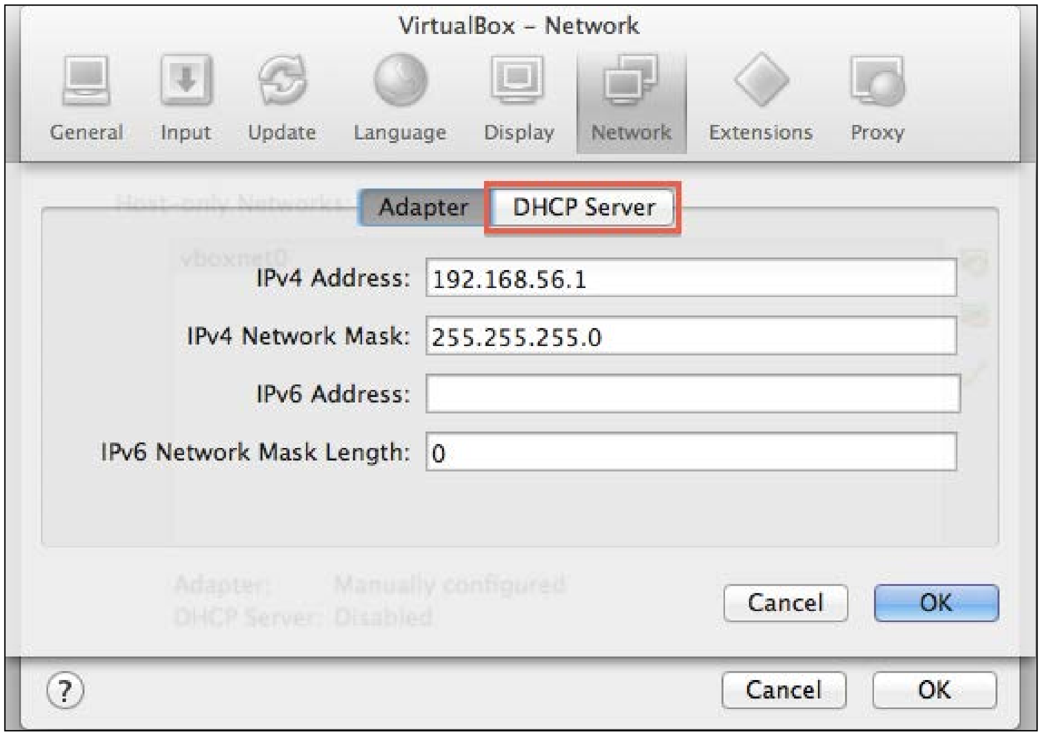


Host-only Networks:

vboxnet0

Adapter: Manually configured
DHCP Server: Disabled

4. After clicking on the **Edit** button, a new section will appear showing an IP address and a network mask. Make sure that the values are similar to the ones shown on the following screenshot. Also, make sure that **DHCP Server** is unchecked by clicking on the **DHCP Server** tab.



IPv4 Address: 192.168.56.1
IPv4 Network Mask: 255.255.255.0
IPv6 Address:
IPv6 Network Mask Length: 0

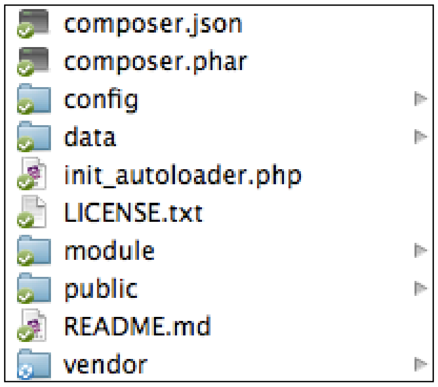
5. Save all the changes.

ZF2 skeleton

The last step is to download the Zend Framework 2 Skeleton application from the official Zend GitHub and put it on the two folders we are sharing with the VM. Then, we should install the dependencies of the framework using the composer.

To download the code, you can clone it using git or just download a ZIP file. For this example, we will download the ZIP file from `https://github.com/zendframework/ZendSkeletonApplication/archive/master.zip`.

After downloading the ZIP file, we need to uncompress the file in the two folders we share with the VM; in my case these folders are located in my home folder at `~/source-api` and `~/source-client`. The folders should look like the following screenshot when uncompressed:



Notice that you need to put two ZF2 installations, one on each folder.

After that, we should install all the dependencies using the composer. To achieve that, we need to first launch the virtual machine.

Hosts file

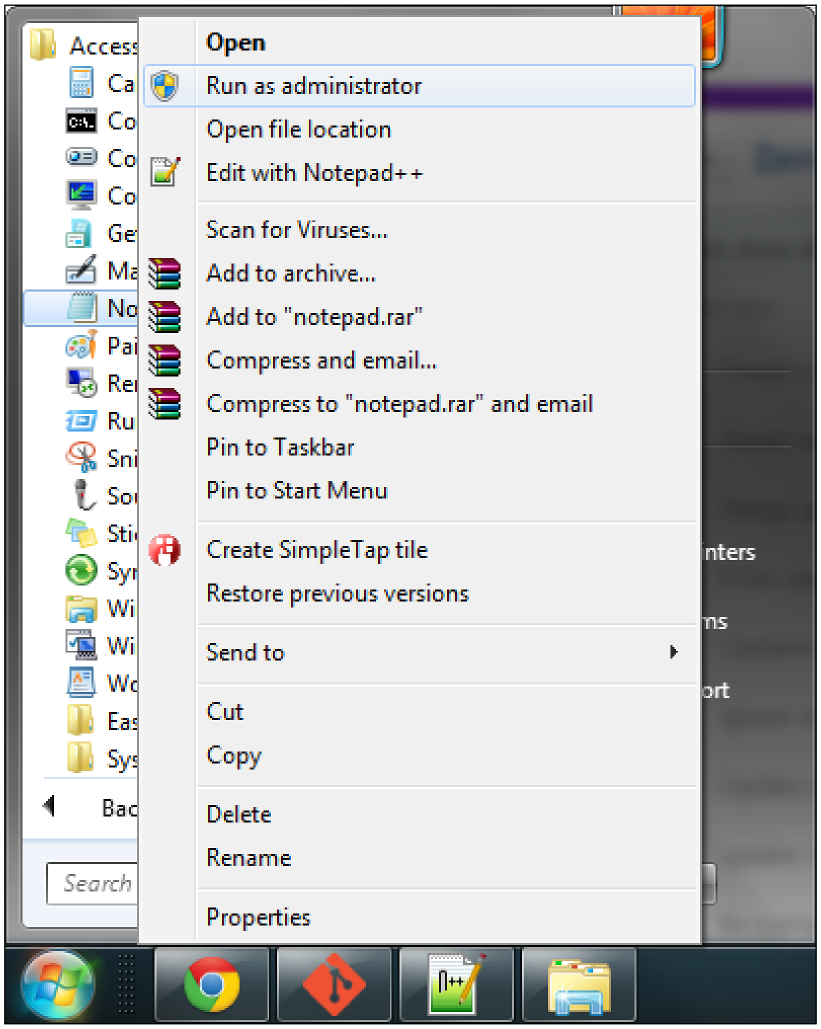
The last change we have to do in our host machine is point the hostname of the virtual host to the actual IP. This is an easy step and can be accomplished in two different ways depending on whether you are working on *nix or Windows. For *nix systems, execute the following two command lines:

```
sudo echo '192.168.56.2 zf2-api' >> /etc/hosts
sudo echo '192.168.56.2 zf2-client' >> /etc/hosts
```

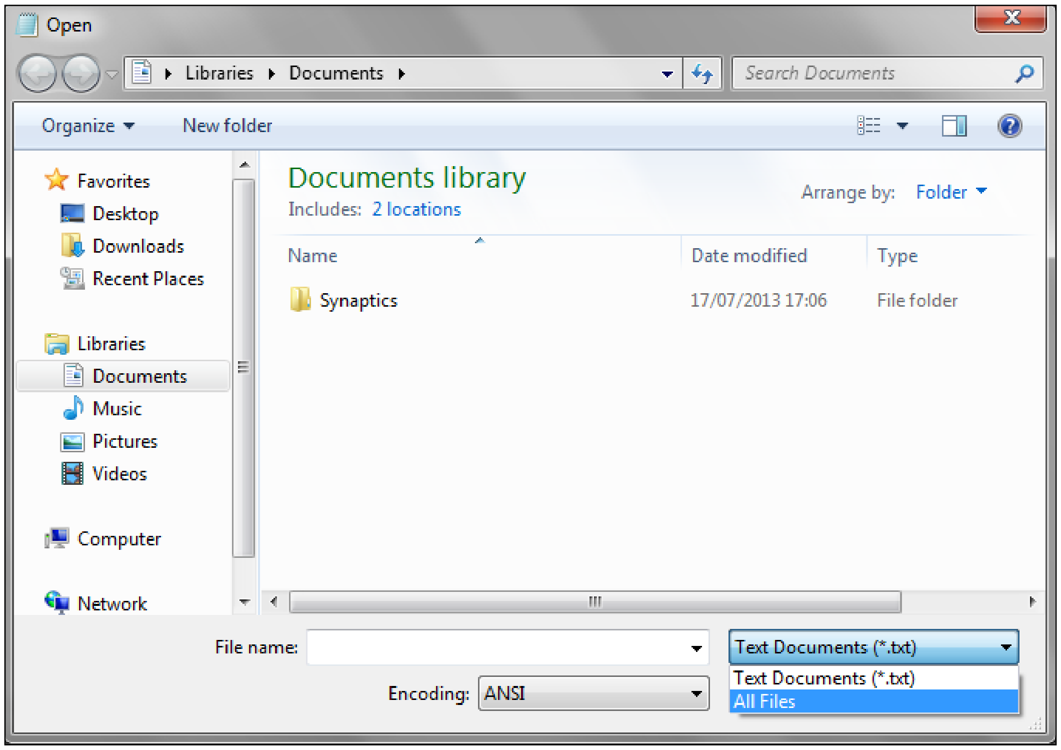
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

1. If you are working on Windows, you should run the Notepad as Administrator. To do that, go to the **Start** menu, click on **All Programs**, then go to **Accessories**, and finally, right-click on **Notepad**. On the contextual menu, you should see an option called **Run as Administrator** as shown in the following screenshot:

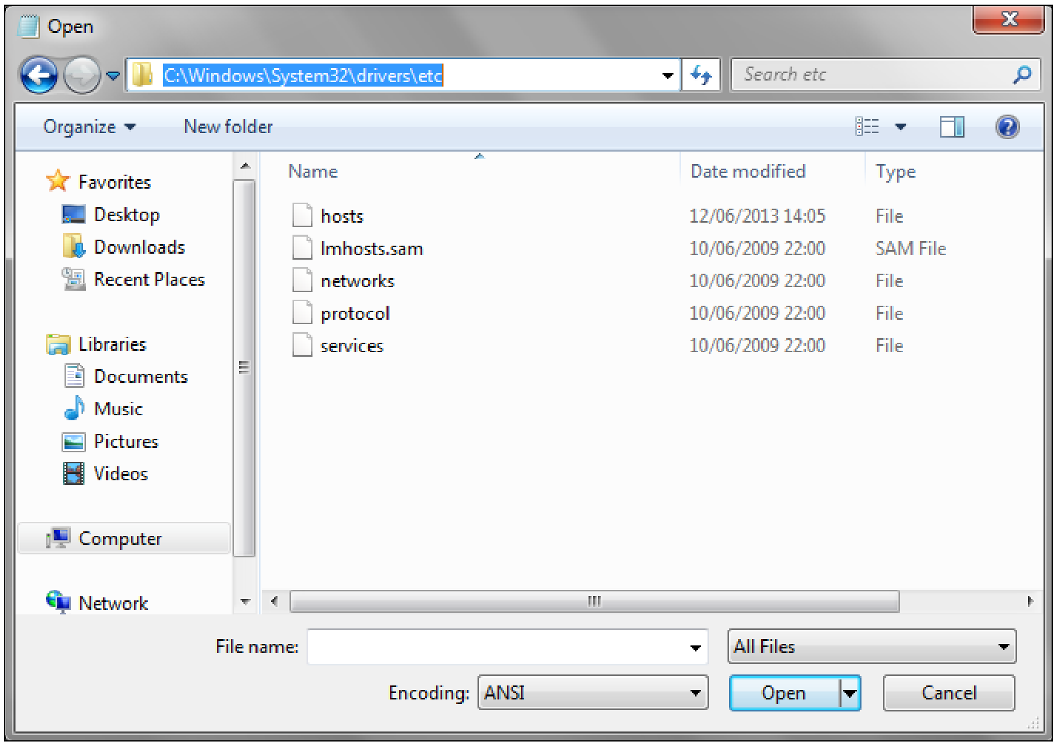


2. Now that Notepad is open, you should go to **File** | **Open** and select the option **All Files** in the file type drop-down list.



3. After this, you should type the following path on the top bar to open the file: `C:\Windows\System32\drivers\etc`

4. When you type the path, you should see a window like the following screenshot:

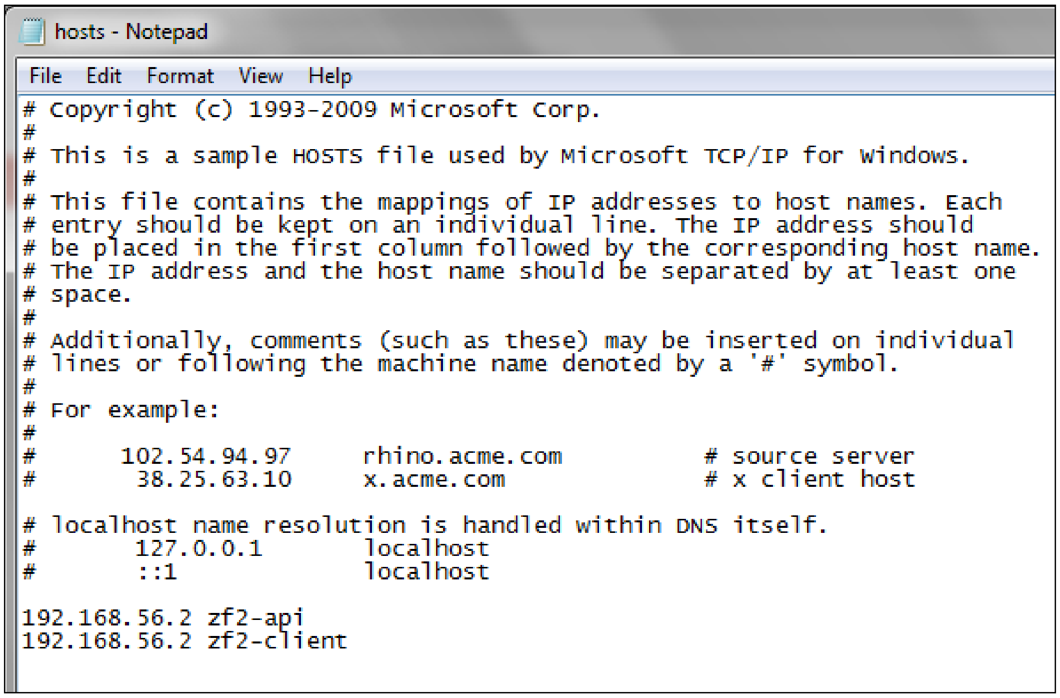


Name	Date modified	Type
hosts	12/06/2013 14:05	File
lmhosts.sam	10/06/2009 22:00	SAM File
networks	10/06/2009 22:00	File
protocol	10/06/2009 22:00	File
services	10/06/2009 22:00	File

5. Now you should open the hosts file shown in the list.
6. Once the file is open, you should manually add the following two code lines at the end:

```
192.168.56.2 zf2-api
192.168.56.2 zf2-client
```

7. The file after the change will look like the following screenshot:



```
# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97     rhino.acme.com          # source server
#       38.25.63.10     x.acme.com              # x client host

# localhost name resolution is handled within DNS itself.
#       127.0.0.1       localhost
#       ::1             localhost

192.168.56.2 zf2-api
192.168.56.2 zf2-client
```

8. Now, save the file and you are ready for the next steps.

When editing the hosts file, the IP you specify should match the IP of the virtual machine. If you have changed the IP on the Vagrantfile for any reason, make sure that the hosts file is pointing to the right IP.

Vagrant commands

We are now closer to having the environment ready to see the default project on our browser. We need to tell Vagrant to build the virtual machine based on the recipes and configurations we have and then install the dependencies of the framework. To do that, we have to open a terminal window or a command prompt depending on your operating system, and navigate to the folder where we have `Vagrantfile`. When we are there, we should run the following command:

```
vagrant up
```

After issuing the command, Vagrant will start building the virtual machine based on the configuration. If you don't get any errors in the build phase, you will have a VM up and running.

In some machines with Windows, when issuing the `vagrant up` command, an error appears. If that happens, try to start the virtual machine from the VirtualBox GUI and the error will disappear. The most common one says **VT-x features locked or unavailable in MSR**. If that's the case, you will need to enable the virtualization features on the BIOS of your computer. To enable that, refer to the manual of your computer.

As a reference, take a look at the Vagrant commands you can find here as you will need to use them while working on the project.

Command	Description
`vagrant up`	This will create the virtual machine if doesn't exist. Otherwise, it will just start the virtual machine if it stops.
`vagrant destroy`	This will completely destroy the virtual machine. Sounds bad, but we can recreate it with the help of the recipes we have inside the `cookbooks` folder.
`vagrant suspend`	This command will suspend the virtual machine, will pause it so we can continue from the point we left it.
`vagrant resume`	This will resume the virtual machine from the point we left it after we issued the `vagrant suspend` command.
`vagrant halt`	This will completely stop the virtual machine and is similar to powering off the system.

Now that we have the virtual machine running, we should install the dependencies of ZF2. To do this, we need to enter the virtual machine. To accomplish that, we need to launch the git bash program or terminal, depending on your operating system. Navigate to the folder where we have `Vagrantfile` and execute the following command:

```
Vagrant ssh
```

This command will connect the git bash program or terminal to the virtual machine and allow us to execute commands inside the virtual machine.

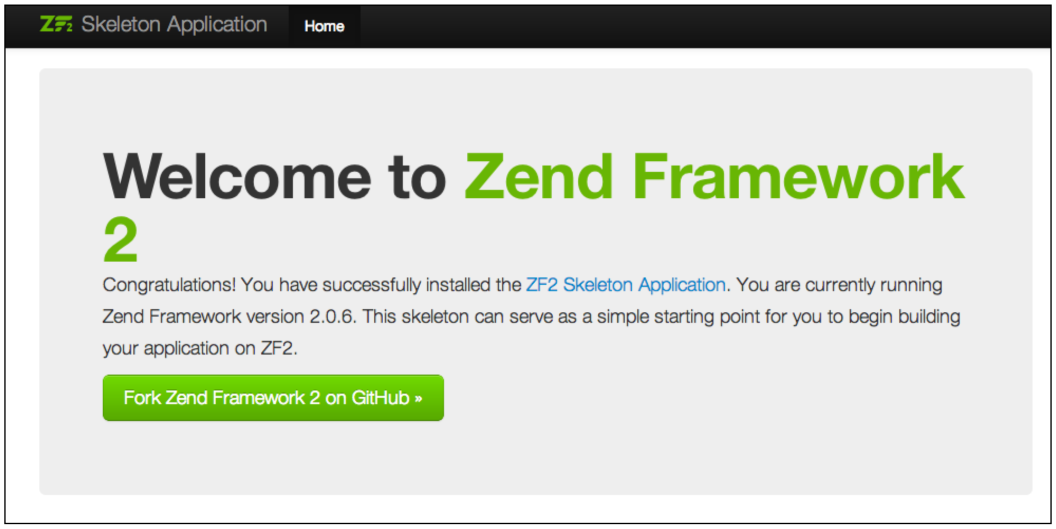
When executed, you will see a new prompt with a line like the following code line:

```
vagrant@precise64:~$
```

This means we are now inside the virtual machine and we can proceed to install the dependencies. In order to do that, we need to execute the following commands:

```
cd /var/source-api
php composer.phar self-update
php composer.phar install
cd /var/source-client
php composer.phar self-update
php composer.phar install
```

If you followed all the steps and didn't get any nasty error message, you should be able to open the browser, go to `http://zf2-client` or `http://zf2/api`, and see something like the following screenshot:



Welcome to Zend Framework 2

Congratulations! You have successfully installed the ZF2 Skeleton Application. You are currently running Zend Framework version 2.0.6. This skeleton can serve as a simple starting point for you to begin building your application on ZF2.

Summary

In this chapter, we learned a lot of different concepts related to the development environment. We saw how developers manage to work with the same environment and how they reduce the differences between them as much as possible by using virtual machines. Also, we learned how to use a powerful tool called Vagrant, combined with VirtualBox and Chef, to build a development environment. This environment will be used throughout this book to build our awesome social network and now your machine is ready to go.

In the next chapter, we will start reviewing the main components of the framework, how they interact to fulfill a request, and also the specific details of each one with their possible configurations.

3
Scratching the Surface of Zend Framework 2

In this chapter, we will focus on the basic elements of Zend Framework 2. You will get an in-depth look at the components involved on a simple request/response operation and this knowledge will be used and expanded on throughout the book. After that we will review how the welcome page is loaded.

By the end of this chapter, you will know the lifecycle of a request and how to interact with the different components of the framework to produce some output.

Bootstrap your app

There are two ways to bootstrap your ZF2 app. The default is less flexible but handles the entire configuration, and the manual is really flexible but you have to take care of everything.

The goal of the bootstrap is to provide to the application, `Zend\Mvc\Application`, with all the components and dependencies needed to successfully handle a request. A Zend Framework 2 application relies on the following six components:

- Configuration array
- ServiceManager instance
- EventManager instance
- ModuleManager instance
- Request object
- Response object

As these are the pillars of a ZF2 application, we will take a look at how these components are configured to bootstrap the app.

To begin with, we will see how the components interact from a high perspective and then we will jump into details of how each one works. When a new request arrives to our application, ZF2 needs to set up the environment to be able to fulfill it. This process implies reading configuration files and creating the required objects and services; attach them to the events that are going to be used and finally create the request object based on the request data.

Once we have the request object, ZF2 will tell the router to do his job and will inspect the request object to determine who is responsible for processing the data.

Once a controller and action has been identified as the one in charge of the request, ZF2 dispatches it and gives the controller/action the control of the program in order to execute the code that will interpret the request and will do something with it. This can be from accepting an uploaded image to showing a sign-up form and also changing data on an external database.

When the controller processes the data, sometimes a view object is generated to encapsulate the data that we should send to the client who made the request, and a response object is created.

After we have a response object, ZF2 sends it to the browser and the request ends.

Now that we have seen a very simple overview of the lifecycle of a request we will jump into the details of how each object works, the options available and some examples of each one.

Configuration array

Let's dissect the first component of the list by taking a look at the `index.php` file:

```
chdir(dirname(__DIR__));

// Setup autoloading
require 'init_autoloader.php';

// Run the application!
Zend\Mvc\Application::init(require 'config/application.config.php')-
>run();
```

As you can see we only do three things. The first thing is we change the current folder for the convenience of making everything relative to the root folder. Then we require the autoloader file; we will examine this file later. Finally, we initialize a `Zend\Mvc\Application` object by passing a configuration file and only then does the run method get called.

The configuration file looks like the following code snippet:

```
return array(
    'modules' => array(
        'Application',
    ),
    'module_listener_options' => array(
        'config_glob_paths'    => array(
            'config/autoload/{,*.}{global,local}.php',
        ),
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

This file will return an array containing the configuration options for the application. Two options are used: `modules` and `module_listener_options`. As ZF2 uses a module organization approach, we should add the modules that we want to use on the application here. The second option we are using is passed as configuration to the `ModuleManager` object. The `config_glob_path` array is used when scanning the folders in search of config files and the `module_paths` array is used to tell `ModuleManager` a set of paths where the module resides.

> ZF2 uses a module approach to organize files. A module can contain almost anything, simple PHP files, view scripts, images, CSS, JavaScript, and so on. This approach will allow us to build reusable blocks of functionality and we will adhere to this while developing our project.

PSR-0 and autoloaders

Before continuing with the key components, let's take a closer look at the `init_autoloader.php` file used in the `index.php` file. As is stated on the first block comment, this file is more complicated than it's supposed to be. This is because ZF2 will try to set up different loading mechanisms and configurations.

```
if (file_exists('vendor/autoload.php')) {
    $loader = include 'vendor/autoload.php';
}
```

The first thing is to check if there is an `autoload.php` file inside the `vendor` folder; if it's found, we will load it. This is because the user might be using composer, in which case composer will provide a PSR-0 class loader. Also, this will register the namespaces defined by composer on the loader.

PSR-0 is an autoloading standard proposed by the PHP Framework Interop Group (`http://www.php-fig.org/`) that describes the mandatory requirements for autoloader interoperability between frameworks. Zend Framework 2 is one of the projects that adheres to it.

```
if (getenv('ZF2_PATH')) {
    $zf2Path = getenv('ZF2_PATH');
} elseif (get_cfg_var('zf2_path')) {
    $zf2Path = get_cfg_var('zf2_path');
} elseif (is_dir('vendor/ZF2/library')) {
    $zf2Path = 'vendor/ZF2/library';
}
```

In the next section we will try to get the path of the ZF2 files from different sources. We will first try to get it from the environment, if not, we'll try from a directive value in the `php.ini` file. Finally, if the previous methods fail the code, we will try to check whether a specific folder exists inside the `vendor` folder.

```
if ($zf2Path) {
    if (isset($loader)) {
        $loader->add('Zend', $zf2Path);
    } else {
        include $zf2Path . '/Zend/Loader/AutoloaderFactory.php';
        Zend\Loader\AutoloaderFactory::factory(array(
            'Zend\Loader\StandardAutoloader' => array(
                'autoregister_zf' => true
            )
        ));
    }
}
```

Finally, if the framework is found by any of these methods, based on the existence of the composer autoloader, the code will just add the `Zend` namespace or will instantiate an internal autoloader, `Zend\Loader\Autoloader`, and use it as a default.

As you can see, there are multiple ways to set up the autoloading mechanism on ZF2 and at the end what matters is which one you prefer, as all of them in essence will behave the same.

ServiceManager

After all this execution of code, we arrive at the last section of the `index.php` file where we actually instantiate the `Zend\Mvc\Application` object.

As we said, there are two methods of creating an instance of `Zend\Mvc\Application`. In the default approach, we call the static method `init` of the class by passing an optional configuration as the first parameter. This method will take care of instantiating a new `ServiceManager` object, storing the configuration inside, loading the modules specified in the configuration, and getting a configured `Zend\Mvc\Application`.

`ServiceManager` is a service/object locator that implements the Service Locator design pattern; its responsibility is to retrieve other objects.

```
$serviceManager = new ServiceManager(
    new Service\ServiceManagerConfig($smConfig)
);
$serviceManager->setService('ApplicationConfig', $configuration);
$serviceManager->get('ModuleManager')->loadModules();

return $serviceManager->get('Application')->bootstrap();
```

As you can see, the `init` method calls the `bootstrap()` method of the `Zend\Mvc\Application` instance.

Service Locator is a design pattern used in software development to encapsulate the process of obtaining other objects. The concept is based on a central repository that stores the objects and also knows how to create them if required.

EventManager

This component is designed to provide multiple functionalities. It can be used to implement simple observer patterns, and also can be used to do aspect-oriented design or even create event-driven architectures.

The basic operations you can do over these components is attaching and detaching listeners to named events, trigger events, and interrupting the execution of listeners when an event is fired.

Let's see a couple of examples on how to attach to an event and how to fire them:

```
//Registering an event listener
$events = new EventManager();
$events->attach(array('EVENT_NAME'), $callback);
//Triggering an event
$events->trigger('EVENT_NAME', $this, $params);
```

Inside the bootstrap method of `Zend\Mvc\Application`, we are registering the events of `RouteListener`, `DispatchListener`, and `ViewManager`. After that, the code is instantiating a new custom event called `MvcEvent` that will be used as the target when firing events. Finally, this piece of code will fire the bootstrap event.

ModuleManager

Zend Framework 2 introduces a completely redesigned `ModuleManager`. This new module has been built with simplicity, flexibility, and reuse in mind. These modules can hold everything from PHP to images, CSS, library code, views, and so on.

The responsibility of this component in the bootstrap process of an app is loading the available modules specified by the config file. This is accomplished by the following code line located in the `init` method of `Zend\Mvc\Application`:

```
$serviceManager->get('ModuleManager')->loadModules();
```

This line, when executed, will retrieve the list of modules located at the config file and will load each module.

Each module has to contain a file called `Module.php` with the initialization of the components of the module if needed. This will allow the module manager to retrieve the configuration of the module. Let's see the usual content of this file:

```
namespace MyModule;

class Module
```

```
{
    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\ClassMapAutoloader' => array(
                __DIR__ . '/autoload_classmap.php',
            ),
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __
NAMESPACE__,
                ),
            ),
        );
    }

    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php';
    }
}
```

As you can see we are defining a method called `getAutoloaderConfig()` that provides the configuration for the autoloader to `ModuleManager`. The last method `getConfig()` is used to provide the configuration of the module to `ModuleManager`; for example, this will contain the routes handled by the module.

Request object

This object encapsulates all data related to a request and allows the developer to interact with the different parts of a request. This object is used in the constructor of `Zend\Mvc\Application` and also is set inside `MvcEvent` to be able to retrieve when some events are fired.

Response object

This object encapsulates all the parts of an HTTP response and provides the developer with a fluent interface to set all the response data. This object is used in the same way as the request object. Basically it is instantiated on the constructor and added to `MvcEvent` to be able to interact with it across all the events and classes.

The request object

As we said, the request object will encapsulate all the data related to a request and provide the developer with a fluent API to access the data. Let's take a look at the details of the request object in order to understand how to use it and what it can offer to us:

```
use Zend\Http\Request;

$string = "GET /foo HTTP/1.1\r\n\r\nSome Content";
$request = Request::fromString($string);

$request->getMethod();
$request->getUri();
$request->getUriString();
$request->getVersion();
$request->getContent();
```

This example comes directly from the documentation and shows how a request object can be created from a string, and then access some data related with the request using the methods provided. So, every time we need to know something related to the request, we will access this object to get the data we need.

If we check the code on `Zend\Http\PhpEnvironment\Request.php`, the first thing we can notice is that the data is populated on the constructor using the `superglobal` arrays. All this data is processed and then populated inside the object to be able to expose it in a standard way using methods.

To manipulate the URI of the request you can get/set the data with three methods, two getters and one setter. The only difference with the getters is that one returns a plain string and the other returns an `HttpUri` object.

- `getUri()` and `getUriString()`
- `setUri()`

To retrieve the data passed in the request, there are a few specialized methods depending on the data you want to get:

- `getQuery()`
- `getPost()`
- `getFiles()`
- `getHeader()` and `getHeaders()`

About the request method, the object has a general way to know the method used, returning a string or nine specialized functions that will test specific methods based on the RFC 2616, which defines the standard methods for an HTTP request.

- `getMethod()`
- `isOptions()`
- `isGet()`
- `isHead()`
- `isPost()`
- `isPut()`
- `isDelete()`
- `isTrace()`
- `isConnect()`
- `isPatch()`

Finally, two more methods are available in this object that will test special requests such as AJAX and requests made by a flash object.

- `isXmlHttpRequest()`
- `isFlashRequest()`

Notice that the data stored on the `superglobal` arrays when populated on the object are converted from an Array to a Parameters object.

The `Parameters` object lives in the `Stdlib` section of ZF2, a folder where common objects can be found and used across the framework. In this case, the `Parameters` class is an extension of `ArrayObject` and implements `ParametersInterface` that will bring `ArrayAccess`, `Countable`, `Serializable`, and `Traversable` functionality to the parameters stored inside the object. The goal with this object is to provide a common interface to access data stored in the `superglobal` arrays. This expands the ways you can interact with the data in an object-oriented approach.

The router

This component of the framework has been rewritten from scratch and it's really powerful and flexible, allowing you to create a lot of combination of routes. The main functionality of the router is matching a request to a given controller/action. It also assembles URLs based on the defined routes. This process is accomplished by inspecting the URI and attempting to match it with a given route that has some constraints, or by generating the URL based on the parameters defined on the routes.

The general flow of this component is as follows:

- A request arrives and the framework creates a request object
- Then the route parses the URI to identify the segments
- After that the router iterates through the list of routes previously ordered
- On each iteration, the segments of the URI are checked to see if it matches the current route
- When a route matches the URI, the request is dispatched to the specific controller and action is configured on the route

We have two parts on the routing system: the router itself that holds the logic to test URIs against routes and generate URLs, and the route that will hold the data of how to match a specific request. The route can also hold default data, extra parameters, and so on.

```
'wall' => array(
    'type' => 'Zend\Mvc\Router\Http\Segment',
    'options' => array(
        'route'    => '/api/wall[/:id]',
        'constraints' => array(
            'id' => '\w+'
        ),
        'defaults' => array(
            'controller' => 'Wall\Controller\Index'
        ),
    ),
)
```

This is an example of a route. As you can see we are specifying a parameter called `id` and we are adding constraints on this `id`.

The router in ZF2 can be used not only for HTTP requests but also for CLI applications. In our case, we will focus our attention on the website-related router but the essence of the router will be the same for the CLI apps.

Zend Framework 2 provides two different routers: `SimpleRouteStack` and `TreeRouteStack`, and eight possible route types to choose from: `Hostname`, `Literal`, `Method`, `Part`, `Regex`, `Scheme`, `Segment`, and `Query`. Here is a detailed view of all of them.

SimpleRouteStack

This is a basic router that gets all the configured routes and loops through them in a LIFO order until a match is found. The routes have priority and as the router is looping them in a LIFO order, the routes that are more likely to be matched (the ones that match more often) should be registered last, and the one that matches less should be registered first. Also, an easy way to set up the priority of the routes is using the third parameter of the `AddRoute()` method.

```
$router->addRoute('foo', $route, 10);
```

That's an example of how you can specify priorities on routes; just pass a number and the routes will be ordered by those values.

TreeRouteStack

This router is a little bit more complex than `SimpleRouteStack` because this allows you to create a tree of routes and will use a binary-tree algorithm to match the routes. The basic configuration of a `TreeRouteStack` route will consist of the following:

- A base route that will be the root of the tree and all the following routes will extend this one.
- An optional configured instance of `RoutePluginManager` to be able to lazy-load routes.
- Another optional parameter will be `may_terminate` that tells the router that no other route will follow this one.
- Finally, a `child_routes` array that can also be optional. If we specify routes here, they will extend the main one; also, a child route can be another `TreeRouteStack`.

You can add routes one by one on both routers by using the `AddRoute()` method or in batches using `AddRoutes()`.

```
$route = Part::factory(array(
     'route' => array(
         'type' => 'literal',
         'options' => array(
             'route' => '/',
             'defaults' => array(
                 'controller' => 'Application\Controller\
IndexController',
                 'action' => 'index'
```

```
                )
            ),
        ),
        'child_routes' => array(
            'forum' => array(
                'type' => 'literal',
                'options' => array(
                    'route' => 'forum',
                    'defaults' => array(
                        'controller' => 'Application\Controller\
ForumController',
                        'action' => 'index'
                    )
                )
            )
        )
    ));
```

As you can see, the first route we defined is the home page – the root of the domain. Then we defined the route for the forum extending from the base one.

Hostname

```
$route = Hostname::factory(array(
    'route' => ':subdomain.domain.tld[/:type]',
    'constraints' => array(
        'subdomain' => 'fw\d{2}'
    ),
    'defaults' => array(
        'type' => 'json',
    ),
));
```

This is a usage example of how to configure a `Hostname` route. In the `route` parameter, we specify the `hostname` to match in the URI. Also, you can set up parameters, such as `subdomain` in this case.

You can set up constraints and default values on the route; in this particular example, we are specifying a regular expression that will limit the matching of the route to all the subdomains starting with the letters `fw` and followed by two digits. Sometimes you will need to pass values to the controllers based on the URL or define some default values for parameters on the URL. In this example we are setting the default value of `type` to the `json` string.

Literal

This route will match a URI path literally as specified by the `route` parameter. As usual, the default data will be the parameters you want returned on a match.

```
$route = Literal::factory(array(
    'route' => '/foo',
    'defaults' => array(
        'controller' => 'Application\Controller\IndexController',
        'action' => 'foo'
    ),
));
```

Method

This type of route will match if the HTTP method used on the request matches the one configured in the route using the `verb` parameter. We can specify multiple methods on the same route if we separate them with comma. A valid list of request methods can be found in the *RFC 2616 Sec. 5.1.1* section.

```
$route = Method::factory(array(
    'verb' => 'post,put',
    'defaults' => array(
        'controller' => 'Application\Controller\IndexController',
        'action' => 'form-submit'
    ),
));
```

Part

The `Part` route type not only extends `TreeRouteStack` but also implements `RouteInterface`. It means that this route can hold a tree of possible routes based on the URI segments. Let's explore the idea of a forum application. We want to show the home of the website when you go to the root of the domain, and then we want to display the forum if you go to `domain.com/forum`. We can accomplish that using the `Part` route type and we will need to configure it as shown in the following code snippet:

```
$route = Part::factory(array(
    'route' => array(
        'type' => 'literal',
        'options' => array(
            'route' => '/',
            'defaults' => array(
                'controller' => 'Application\Controller\
IndexController',
                'action' => 'index'
            )
        ),
    ),
    'route_plugins' => $routePlugins,
    'may_terminate' => true,
    'child_routes' => array(
        'forum' => array(
            'type' => 'literal',
            'options' => array(
                'route' => 'forum',
                'defaults' => array(
                    'controller' => 'Application\Controller\
ForumController',
                    'action' => 'index'
                )
            )
        )
    )
));
```

As you can see in the previous code example, we are defining a `child_routes` parameter with more routes inside, thus creating a tree of routes. Of course, the routes added as child will extend the parent route and therefore the matching URI path. Now with this in place if we get a request on `domain.com/forum`, ZF2 will inspect the request object and will match the URI against the routes we defined. Finally, as the forum route will match, the request will be dispatched to `ForumController`.

Regex

A `Regex` route will be based on a regular expression test match of the URI. You need to specify `regex` in the `regex` parameter and it can be any valid regular expression.

When working with this route we have to keep two things in mind. The first one is to use named captures on regex for any value we want to return as parameter on a match, the second is that we need to specify a value for the `spec` parameter to map each regex capture to the specific parameter. This will be used by the URL helper when building URLs based on the routes. For instance:

```
$route = Regex::factory(array(
    'regex' => '/blog/(?<id>[a-zA-Z0-9_-]+)(\.(?<format>(json|html|xm
l|rss)))?',
    'defaults' => array(
        'controller' => 'Application\Controller\BlogController',
        'action'     => 'view',
        'format'     => 'html',
    ),
    'spec' => '/blog/%id%.%format%',
));
```

In the previous example, you will notice that regex is using named captures enclosed in `<>` and we are specifying the format of the URL in the `spec` parameter, identifying each parameter with a text between percentage symbols. This will help the router to put the value of the parameters in the correct place in the URL while assembling it.

Scheme

The `Scheme` route type will match a URI based only on the URI scheme. This route type is similar to the `Literal` route because the match has to be exact and will return the default parameters on a successful match.

```
    $route = Scheme::factory(array(
        'scheme' => 'https',
        'defaults' => array(
            'https' => true,
        ),
    ));
```

As you can see in the previous code example, if the route matches, we will return a parameter called `https` with a `true` value.

Segment

```
    $route = Segment::factory(array(
        'route' => '/:controller[/:action]',
        'constraints' => array(
            'controller' => '[a-zA-Z][a-zA-Z0-9_-]+',
            'action'     => '[a-zA-Z][a-zA-Z0-9_-]+',
        ),
        'defaults' => array(
            'controller' => 'Application\Controller\IndexController',
            'action'     => 'index',
        ),
    ));
```

This is an example of how to use the `Segment` route that will allow us to match against any segment of the URI path. In the `route` parameter, the segments are defined using a specific notation; this is a colon followed by alphanumeric characters. The segments on a URI can be optional and that will be denoted by enclosing the segment in brackets.

Each segment must have constraints associated with it; the constraint will be a regular expression defining the requirements for a positive match.

Wildcard

This type of route allows you to get all the segments of a URI specifying the separator between keys and values. Consider the following code example to get a better understanding:

```
'route_name' => array(
    'type' => 'Zend\Mvc\Router\Http\Wildcard',
    'options' => array(
        'key_value_delimiter' => '/',
        'param_delimiter' => '/',
    )
)
```

That's how the route looks when configured, and it will work like this. If you have a URI such as `/id/1/name/john`, this route will return the two variables `id` and `name`. The first one will contain the number `1` and the second will contain the string `john`.

Query

This is the last type of route offered by the route system of Zend Framework 2. This will allow us to capture and define parameters on a query string. An important point is that it's designed to be used as a child route.

```
$route = Part::factory(array(
    'route' => array(
        'type'    => 'literal',
        'options' => array(
            'route'    => 'page',
            'defaults' => array(
            ),
        ),
    ),
    'may_terminate' => true,
    'route_plugins'  => $routePlugins,
    'child_routes'  => array(
        'query' => array(
            'type' => 'Query',
            'options' => array(
                'defaults' => array(
                    'foo' => 'bar'
                )
            )
        ),
    ),
));
```

In this example we can see how the route has been defined as a child of a `Literal` one. Then we can use this to generate a URL with a query string. For example:

```
$this->url(
    'page/query',
    array(
        'name'=>'my-test-page',
        'format' => 'rss',
        'limit' => 10,
    )
);
```

In the previous code example, we should specify `/query` in order to activate the child route while assembling the URL and be able to append the parameters to the query string of the URL generated.

DispatchListener

One of the big changes on this new version of the framework is that everything is built around events. That's why the dispatchers in just an event listener waiting for the `EVENT_DISPATCH` event.

When this event is triggered, it will be captured by `DispatchListener` and this will lead to a few things. Let us see the following code snippet step-by-step to fully understand the dispatch flow:

```
$routeMatch = $e->getRouteMatch();
$controllerName = $routeMatch->getParam('controller', 'not-found');
$application = $e->getApplication();
$events = $application->getEventManager();
$controllerLoader = $application->getServiceManager()-
>get('ControllerLoader');
```

First the listener will retrieve information from the event itself, such as controller, route matched, and the application, and will try to load the controller. Notice that we are passing a second parameter to the `$routeMatch->getParam()` method; if for any reason the parameter does not exists on the route, this one will be the returned value by default.

If everything is successful, it will execute the method dispatch of the controller. Let's see the actual code that accomplishes that.

```
try {
$return = $controller->dispatch($request, $response);
} catch (\Exception $ex) {
   $e->setError($application::ERROR_EXCEPTION)
   ->setController($controllerName)
      ->setControllerClass(get_class($controller))
      ->setParam('exception', $ex);
$results = $events->trigger(MvcEvent::EVENT_DISPATCH_ERROR, $e);
$return = $results->last();
if (! $return) {
$return = $e->getResult();
}
}

return $this->complete($return, $e);
```

The dispatch responsibility is shared between `DispatchListener` and the controller; actually the dispatch itself is located in the `AbstractActionController` class. This method has the logic to examine the `RouteMatch` object, get the action to be called, and figure out if it exists and is callable. If it is, it will take care of calling it and returning the results.

```
$action = $routeMatch->getParam('action', 'not-found');
$method = static::getMethodFromAction($action);

if (!method_exists($this, $method)) {
$method = 'notFoundAction';
}

$actionResponse = $this->$method();
$e->setResult($actionResponse);
return $actionResponse;
```

Front controller

The concept of front controller no longer exists in ZF2. In this new version of the framework, any class that implements `DispatchableInterface` can be a MVC controller.

```
interface DispatchableInterface
{
public function dispatch(Request $request, Response $response = null);
}
```

As you can see, this interface only defines one method: dispatch. You can create a controller following this approach, but you will need to define this method and the logic in every controller or in a parent controller. In such cases, it's much better to use the already coded `AbstractActionController` that provided this method and the logic required to dispatch a request.

The MVC layer of ZF2 provides a few more interfaces that you can implement to provide the controllers with even more functionality. Let's take a look at the common interfaces that can be used with controllers.

InjectApplicationEvent

This interface is used to tell the `Application` instance to inject its `MvcEvent` into this controller. The controller can access the request, response, and route matched using the `MvcEvent` object and can modify the contents if needed. Let's see an example on how this interface can be used inside a controller to validate that the parameter `id` is present in the request:

```
$matches = $this->getEvent()->getRouteMatch();
$id      = $matches->getParam('id', false);
if (!$id) {
    $response = $this->getResponse();
    $response->setStatusCode(500);
    $this->getEvent()->setResult('Invalid identifier; cannot complete
request');
    return;
}
```

The `InjectApplicationEvent` class defines two methods: `setEvent()` and `getEvent()`. If you take a look at the first line of the previous code example, you can see that we are calling `getEvent()` to retrieve the `MvcEvent` object; from there we will call `getRouteMatch()` to get the `RouteMatch` object. From there it's just a matter of getting a parameter providing `false` as the default value. In case the parameter is not in the request, `false` will be returned and the condition below will succeed taking care of returning the proper error.

ServiceLocatorAware

In ZF2, you usually develop your controllers in a way that the dependencies are injected on the constructor or using setters. Sometimes you can have objects that will not be needed on every request handled by the controller and it makes no sense to have it injected every time, for example during navigation, forms, and pagination. In this case, you will need to use the `ServiceLocatorAware` interface to tell `ServiceManager` to inject itself to the controller so that you can use it to retrieve the dependencies you need.

```
class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        $config = $this->getServiceLocator()-
>get('ApplicationConfig');

        //More code here

    }
}
```

This is a usage example of this interface. As you can see on the class definition, we are extending the `AbstractActionController` class. We said before that this controller already provides some boilerplate code; in this case, this class not only implements this interface but also implements `InjectApplicationEvent` and `EventManagerAware`. So, if you stick with it while developing your controllers, you will benefit from this automatically.

As in the case of the previous interface, this one also defines two methods: `setServiceLocator()` and `getServiceLocator()`.

EventManagerAware

As with the other two interfaces, this one is also to tell `ServiceManager` that we want something injected by default. In this case, we want the `EventManager` instance to be injected by default. This interface defines two methods: `setEventManager()` and `getEventManager()`.

Let's see how to use it on a controller:

```
$eventManager = $this->getEventManager();
$eventManager->trigger(INDEX_LOADED);
```

As you can see, once the interface is implemented, it is easy to retrieve the `EventManager` instance and use it; in this case we are firing a custom event called `INDEX_LOADED`.

Controller plugins

The MVC layer of Zend Framework 2 not only provides interfaces to help you with controllers but also provides plugins to use inside controllers. There are a few types, but the most common ones are `URL`, `Redirect`, `FlashMessenger`, and `PostRedirectGet`. In order to ease the usage of these plugins, ZF2 provides `PluginManager`. As usual, to utilize it you should define three methods: `setPluginManager()`, `getPluginManager()`, and `plugin()`, unless you are extending the `AbstractActionController` class that provides this by default.

```
$pm = $this->getPluginManager();
$flashMessenger = $pm->get('FlashMessenger');
$flashMessenger->addMessage('Using the flash messenger plugin');
```

As you can see in the example, it is pretty straightforward; you only need to get the `PluginManager` configuration, and then get any plugin you need, referencing them by name as in the previous code example, the `FlashMessenger` view helper.

Controller types

The last thing provided by the new implementation of the MVC pattern in ZF2 is a collection of two base controllers that we can extend to create our controllers. These two base controllers are designed to provide some functionality by default and both of them extend a common controller called `AbstractController`.

AbstractActionController

As you probably may have noticed, ZF2 provides a controller called `AbstractActionController` that implements all the interfaces we discussed in this chapter. This should be your starting point for a controller. For sure you can create it from scratch if you need it, but it will be simpler and will require less duplication of code if we choose this provided controller at the start.

To be able to work, this controller has two conventions. The controller expects an action parameter defined in the `RouteMatch` object, which means you have to define the parameter when you define the route itself. Also, the second constraint is that the action parameter will be converted to camelCase format, and then we will append the word action to create the method name, which means that a method with the final name should be present on the controller. The controller will check if the method exists, otherwise the method `notFoundAction()` will be executed.

From a global point of view, if we are using this type of controller as starting point we will need to define controller and action parameters in all the routes in order to tell the router and dispatcher who has to fulfill the request.

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class BarController extends AbstractActionController
{
    public function bazAction()
    {
        return array('title' => __METHOD__);
    }

    public function batAction()
    {
        return array('title' => __METHOD__);
    }
}
```

As you can see, a controller extending `AbstractActionController` is pretty short, easy, and simple, compared to controllers made from scratch.

AbstractRestfulController

This is also a specialized version of `AbstractController`. In this case, this controller will help us when we create RESTful web services.

This controller will map all the actions used on a RESTful approach to predefined methods that we have to implement by convention. Let's see how the HTTP methods and controller methods are connected.

HTTP method	Controller method	Parameters	Description
`GET`	`get()` `getList()`	id. Optional	If the id parameter is passed as an argument, `get()` will be called; otherwise `getList()` is executed.
`POST`	`create()`	$data	This method expects the data as an argument, usually the `$_POST superglobal` array.

HTTP method	Controller method	Parameters	Description
`PUT`	`update()`	id	An id parameter must be present on the request to be able to update the entity based on the provided data.
`DELETE`	`delete()`	id	This method also needs an id to identify which entity to delete.

Also, this controller has some constrains about the response it should generate for each action and the response codes each one should have.

HTTP method	Response code	Headers	Body content
`GET`	200	Nothing extra	`get()` will return one entity and `getList()` a list of entities
`POST`	201	The Location header should provide the URL of the entity	The body should contain the representation of the entity
`PUT`	200 or 202	Nothing extra	A representation of the entity should be returned
`DELETE`	200 or 204	Nothing extra	EMPT.

The controllers that extend this one can also be used as a normal controller; this means that you can still define methods ending in action and call them as usual. This can be useful to organize the code that shows a form for example and the code that actually processes the data.

The response object

The response object is a simple class that wraps the HTTP response data. There are two ways to create a response object, the first one is using a string containing the response and the other one is instantiating the `Zend\Http\Response` class.

```
use Zend\Http\Response;
$response = Response::fromString(<<<EOS
HTTP/1.0 200 OK
HeaderField1: header-field-value
HeaderField2: header-field-value2

<html>
<body>
    Hello World
</body>
</html>
EOS);

// OR

$response = new Response();
$response->setStatusCode(Response::STATUS_CODE_200);
$response->getHeaders()->addHeaders(array(
    'HeaderField1' => 'header-field-value',
    'HeaderField2' => 'header-field-value2',
);
$response->setContent(<<<EOS
<html>
<body>
    Hello World
</body>
</html>
EOS);
```

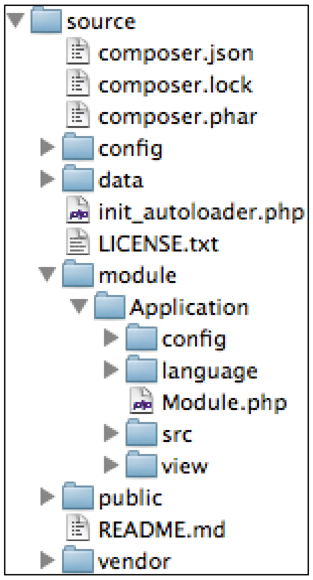
This class provides a few methods that are useful to manipulate or check the status of a response, the most important ones are as follows:

- `setStatusCode()` and `getStatusCode()`
- `isForbidden()`
- `isNotFound()`
- `isOk()`
- `isServerError()`
- `isRedirect()`

An example is worth a thousand words

Ok, now that we have the knowledge, let's dissect a simple example and see how everything is connected in the code to show the welcome page of the framework.

The folder hierarchy looks like the following screenshot:



All the code related to the welcome screen we saw in the previous chapter is located inside the application folder inside **module**. When you add more functionality to a project, it's better to separate each section and put it in its own folder. Let's see the purpose of each folder inside a module.

Name	Description
config	This folder will contain the configuration file for this specific module. You can have more than one file and also one file per environment and load them accordingly.
language	Here we will store all the translation files that affect this module. We will see how the translation works on ZF2 later in the book.
Module.php	This is the configuration file for `ModuleManager`.
src	The source code of the module will be stored here. If you navigate to the folder you'll see that there are more folders inside that separate the controllers from the models, and so on.
view	The view code will be stored here. There is one folder per controller and one file inside each folder per controller action.

The config folder

Each module can have its own config folder with configurations related to that specific module. Later, while fulfilling a request on ZF2, the configuration of all the modules will merge into one single configuration object allowing us to override configuration variables of other modules if needed. Let's take a look at the `module.config.php` file located inside the config folder.

As you can see, the first section is route configuration. Here, we should define the controller, action, and extra parameters for each URL affecting this module; the routing system should be used to fulfill the request.

```
'child_routes' => array(
    'default' => array(
        'type'    => 'Segment',
        'options' => array(
            'route'    => '/[:controller[/:action]]',
            'constraints' => array(
                'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
            ),
            'defaults' => array(
            ),
        ),
    ),
),
```

The previous code is the child route of the main. This new route acts as a wildcard and allows you to drop new controllers and actions on the module without needing to change the route. The controller and action are picked up from the URL and controlled by the regex in the constraints section.

Let's now jump to the next section of the configuration file, service manager.

The service manager configuration is an important part of the file as it allows you to specify all the components, classes, factories, and so on available on your module. In this section you can specify the following data:

Configuration key	**Description**
`abstract_factories`	This should enumerate all the classes implementing `AbstractFactoryInterface`.
`aliases`	This allows you to map names to known services.

Configuration key	Description
`factories`	Here, we should put all those factory classes implementing `FactoryInterface`, instances of classes implementing `FactoryInterface`, or PHP callbacks.
`invokables`	Classes that are instantiables on our module should be added here.
`services`	We can add services with their corresponding names here. The value should be objects.
`shared`	This is a configuration array specifying the services that are shared; by default all the services are shared.

```
'service_manager' => array(
    'factories' => array(
        'translator' =>
            'Zend\I18n\Translator\TranslatorServiceFactory',
    ),
),
```

Here, we are telling the `ServiceManager` object that it should add a new service; in this case we are going to be using the `TranslatorService` factory.

As we are using the `TranslatorService` factory we need to configure it as follows:

```
'translator' => array(
    'locale' => 'en_US',
    'translation_file_patterns' => array(
        array(
            'type'     => 'gettext',
            'base_dir' => __DIR__ . '/../language',
            'pattern'  => '%s.mo',
        ),
    ),
),
```

Translation files contain only one language per file, that's why we define a pattern for the filename of the translation files. As ZF2 supports multiple types of translation files, we should define the type in this section. In our case we will be using gettext files.

Now we move on to the next section. This one is related to the `ServiceManager` object and is where we define the controllers available and invokable, the ones that can be used by the `ServiceManager` object.

```
'controllers' => array(
    'invokables' => array(
        'Application\Controller\Index' => 'Application\Controller\
IndexController'
    ),
),
```

This is only an associative array with the name of the service as keys and path as values.

```
'view_manager' => array(
    'display_not_found_reason' => true,
    'display_exceptions'       => true,
    'doctype'                  => 'HTML5',
    'not_found_template'       => 'error/404',
    'exception_template'       => 'error/index',
    'template_map' => array(
        'layout/layout'           => __DIR__ . '/../view/layout/
layout.phtml',
        'application/index/index' => __DIR__ . '/../view/application/
index/index.phtml',
        'error/404'               => __DIR__ . '/../view/error/404.
phtml',
        'error/index'             => __DIR__ . '/../view/error/index.
phtml',
    ),
    'template_path_stack' => array(
        __DIR__ . '/../view',
    ),
),
```

The last section of the config file is related to the `ViewManager` component. Here we can configure a bunch of options; the most important ones are as follows:

- `display_exceptions`: This controls the additional information on screen about an exception
- `Doctype`: This is used to control the doctype we use on our layouts
- `not_found_template` and `exception_template`: This controls the templates used for not found and exception situations

- `template_map`: This is used to specify the real path of files used and will be pushed to a `TemplateMapResolver` object
- `template_path_stack`: This is used to specify the folders where the system must search for view files

Language files

Inside the language folder you will see a lot of files. They can be organized into two types: the files ending in .po and the files ending in .mo. We are using gettext in this example and these are actual gettext files, the compiled file ends in .mo and the readable/editable one has .po as extension. When you develop applications that are internationalized, you'll need to provide these files. In order to create them you can use desktop software such as PoEdit (`http://www.poedit.net/`) that makes life much easier and also provides the translators with an interface to manage the strings.

Module.php

This is the file that the `ModuleManager` object will read to get info about the module. In this file we define three methods. The first one called `onBootstrap` is used when `EVENT_BOOTSTRAP` is fired and will take care of loading and setting up the translator and `ModuleRoute`. As an example, ZF2 comes by default with the code to instantiate the translator, but as this is an example we can remove it if desired.

This method should be used on your module when you need to configure or do something when the application is being bootstrapped.

The `getConfig` method returns a config array of the module. This method is used by the `ModuleManager` object to read the config of the module.

Finally, the last method in this file is used to configure `AutoloaderListener` inside `ModuleManager`. Essentially we are adding the current namespace and the path where the loader can find the files.

Src folder

This is the folder that will contain the source code of the module. Inside the folder you can store any kind of code you need, but they will essentially be controllers, models, and some extra stuff such as libraries and entities. In this example there is only one controller inside the `Controller` folder. You will notice that the filename of the controller follows a convention: it's the name of the controller followed by the word Controller using camelCase.

Let's see what's inside the controller:

```
namespace Application\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        return new ViewModel();
    }
}
```

This example is pretty basic. There is not much code on the controller because everything is static and stored in the `view` file. ZF2 is taking advantage of the namespaces introduced in PHP 5.3, so the first thing you should do is declare the namespace for the current file. After that we declare the components we are going to use on the controller; in this example as we are extending `AbstractActionController`, so we need to tell PHP that we want to use that specific component. As this controller is tied to a view we should also load the `ViewModel` class.

The namespaces were created to solve two main issues. The first one is name collision between the code you create and constants, functions, or classes on third-party libraries. The second reason was to allow the developer to create an alias of long names consequent of the solution to the first problem.

View folder

The `view` folder has a few folders inside; we can have as many as we want. In our case, we have the application folder that contains all the views related to the code. If you take a closer look you will see that there is a folder with the same name as the controller and a file with the same name as the action. This is a good convention to have everything organized and you should follow it as well. Then we have an error folder containing the views used in error situations, such as a 404 not found or an exception in the execution of the program. Finally, the last folder we have is layout. There we have the layout for our module; we use it everywhere unless we override it in our own module. This file is used with every controller/action and is merged with the view from the controller.

Summary

Good job! Right now you should have a clear picture about how a request is handled in ZF2, how the components are involved in the process, and how they are used to complete the request. These components are the basis of ZF2, so it's really important that you understand how to use them and how to interact with the data using these components.

We discovered how to bootstrap the application and the options we have to customize it. We have also learned how the request object holds the data and how to access them. The chapter also showed how the router will use the request data and the set of routes we configure to determine which controller and action should be executed. Also, we saw how the dispatch process is executed and how the actual controller/action gets called. In the meantime we explored the different solutions and helpers ZF2 provides for some common problems and how they help us to solve them. We finally saw the last step of the request, the response.

Now with all this knowledge, we have a solid base on which to start developing all the components of our social network step-by-step. We will be developing the API side using `AbstractRestfulController` as a base to build a web service and the `AbstractActionController` class for the web client. We are going to be developing the two sides at the same time to be able to see the progress in the browser and be able to interact with the forms and the different features we develop.

4
The First Request/Response – Building the User Wall

In this chapter, we will focus on the wall of the user. We will cover the changes we should make on the API level to provide the wall data as well as the code we need on the client side to display the user wall. As this is the first section of the social network we implement, we will also cover the basic access to the database. We will assume for now that the data is already in the database; we don't care how the data arrived there because our only job here is to show that data on the user wall. This means that the API will provide, for now, just the access to the data and the frontend will show the data. It is really simple and a good exercise to check how the frontend and API integrate and work together.

By the end of this chapter, you will know how the frontend and API work together to display a page on the social network. Also, you will understand how we manage to connect to the database and get the data we need.

API development

As we are working with the API-centric architecture, we should first develop the API code in order to provide data to the clients. This is the general approach that we will follow through the book to develop any section of the social network.

In the previous chapter, we saw that ZF2 provides two controller types we can use. The first one called `AbstractActionController` will be used on the client side to develop the client itself. The second type, `AbstractRestfulController`, is the one we will use on the API side to build a RESTful web service to provide access to the data and functionalities.

Requirements

For the API level of the wall, we need to use a new endpoint. In this case the API will expose the data using the URI, `/wall/:id`. Let's see a more formal description of this end-point and the functionality provided.

HTTP method	Controller method	Parameters	Functionality

`GET`	`get()`
	`getList()`
		id	This is the only method providing data to the world. The id parameter is mandatory and should contain the username. This method will return all the info related to the user wall.
			As the id is mandatory, `getList()` will return an HTTP error 405 Method Not Allowed.

`POST`	`create()`	$data	This method is not allowed.
`PUT`	`update()`	id	This method is not allowed.
`DELETE`	`delete()`	id	This method is not allowed.

As you can see, `get()` is the only method allowed at this endpoint. This method will return the data related to the profile as a JSON encoded string. In case the user is not found, this method will return an HTTP 404 error.

The API will always return a JSON encoded string as a result of the requests even if we are dealing with errors; this means that the API code will not have a view per se.

Working with the database

The first task we have to complete is the database work. We will need to create a table to store the data of the user. In order to create the table on the virtual machine, we need to connect to the database server and then create the table in the database. The first task is really easy because the MySQL port is open to the host machine.

Open your favorite SQL editor/manager and connect to the server using the IP of the VM. If you haven't made changes to the IP on the `Vagrantfile`, it should be `192.168.56.2`. As a username you should use `root` and as the password you must use the one specified in `Vagrantfile`. If you haven't changed it, it should also be root.

If you prefer to use the command-line interface to connect to the MySQL server, it will be like the following command line:

```
mysql -uroot -proot -h192.168.56.2
```

Once you connect, you should be able to see a database named `sn`.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sn                 |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

Now let's create a table named `users` inside the `sn` database. This table will contain the following fields:

- Id
- Username
- Password
- Email
- Avatar_id
- Name
- Surname
- Bio
- Location
- Gender

The create table statement is as follows:

```
CREATE TABLE `users` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT NULL,
  `email` varchar(254) DEFAULT NULL,
  `password` binary(60) DEFAULT NULL,
  `avatar_id` int(11) unsigned DEFAULT NULL,
  `name` varchar(25) DEFAULT NULL,
  `surname` varchar(50) DEFAULT NULL,
  `bio` tinytext,
  `location` tinytext,
  `gender` tinyint(1) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_username` (`username`),
  KEY `idx_email` (`email`(191))
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

As you can see, `gender` is a `tinyint` value; in this case we need a convention to express male and female. We will use `1` for male and `0` for female.

Let's take an overview on how ZF2 manages to connect and retrieve data from a database.

There are three main components involved with databases: `Zend\Db\Adapter`, `Zend\Db\TableGateway`, and `Zend\Db\RowGateway`.

Zend\Db\Adapter

This is the most important subcomponent of `Zend\Db` collection of objects. This object translates the queries we create on the code to vendor- and platform-specific code and takes care of all the differences between them. It creates an abstraction layer in the databases and provides a common interface.

When creating an instance of this object, we should provide minimum information such as the driver to use, the database name to connect, and username and password for the connection itself. After that we can use this object to query the database directly but, as mentioned, ZF2 provides two more objects that will ease the process.

Zend\Db\TableGateway

This object is a representation of a table on the database and provides the usual operations we would like to do over a table such as, `select()`, `insert()`, `delete()`, or `update()`. This part of the library provides two implementations of `TableGatewayInterface`: `AbstractTableGateway` and `TableGateway`. Two important things we should know about this component is that we can configure it to return each row in a `RowGateway` object or provide a class that will act as an entity and will store the row data.

The other interesting functionality we should be aware of is the Features API. This internal API allows us to extend the functionality of the `TableGateway` object without the need to extend the class and customize the behavior to meet our needs. As usual ZF2 provides a few built-in features we can take advantage of:

- `GlobalAdapterFeature`: This feature allows us to use a global adapter without the need to inject it into the `TableGateway` object.
- `MasterSlaveFeature`: This feature modifies the behavior of `TableGateway` to use a specific master adapter for inserts, updates, and deletions and a slave adapter for select operations.
- `MetadataFeature`: This feature gives us the ability to feed `TableGateway` with the column information provided by a `Metadata` object.
- `EventFeature`: As you can imagine, this allows us to use events in `TableGateway` and subscribe to events of the `TableGateway` lifecycle.
- `RowGatewayFeature`: This modifies the behavior of `TableGateway` when executing select statements and forces the object to return a `ResultSet` object. When iterating the `ResultSet` object, we will be working with the `RowGateway` objects.

Zend\Db\RowGateway

Following the concept of `TableGateway`, `RowGateway` represents a row inside the table of the database. This object also provides some methods related to the actions we usually make over a row, such as `save()` and `delete()`. As we mentioned before if you use `RowGatewayFeature` over `TableGateway`, the result of the select queries will be a `ResultSet` object containing instances of this object. Each object will contain the data of the row it represents and we will be able to manipulate it, change it, and save it back to the database, everything using the same object.

Setting up the connection

Now we have a clear understanding about each object involved in the database connection, let's set up the code needed to actually connect to the database.

The first step is providing the information needed by `Zend\Db\Adapter`. We will do this in two different config files: `global.php` and `local.php`. You can find them in the `config` folder; the first one will be there and the second one should be created manually. The reason why we use two files is because `local.php` will be ignored by git as stated in the `.gitignore` file.

You can see in the folder that this allows us to put the general configuration that will be committed on the `global.php` file, and keep the private data such as usernames and passwords on the `local.php` file that will not be committed.

```
return array(
    'db' => array(
        'driver' => 'Pdo',
        'dsn' => 'mysql:dbname=sn;host=localhost',
        'driver_options' => array(
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
    ),
    'service_manager' => array(
        'factories' => array(
            'Zend\Db\Adapter\Adapter' => 'Zend\Db\Adapter\
AdapterServiceFactory',
        ),
    ),
);
```

This is the config data you should put in the `global.php` file. Here, we are specifying a new entry called `db` and inside that we define the options: `driver`, `dsn`, and `driver_options`. As you can see, all the data is pretty straightforward.

> When connecting to databases, ZF2 is able to use different drivers; the options available are `Mysqli`, `Sqlsrv`, `Pdo_Sqlite`, `Pdo_Mysql`, and `Pdo`. If you want to know more about the options available to connect to databases, you can check out the documentation at `http://framework.zend.com/manual/2.0/en/modules/zend.db.adapter.html`.

The second block of config is related to `ServiceManager`. We are registering a new service called `Zend\Db\Adapter\Adapter` and we are pointing it to `AdapterServiceFactory` of the `Db` package.

Let's see the data we should add on the `local.php` file. Keep in mind that our VM is running with the `skip-grant-tables` options so these credentials will not take effect but it's a good practice to define the data on the config file.

```
return array(
    'db' => array(
        'username' => 'root',
        'password' => 'root',
    ),
);
```

As you can notice we are extending the `db` config section and adding the username and password for the `db` connection.

ViewManager configuration

The API response will always be text. That's the reason we are going to encode every response as JSON. There are a few ways to accomplish this task and we will see the most straightforward and easy. The `View` component in Zend is created with flexibility in mind and provides a way to modify how the view works using strategies.

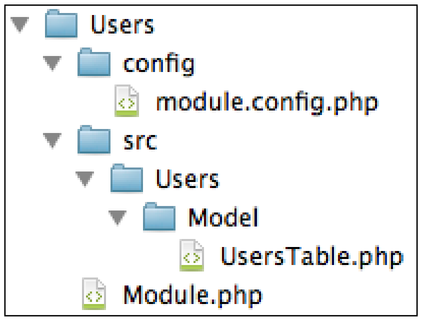
For our case, ZF2 provides us with a strategy called `ViewJsonStrategy`. Basically this component attaches two methods to the `EVENT_RENDERER` and `EVENT_RESPONSE` events of the view and modifies the way the view works based on the data we send to the view. To make `ViewJsonStrategy` work, we should return the `JsonModel` objects on our controllers. The `JsonModel` objects will be picked up by `ViewJsonStrategyand`. We will take care of setting up the correct `Content-Type` header. After that the component will use the provided `JsonModel` object to get a serialized version of the data to be sent as a response.

To activate this strategy and start returning JSON data, we only have to add the following four lines to the `global.php` file:

```
'view_manager' => array(
    'strategies' => array(
        'ViewJsonStrategy',
    ),
),
```

Creating the Users module

This will be the first module we create and is only going to contain the `TableGateway` object for the table that we just created on the database. Before starting to write the code, we need to create a new module with the following structure:



```
Users
  config
    module.config.php
  src
    Users
      Model
        UsersTable.php
  Module.php
```

Creating the UsersTable.php file

Now that we have the folder structure, the next step is to create a `TableGateway` object in order to be able to fetch the data from the database. Let's start by creating a file called `UsersTable.php` inside the `Model` folder and put the following code inside:

```
<?php
namespace Users\Model;

use Zend\Db\Adapter\Adapter;
use Zend\Db\TableGateway\AbstractTableGateway;
use Zend\Db\Adapter\AdapterAwareInterface;

class UsersTable extends AbstractTableGateway implements
AdapterAwareInterface
{
    protected $table = 'users';

    public function setDbAdapter(Adapter $adapter)
    {
        $this->adapter = $adapter;
        $this->initialize();
    }

    public function getByUsername($username)
    {
        $rowset = $this->select(array('username' => $username));

        return $rowset->current();
    }
}
```

As this is the first `TableGateway` we created, let's have a closer look at it. First thing we should do is declare the namespace where this class resides. After that we have to declare the components we are going to use inside the class using the `use` keyword.

The class is extending `AbstractTableGateway` and implementing the interface, `AdapterAwareInterface`. This interface allows the dependency injector to set the adapter we have to use for the connections to the database. As we are extending `AbtractTableGateway`, we have to define the name of the table we are representing; that's the job we do when we declare the protected variable, `$table`. After that we arrive at the first method. This is just the implementation of the method stated in the interface. In this case we store the adapter on the local variable, and then we call the `initialize()` method to set up the wiring on the object.

Finally, we define a custom method that we will use to get info from the table. We need to be able to retrieve users by their username and that's the job of the last method on this class. `getByUsername()` uses the internal `select()` method by passing an array as an argument. This array basically is the where condition for the select statement and in this case we are just filtering by the `username` column. As we are fetching the info of one user, instead of returning Rowset we call the `current()` method to return the first result. In theory we are going to have only one user with a specific username because we defined a unique index on the `username` column.

Module configuration

Let's now have a look at the contents of the configuration file.

```
return array(
    'di' => array(
        'services' => array(
            'Users\Model\UsersTable' => 'Users\Model\UsersTable'
        )
    ),
);
```

As we are just storing the `TableGateway` object, the only entry on the config file is to define the availability of this object on the dependency injector.

> Dependency injection is a software design pattern that allows the developer to remove the hard-coded dependencies and makes it possible to swap them with different ones at runtime.

The module.php file

As we have seen before in other classes, we have to define the current namespace and import the classes we are going to use; in this case the code is as follows:

```
namespace Users;
```

After that we define a class called `Module` and then we jump to the first method, `getConfig()` used by `ModuleManager` to retrieve the config file of the module. The method looks like the following code snippet:

```
public function getConfig()
{
    return include __DIR__ . '/config/module.config.php';
}
```

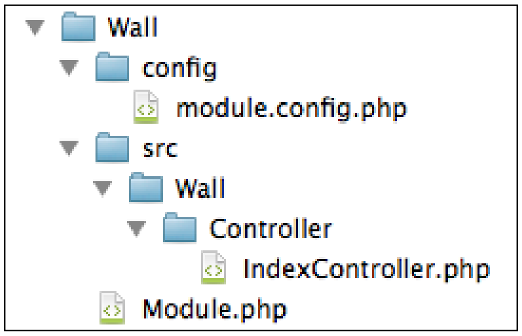
As you can see we are just returning the array contained in the configuration file for the module. We will later explore the contents of this file.

We already saw that ZF2 is PSR-0 compliant and has its own auto-loading mechanism using the `Zend\Loader` components. The last method we declare on the `Module` class is called `getAutoloaderConfig()` and it's used to tell the autoloader where it can find the files for this module.

```
public function getAutoloaderConfig()
{
    return array(
        'Zend\Loader\StandardAutoloader' => array(
            'namespaces' => array(
                __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
            ),
        ),
    );
}
```

Adding the Wall module

Our next task is to create the module that will contain the code related to the user wall. Take a look at the following folder structure and recreate it inside the module folder:



```
Wall
    config
        module.config.php
    src
        Wall
            Controller
                IndexController.php
    Module.php
```

Let's define the contents of the `Module.php` file that should be placed at the root level of the `Wall` module folder. Remember that this file is the configuration class used by `ModuleManager` to get extra information about our modules.

Module.php contents

As we saw before in the `Users` module, we have to define the current namespace and import the classes we are going to use. In this case, the code looks exactly the same as the `Users/Module.php` file and the only difference is the definition of the current namespace.

```
namespace Wall;
```

Module configuration

Let's now review the contents of the configuration file. The first section is related to the router and the routes we are listening to. We need to define a route to access the data and the controller/action that will fulfill the request. Let's see the following code snippet to know how it's configured:

```
'router' => array(
    'routes' => array(
        'wall' => array(
            'type' => 'Zend\Mvc\Router\Http\Segment',
            'options' => array(
                'route'    => '/api/wall[/:id]',
                'constraints' => array(
                    'id' => '\w+'
                ),
                'defaults' => array(
                    'controller' => 'Wall\Controller\Index'
                ),
            ),
        ),
    ),
),
```

As you can see, this way of configuring the routes is very verbose, but it's also very convenient. We are defining a new route identified by the name, wall, of type `Zend\Mvc\Router\Http\Segment` and we define the structure of the route with an optional parameter, the `id` of the user, which in our case will be the username. We also set up the constraint about what the `id` should look like. Finally, in the defaults section, we define the controller in charge for requests sent to this URL.

Let's now dig in the last section of the configuration file that basically specifies the controllers available on this module.

```
'controllers' => array(
    'invokables' => array(
        'Wall\Controller\Index' =>
```

```
            'Wall\Controller\IndexController'
        ),
    )
```

This section is as simple as defining an identifier for the controller and the class of the file containing the controller.

Adding the IndexController.php file

We have arrived at the last file of the API, the controller. Here, we put everything together and we fulfill the request with the data the clients need. As we said before, the controllers on the API side are based on `AbstractRestfulController`. That means the action we will execute to fulfill the request is determined based on the type of requests.

The first thing is to define the namespace where this file resides, and then the components we will use in the class.

```
    namespace Wall\Controller;

    use Zend\Mvc\Controller\AbstractRestfulController;
    use Zend\View\Model\JsonModel;
```

Then we proceed to create the class itself extending `AbstractRestfulController`. The class will be called `IndexController`.

In this controller, we have to accomplish two things: the first one is to implement the abstract methods of the parent class and the second one is to have access to the `UsersTable` object. Let's see first how we get access to the `UsersTable` object.

```
    protected $usersTable;

    protected function getUsersTable()
    {
        if (!$this->usersTable) {
            $sm = $this->getServiceLocator();
            $this->usersTable = $sm->get(Users\Model\UsersTable');
        }
        return $this->usersTable;
    }
```

We define a new protected variable inside the class that will hold the `UsersTable` object. After that we create a new method called `getUsersTable()`, which will get the `UsersTable` object from `ServiceManager` and will store it in the internal variable.

Now let's implement the methods of the parent class. We have a bunch of them as we defined in the requirements. As we said, this controller will only respond to the `get()` method, the rest will not be used. So in that case we should follow the HTTP specification and issue a 405 code. Let's see how we deal with the methods that we don't use and then also see how we implement the `get()` method.

```
public function getList()
{
    $this->methodNotAllowed();
}

public function create($data)
{
    $this->methodNotAllowed();
}

public function update($id, $data)
{
    $this->methodNotAllowed();
}

public function delete($id)
{
    $this->methodNotAllowed();
}

protected function methodNotAllowed()
{
    $this->response->setStatusCode(
        \Zend\Http\PhpEnvironment\Response::STATUS_CODE_405
    );
}
```

As you see the unused methods call the `methodNotAllowed()` method that will set the status code of the response to 405.

Let's see now how we tackle the `get()` method.

```
public function get($username)
{
    $usersTable = $this->getUsersTable();
    $userData = $usersTable->getByUsername($username);

    if ($userData !== false) {
        return new JsonModel($userData->getArrayCopy());
    } else {
        throw new \Exception('User not found', 404);
    }
}
```

The first thing we do is retrieve the `UsersTable` object. After that, we call the `getByUsername()` method we created before. If we have data, it means that the user exists so we proceed to return a `JsonModel` object containing a copy of the data. In case the user is not in our database, we throw an exception that will be caught by a listener, which we will implement in the next section.

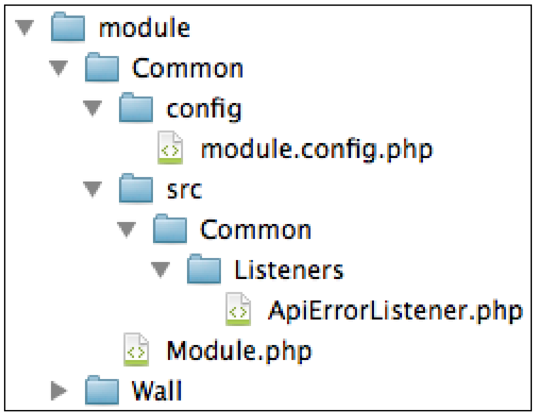
The API-Error approach

Usually when we develop APIs, we use the HTTP status codes to report errors that we encounter while processing the request. But this is not enough in most cases and we end up creating our own objects to represent an error status on the API level. To solve this issue and standardize the way we respond to errors on API based on JSON or XML, two proposals are gaining track and they use the error media types, such as `application/api-problem+json` and `application/vnd.error+json`.

In this section we will build the foundation of this approach but we will cover it in more depth and implement it properly later to avoid overloading the chapter with a lot of new stuff.

Adding the Common module

We will implement the API-Error listener in a new module to have the code separated by responsibility. Let's first create the folder structure. Then we will jump to the code we should add and how it works.



```
module
  Common
    config
      module.config.php
    src
      Common
        Listeners
          ApiErrorListener.php
    Module.php
  Wall
```

Modifying the application.config.php file

Now, the first thing we should do is add the new modules to the `application.config.php` file so the framework will know that they are available. In order to do that we need to modify the modules configuration like the following code snippet:

```
'modules' => array(
    'Wall',
    'Users',
    'Common'
),
```

As you can see we just add as many entries as we have modules and they are called after the module name.

Modifying the Module.php file

We have already seen that all the modules must have a file called `Module.php`. As we discussed earlier, this file is used to get extra configuration and information about the module and it's used by `ModuleManager`. For this module, we will add the usual code in our `Module.php` to return the config file and configure the autoloader. Also, we will initialize our listener when the module is being bootstrapped.

```
public function getConfig()
{
    return include __DIR__ . '/config/module.config.php';
}

public function getAutoloaderConfig()
{
    return array(
        'Zend\Loader\StandardAutoloader' => array(
            'namespaces' => array(
                __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__
            ),
        ),
    );
}

public function onBootstrap($e)
{
    $app = $e->getTarget();
    $services = $app->getServiceManager();
    $events = $app->getEventManager();
```

```
        $events->attach(
            $services->get('Common\Listeners\Listener')
        );
    }
```

The `onBootstrap()` method is called when `ModuleManager` is bootstrapping the module. We use this to configure the listener, adding it to `EventManager`.

Modifying the module.config.php file

Now that we have the first bits of our module and the manager knows what to do with it, let's review the contents of the `module.config.php` file. As you will see it is really simple.

```
return array(
    'service_manager' => array(
        'invokables' => array(
            'Common\Listeners\ApiProblemListener' =>
            'Common\Listeners\ApiProblemListener',
        ),
    ),
);
```

In this code, we are adding a new object as invokable to the `service_manager` array. So, we will be able to retrieve it later.

Adding the ApiErrorListener.php file

Finally, let's have a look at the listener that will help us spot issues on the API and will modulate the standard response for API-Problem in the future.

As usual, the first thing to do is declare the namespace and the components we are going to use.

```
namespace Common\Listeners;

use Zend\EventManager\AbstractListenerAggregate;
use Zend\EventManager\EventManagerInterface;
use Zend\Mvc\MvcEvent;
use Zend\View\Model\JsonModel;
```

Right after that we will define a new class called `ApiErrorListener` and we will make this class extend the `AbstractListenerAggregate` class, which will allow the class to attach one or more listeners.

```
class ApiErrorListener implements AbstractListenerAggregate
```

Now we should track the events we are listening to in order to accomplish our need to add a property to the class to store them. Then we should define the `attach()` method as required by the interface, `ListenerAggregateInterface`. We have to attach our methods to the events we are interested in by calling the `attach()` method of `EventManager` and passing the event name, the method to call, and the priority as parameters.

```
public function attach(EventManagerInterface $events)
{
    $this->listeners[] = $events->attach(
        MvcEvent::EVENT_RENDER,
        'ApiErrorListener::onRender',
        1000
    );
}
```

Finally, let's see the code that inspects the response and determines if there is an error and format it.

```
public static function onRender(MvcEvent $e)
{
    if ($e->getResponse()->isOk()) {
        return;
    }

    $httpCode = $e->getResponse()->getStatusCode();
    $sm = $e->getApplication()->getServiceManager();
    $viewModel = $e->getResult();
    $exception = $viewModel->getVariable('exception');

    $model = new JsonModel(array(
        'errorCode' => $exception->getCode() ?: $httpCode,
        'errorMsg' => $exception->getMessage()
    ));
    $model->setTerminal(true);

    $e->setResult($model);
    $e->setViewModel($model);
    $e->getResponse()->setStatusCode($httpCode);
}
```

Let us go line by line in this big chunk of code. At the top we check if the response is marked as 200 OK. If it is, we don't have to continue worrying about errors; otherwise, we retrieve the status code and the exception variable from `ViewModel`. We are assuming that if something goes wrong, an exception will be thrown. That's why on our controller if the user is not in the database, we throw an exception because it will be caught here.

With the data we extracted, we create a new `JsonModel` object specifying `errorCode` based on the code used in the exception; otherwise, we use the HTTP error and we also specify `errorMsg` based on the text of the exception.

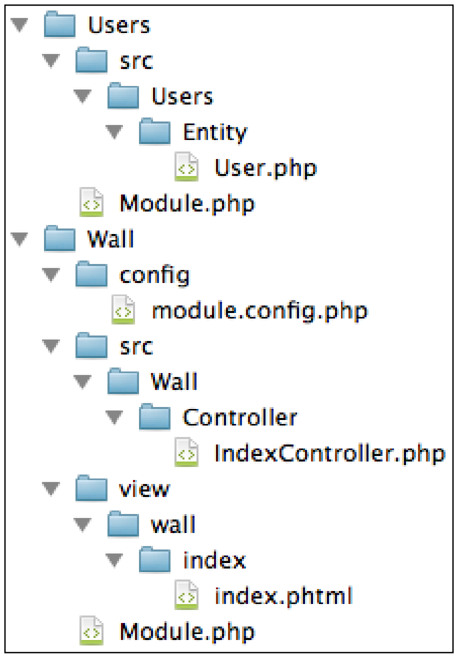
Then we use the `setTerminal()` method of the `JsonModel` object to set the terminal flag to `true` to tell ZF2 to not wrap the returned model in a layout.

The last section of the method takes care of attaching the new `JsonModel` object to the response overwriting the previous one and setting the corresponding status code.

Frontend

Nice! If you have arrived here that means that we have a shiny API to retrieve info of the user wall. Now that we have the data on a JSON object, we have to create a beautiful HTML client to show the information on a browser.

To build this section of the social network we need to create additional modules that will contain all the code related to the user wall, the user data, and the API access. At this point, as we have an API, we need some sort of a client in order to get the data. The `Api` module does this job but the contents of it are still complex for us and we will see how the concepts work later on. For now just trust me and use the code. When we reach *Chapter 7*, *Dealing with URLs – Posting Links*, we will see how to make requests to external URLs and then you can review the contents of the `Api` module and you'll understand it.



```
Users
  src
    Users
      Entity
        User.php
  Module.php
Wall
  config
    module.config.php
  src
    Wall
      Controller
        IndexController.php
  view
    wall
      index
        index.phtml
  Module.php
```

This is what the `Users` module and the `Wall` module looks like. Create them and let's move on and see what to do with them.

The `Wall` module is the one responsible for getting the data and showing it to the user. This module uses the entity located in the `Users` module to store the data coming from the API and at the same time utilizes the `Api` module to get the data from the backend. Let's have a quick look at what the `User` entity is and then we will see the `Wall` module itself.

Adding the Users module

Right now this module is really simple and just contains the `User` entity that will store the data passed from the API. Of course this module will be extended later on to add more functionality.

Contents of the Module.php file

```
namespace Users;

class Module
{
    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __
NAMESPACE__,
                ),
            ),
        );
    }
}
```

These are the contents of the file. As you can see it's pretty simple and we are just adding the module to the autoloader.

Creating the User.php file

```
namespace Users\Entity;

class User
{
    const GENDER_MALE = 1;

    protected $id;
    protected $username;
    protected $name;
    protected $surname;
    protected $avatar;
    protected $bio;
    protected $location;
    protected $gender;
    protected $createdAt = null;
    protected $updatedAt = null;

[...]
```

This is the beginning of the file. Here, we are defining the namespace where this class lives, the constant that represents the value of the male gender in the database, and then a bunch of protected properties, one for each column in the database. After this the file just contains getters and setters for the properties we just saw.

Requesting the wall content

Now it's time to see how to use the `Api` module to get the contents of the wall and show them to the user.

Modifying the Module.php file

As we did before with the module on the API side of the project, we should create a new `Module.php` file for this module. The contents of the file are pretty standard and now you should be able to identify what each method does.

```
namespace Wall;

use Zend\Mvc\ModuleRouteListener;
use Zend\Mvc\MvcEvent;

class Module
{
    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php';
    }

    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ =>
                    __DIR__ . '/src/' . __NAMESPACE__,
                ),
            ),
        );
    }
}
```

Module configuration

Let's see the contents of the configuration file. In this case we only have three sections. Two of them will be familiar and the third one is a completely new one.

```
'router' => array(
    'routes' => array(
        'wall' => array(
            'type' => 'Zend\Mvc\Router\Http\Segment',
            'options' => array(
                'route'    => '/:username',
                'constraints' => array(
                    'username' => '\w+'
                ),
                'defaults' => array(
                    'controller' => 'Wall\Controller\Index',
                    'action'     => 'index',
                ),
            ),
        ),
    ),
),
```

The first one is the route definition. As you can see, we will have a parameter on the URL that will be the username of the user we want to see. The route is also a `Zend\Mvc\Router\Http\Segment` type as we saw in the API section. We also define the constraints of the parameter, but we have a subtle difference with the route defined in the API. If you take a closer look at the defaults section, we not only define the controller but also the action that should be called. This is because the controller in the frontend doesn't extend the controller, `AbstractRestfulController`. Instead, we extend `AbstractActionController`.

Then, we define the controllers available; this section is really simple as we only have one controller.

```
'controllers' => array(
    'invokables' => array(
        'Wall\Controller\Index' =>
        'Wall\Controller\IndexController'
    ),
),
```

Now we arrive at the new section called `view_manager` and take care of the configuration of the view object. As this is the client, we will have a proper view outputting HTML code to the browser.

```
'view_manager' => array(
    'display_not_found_reason' => true,
    'display_exceptions'       => true,
    'doctype'                  => 'HTML5',
    'not_found_template'       => 'error/404',
    'exception_template'       => 'error/index',
    'template_path_stack' => array(
        __DIR__ . '/../view',
    ),
),
```

We are defining some variables here that control the information we show when an action is not found or an exception is thrown. We also define `doctype` of the HTML code. You can do it here or you can hardcode `doctype` in the layout. Finally, we define two templates for specific situations and the path where the templates reside. As we are not defining a layout file on this module, the view will fall back to the one located in the Application module provided by ZF2 by default.

Changes in the IndexController.php file

Let's see how we connect the client to the API. This is accomplished on the `IndexController.php` file.

```
namespace Wall\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\Stdlib\Hydrator\ClassMethods;
use Wall\Entity\User;
use Api\Client\ApiClient as ApiClient;

class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        $viewData = array();

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

```
            $response = ApiClient::getWall($username);

            if ($response !== false) {
                $hydrator = new ClassMethods();

                $user = $hydrator->hydrate($response, new User());
            } else {
                $this->getResponse()->setStatusCode(404);
                return;
            }

            $viewData['profileData'] = $user;

            return $viewData;
        }
    }
```

The first chunk of code declares the namespace the controller is living on and then we specify the components we are going to use. As you can see, the controller, `IndexController` extends `AbstractActionController`. The only method we have is the `indexAction()` method and it's the one in charge of the wall requests. If you take a closer look, you can see that we retrieve the username from the URL. After that we pass it to the layout, and then we use `ApiClient` to get the data of the wall by calling `getWall()`.

After that, we examine the response to see if it's successful or not; this can be improved but for now it's enough.

If the response is correct, we decode it; remember that the communication between the frontend and the API is done with the JSON objects. Once the data is decoded, we use a `Hydrator` to store the data inside a custom entity.

The `Hydrator` is a component from the `Stdlib` section of ZF2. This type of object is used to populate objects with data. In this case we are using a hydrator based on class methods; this means that when the hydrator will try to populate, an object will try to use setter functions inside the object.

After the entity has been populated, we return an array containing the user object to the view.

Outputting the user wall

Let's now talk about the view itself. The views on ZF2 are based on layout and the content. The content is a view by itself and can contain PHP code, use helpers, partials, and so on. Usually we have a default layout for the application and we merge that with the HTML generated by a view. The views are tied to controllers and actions following a convention. Inside the `view` folder, the framework will search for a folder named as the controller. Inside that folder each action will have a view and will be named before the action name. If you review the picture of the folder structure again, you'll see that convention in place.

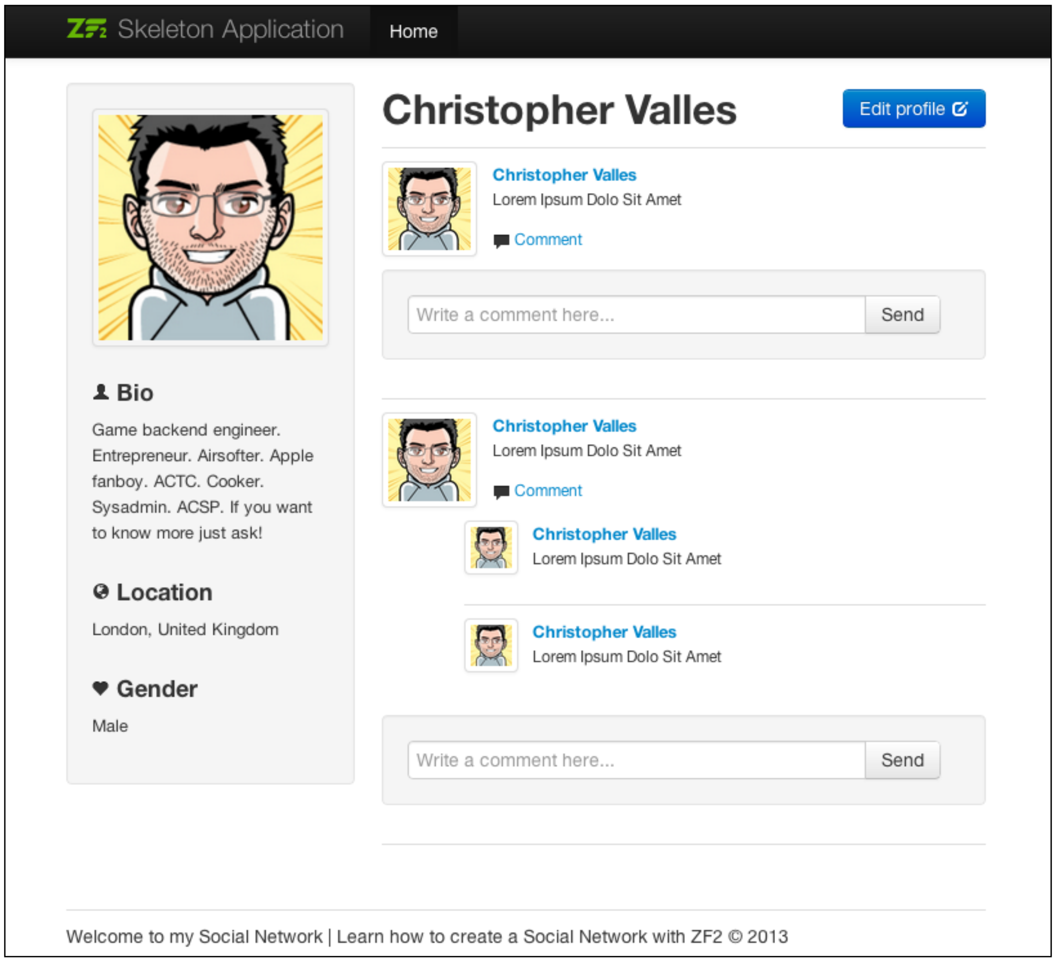
If you want, you can use a template engine in order to use a different syntax in the views instead of PHP but consider that at the end what a template engine does is convert the special syntax to plain PHP. It will probably add a compile step in the middle to translate the views to the final file. As ZF2 already provides good support for the view using plain PHP, helpers, partials, and so on, we are not going to deal with how to use a template engine. But if you want, take a look at Twig at `http://twig.sensiolabs.org` or Smarty available here `http://www.smarty.net`.

We are not going to put the code of the view here because it's huge but we are going to comment on how we print the info coming from the `User` object. If you want to take a look at the code, just download the code that comes with this book and check it out.

```
<?php echo $profileData->getLocation() ?>
<?php echo $profileData->getGenderString() ?>
<?php printf(
    '%s %s',
    $profileData->getName(),
    $profileData->getSurname()
) ?>
```

These are a few examples on how we put the info from the `User` object in the view. As you can see we are just using the getters of the class. The second example is using the helper function we made to get the gender in a string representation.

To be able to test this functionality, we only need to insert data into the table, launch the browser, and point to our client hostname adding the username in the URL. You should be able to see something like the following screenshot:



Christopher Valles

Christopher Valles
Lorem Ipsum Dolo Sit Amet

Christopher Valles
Lorem Ipsum Dolo Sit Amet

Christopher Valles
Lorem Ipsum Dolo Sit Amet

Christopher Valles
Lorem Ipsum Dolo Sit Amet

Bio

Game backend engineer. Entrepreneur. Airsofter. Apple fanboy. ACTC. Cooker. Sysadmin. ACSP. If you want to know more just ask!

Location

London, United Kingdom

Gender

Male

Welcome to my Social Network | Learn how to create a Social Network with ZF2 © 2013

Summary

Wow! It was a long trip, congratulations!

In this chapter we covered all the basics on both the API and frontend sides.

We saw the components involved in the database access and the possibilities ZF2 gives us, how to work with the database, and how to retrieve information based on the parameters we have. We saw how to create a module from scratch and how to configure it, create the routes, and process the information to output it as JSON or HTML. We also covered how to encode and decode JSON data and how to make a simple request to our API. We also worked with the two types of controllers provided by ZF2: `AbstractRestfulController` and `AbtractActionController`. Finally, we saw how to populate entities using hydrators.

Now we have the base from which to continue working on the social network. We will keep improving its functionality and the quality of the code performing some refactors along the way to keep it simple and maintainable. The next chapters will cover different aspects of the framework while adding more code, more functionality, and more fun!

5
Handling Text Content – Posting Text

In this chapter, we will start by adding features to the user wall. The user will now be able to post text on their wall as a status update, and will also be able to browse through the statuses, added previously. As usual, we will cover both the API and frontend sides. As we implement the first form, we will discover how to do the basic validation and filtering of the data prior to sending it to the API, and also how to validate that on the API side.

By the end of this chapter, you will learn how to create simple forms with a couple of fields, how to add validation to be sure that the user has provided the correct data, and how to filter it to be sure that the information we store is safe and doesn't contain any malicious code that can compromise the platform.

API development

Let's see the tasks we have to accomplish on the API level. We will update the `get()` method that returns the wall data to include all the statuses of the user, and we will also add a new method called `create()` to be able to insert new status into the database for a specific user. In order to do that we need to add a new table in the database and the corresponding table gateway. After that we'll add a new route for the `create()` method. Finally on the API side, we will see how to validate and filter the data that comes from the client.

Requirements

In order to post a new status, we will modify the endpoint we created in *Chapter 4, The First Request/Response – Building the User Wall*, to accept the `POST` requests. Let's see the formal requirements of the new method in the following table:

HTTP method	Controller method	Parameters	Functionality
`GET`	`get()`	`id`	This is the only method providing data to the world. The `id` parameter is mandatory and should contain the username. This method will return all the info related to the user wall.
	`getList()`		As the `id` parameter is mandatory, `getList()` will return an HTTP error 405.
`POST`	`create()`	`data`	This is the method used to post content to the wall. The parameter is an array of data containing `user_id` and the parameters needed to create the content.
`PUT`	`update()`	`id`	This method is not allowed.
`DELETE`	`delete()`	`id`	This method is not allowed.

The `create()` method will inspect the contents of the `$data` parameter to see if we have the information needed to create the new status on the database. In the future, as we expand this method, we will examine the data to decide the type of content we should create.

Working with the database

As mentioned before, in order to be able to see the status updates of users, we will need to store them somewhere. Let's now create a new table on the database that will tie the status data to the user itself.

This table is called `users_statuses` and will hold the following data:

- Id
- User_id
- Status
- Created at
- Updated at

The column, `user_id`, will be filled with `id` of the user located in the table, `users`.

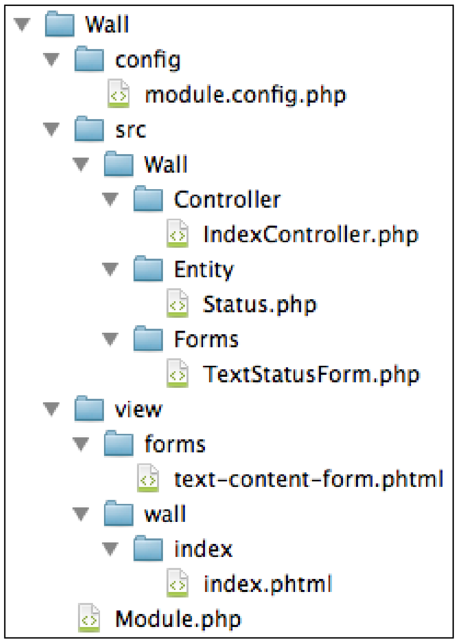
Now that we have seen the data this table will store, let's see the actual statement to create it. We need to connect to the database server through the VM as we did in *Chapter 4, The First Request/Response – Building the User Wall*, to be able to execute the following SQL statement:

```
CREATE TABLE `user_statuses` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(11) unsigned DEFAULT NULL,
  `status` text,
  `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;
```

While creating tables on a database, you have to make a choice about using foreign keys and constraints or not. Depending on the size of the tables, how they are sharded, and the database architecture, using them can be problematic but at the end it is up to you. For simplicity, we will not use them in this book.

Understanding the module structure

As we are going to add new files to the structure we already have, let's see how the structure looks with the new files added to the project in the following screenshot:



```
Wall
  config
    module.config.php
  src
    Wall
      Controller
        IndexController.php
      Entity
        Status.php
      Forms
        TextStatusForm.php
  view
    forms
      text-content-form.phtml
    wall
      index
        index.phtml
  Module.php
```

Creating UserStatusesTable.php

Now that we have the table created on the database, let's create the table gateway to be able to access it.

Create a new file called `UserStatusesTable.php` inside the `Model` folder of the `Users` module, and let's see what we have to add inside.

As with previous classes, the first thing we should do is declare the `namespace` for the class and the dependencies that we are going to use inside the class. In this case we have the following code:

```
namespace Users\Model;

use Zend\Db\Adapter\Adapter;
use Zend\Db\TableGateway\AbstractTableGateway;
use Zend\Db\Adapter\AdapterAwareInterface;
use Zend\Db\Sql\Expression;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Factory as InputFactory;
```

As you can see we have a lot of dependencies. You should be familiar with the first three dependencies as they were used in *Chapter 4, The First Request/Response – Building the User Wall*, when we created the `UsersTable` class. The next dependency is `Zend\Db\Sql\Expression`. This class allows us to use SQL expressions, such as `count(*)` or `NOW()`. The last two dependencies will provide us with the ability to create filters that will check the input data, and will assure that they meet the requirements we have for each parameter.

The following block of code defines the class itself, the table we are using, and the `setDbAdapter()` method required by the `AdapterAwareInterface` interface:

```
class UserStatusesTable extends AbstractTableGateway implements
AdapterAwareInterface
{
    const COMMENT_TYPE_ID = 1;

    protected $table = 'user_statuses';
    const TABLE_NAME = 'user_statuses';

    public function setDbAdapter(Adapter $adapter)
    {
        $this->adapter = $adapter;
        $this->initialize();
    }
```

As you can see this code is almost identical to the one we wrote for the `UsersTable` class. You might notice that we have defined two constants called `COMMENT_TYPE_ID` and `TABLE_NAME`. Ignore them for now, they will be handy when we arrive at *Chapter 8, Dealing with Spam – Akismet to the Rescue*.

The following section shows the `getByUserId()` method, takes care of retrieving the statuses of a user based on the `user_id` column, and orders them by the creation date stored in the `created_at` column:

```
public function getByUserId($userId)
{
    $select = $this->sql->select()
        ->where(array('user_id' => $userId))
        ->order('created_at DESC');
    return $this->selectWith($select);
}
```

In this method, as we want to order the results, we use a `Select` object retrieved by `$this->sql->select()`.We then customize it with the `where()` method and with the `order()` method. To execute a statement with a given `Select` object, we use the method, `selectWith()`, which accepts the `Select` object as a parameter.

The next method we are going to review is `create()` as shown in the following code:

```
public function create($userId, $status)
{
    return $this->insert(array(
        'user_id' => $userId,
        'status' => $status,
        'created_at' => new Expression('NOW()'),
        'updated_at' => null
    ));
}
```

As you can see, we are just building an array with the data we want to insert using the parameters passed in the method. To populate the `created_at` column, you have two options; the first one will be passing the date on the format that we use for that column on the database. The second option will be using the `NOW()` function from SQL to populate the value with the current date and time from the server. We will go with the second option, and in order to use it we should wrap the `NOW()` function inside an instance of `Zend\Db\Sql\Expression`.

The last section of the class will be the method called `getInputFilter()`. Inside this method, we will configure the input filters needed for each field. For each input, we will define the filters and validators that apply to that specific value and will be used while validating the input sent by the clients.

Zend Framework 2 (**ZF2**) provides a huge amount of filters and validators. They cover the common needs for web applications, and usually you'll have enough with the provided ones. If not, as always, you have the option to create your own filters and validators by extending the appropriate classes and implementing the corresponding methods.

You can have a look at the available filters and validators at `http://framework.zend.com/manual/2.2/en/modules/zend.filter.set.html` and `http://framework.zend.com/manual/2.2/en/modules/zend.validator.set.html`.

Filters and validators can be used in two different ways, attached to a form or standalone. But in both cases each field or value has its own configuration of filters and validators and each one has its own options.

As we are working on the API, we are going to see how to use them in the standalone approach. This is illustrated in the following code:

```
public function getInputFilter()
{
    $inputFilter = new InputFilter();
    $factory = new InputFactory();
```

The first thing we do is create an instance of the `InputFilter` component.

The `InputFilter` component can be used to filter and validate generic groups of data using associative arrays to provide the data. This component can hold multiple configurations of filters and validators, each one for a specific input or variable.

A new `Zend\InputFilter\Factory` instance is also created to be able to initialize the `InputFilter` objects from a configuration stored in an array.

Notice that we created an alias, `InputFactory`, for this object when we declared the dependency.

The following code represents the new `Zend\InputFilter\Factory` dependency:

```
    $inputFilter->add($factory->createInput(array(
        'name'     => 'user_id',
        'required' => true,
        'filters'  => array(
            array('name' => 'StripTags'),
            array('name' => 'StringTrim'),
            array('name' => 'Int'),
        ),
        'validators' => array(
            array('name' => 'NotEmpty'),
            array('name' => 'Digits'),
            array(
                'name' => 'Zend\Validator\Db\RecordExists',
                'options' => array(
                    'table' => 'users',
                    'field' => 'id',
                    'adapter' => $this->adapter
                )
            )
        ),
    )));
```

The first block of code will configure the filters and validators for a field called `user_id`. As a filter we add `StripTags`, `StringTrim`, or `Int`. The first one will remove all the HTML tags present in the value, the second one will remove whitespaces before and after the value itself, and the last one will try to convert a scalar value to an integer.

> As a good practice you should always remove any HTML tags and sanitize the data that comes from the user, in order to minimize the chances of being vulnerable to a **Cross-site scripting** (**XSS**) attack. Just remember to never trust anything that comes from the user.

In order to validate the data for this field, we need to check the following points:

- The field is not empty
- The value should be a number
- If we are adding the status of a user, the user must exist in the user's table

Take a look at the `validators` section of the configuration for the first input. The first two are pretty easy, just including the validator and the default configuration for them will do the job. For the third one, we will use a validator provided by ZF2 called `Zend\Validator\Db\RecordExists`. This validator checks if a row exists in the database based on a configuration. The configuration should define the table and column to be checked if the record exists. Finally, to be able to query the database, an adapter must be provided as shown in the following code:

```
$inputFilter->add($factory->createInput(array(
    'name'     => 'status',
    'required' => true,
    'filters'  => array(
        array('name' => 'StripTags'),
        array('name' => 'StringTrim'),
    ),
    'validators' => array(
        array('name' => 'NotEmpty'),
        array(
            'name'    => 'StringLength',
            'options' => array(
                'encoding' => 'UTF-8',
                'min'      => 1,
                'max'      => 65535,
            ),
        ),
    ),
        )));

        return $inputFilter;
    }
```

Now let's take a look at the second input where we will try to filter and validate the data. The data will be the status of the user, this means text. The filter section will be pretty standard, and we will only remove HTML tags and whitespaces before and after the data. The validator's section will contain the `NotEmpty` validator, as before, to be sure that we have the data. At the end, it makes no sense to store an empty status. After that, we will validate the length of the text. As we defined the column on the table as `TEXT` type, this will imply the minimum and maximum size of the text. With the `StringLength` validator, we define that the status should have a minimum of one character and a maximum of 65,535 characters. Also, we define the encoding of the text.

Extending IndexController.php

As we saw in the beginning of the chapter, we will need to extend the `IndexController.php` file to add a new method called `create()`. This method was returning a 404 response, because at that point the logic was not implemented. But now we will add the required code to validate the data coming from the client, access the database, store the status of the user, if the data is valid, and finally return the number of rows affected in the operation.

We need to add a new property to the controller to hold a copy of the new table gateway. This property will be used in the `getUserStatusesTable()` method to store the object and avoid creating more than one instance as shown in the following code:

```
    protected $userStatusesTable;
```

The first change we need to make in this controller is adding the status of a user when the client requests the data of the profile. The previous version of the `get()` method was as follows:

```
    public function get($username)
    {
        $usersTable = $this->getUsersTable();
        $userData = $usersTable->getByUsername($username);
        $wallData = $userData->getArrayCopy();

        if ($userData !== false) {
            return new JsonModel($wallData);
        } else {
            throw new \Exception('User not found', 404);
        }
    }
```

Now, this method will be seen as follows:

```
    public function get($username)
    {
        $usersTable = $this->getUsersTable();
        $userStatusesTable = $this->getUserStatusesTable();

        $userData = $usersTable->getByUsername($username);
        $userStatuses = $userStatusesTable->getByUserId(
            $userData->id
        )->toArray();

        $wallData = $userData->getArrayCopy();
        $wallData['feed'] = $userStatuses;

        usort($wallData['feed'], function($a, $b){
            $timestampA = strtotime($a['created_at']);
            $timestampB = strtotime($b['created_at']);

            if ($timestampA == $timestampB) {
                return 0;
            }

            return ($timestampA > $timestampB) ? -1 : 1;
        });

        if ($userData !== false) {
            return new JsonModel($wallData);
        } else {
            throw new \Exception('User not found', 404);
        }
    }
```

Let's take a closer look at the differences. We are retrieving the new table gateway to be able to access the status of a user. After retrieving the user data, we retrieve the status of the user from the database by calling the method, `getByUserId()`, and passing the ID of the user we just got from the database. Then, we create an array to hold the data of the user itself and the statuses.

Earlier, we were returning just a copy of the data of the user as an array. Now we have to compose our own array and merge the data and the statuses of the user on the same array. Finally, instead of returning just the data of the user, we return the array we created with both the user data and the statuses.

Now that we updated the `get()` method, we will be able to see the statuses of the client, or at least the client will be able to receive this data while requesting the data of a user. Now let's write the code for the `create()` method that will allow users to post new statuses on their wall, as shown in the following code:

```
public function create($data)
{
    $userStatusesTable = $this->getUserStatusesTable();

    $filters = $userStatusesTable->getInputFilter();
    $filters->setData($data);

    if ($filters->isValid()) {
        $data = $filters->getValues();

        $result = new JsonModel(array(
            'result' => $userStatusesTable->create(
                $data['user_id'], $data['status']
            )
        ));
    } else {
        $result = new JsonModel(array(
            'result' => false,
            'errors' => $filters->getMessages()
        ));
    }

    return $result;
}
```

The first thing we do is retrieve the table gateway we are going to use. After that we get the filters we need to validate the data sent by the client. The `InputFilter` component has a method called `setData()`, and this is used to provide the filters and validations with the data we want to validate. We just need to call the method passing the `$data` array, which we get as a parameter for the `create()` method. This variable will be populated automatically by `AbstractRestfulController` with the `$_POST` data. To execute the validation itself, we call the method, `isValid()`, from the `InputFilter` component that will apply to all the filters and validators on the data we passed and will determine if the data is valid or not.

Right after that, we need to decide what to do in case the data is valid or not. In the first case, we will insert the data to the database using the method, `create()`, from `UserStatusesTable`, and return the affected rows in `JsonModel`. In case the data is invalid, we will compose a new array specifying the errors encountered by the `InputFilter` component. The error messages are going to be passed to the client in another `JsonModel`.

The last bit we should add to this controller is the `getUserStatusesTable()` method that will be stored in `$userStatusesTable`, an instance of this table gateway, as shown in the following code:

```
protected function getUserStatusesTable()
{
    if (!$this->userStatusesTable) {
        $sm = $this->getServiceLocator();
        $this->userStatusesTable = $sm->get(
            'Users\Model\UserStatusesTable'
        );
    }
    return $this->userStatusesTable;
}
```

As there is no change, the method is exactly the same as `getUsersTable()`. The only difference is the name of the component we extract from `ServiceManager`.

Modifying module.config.php

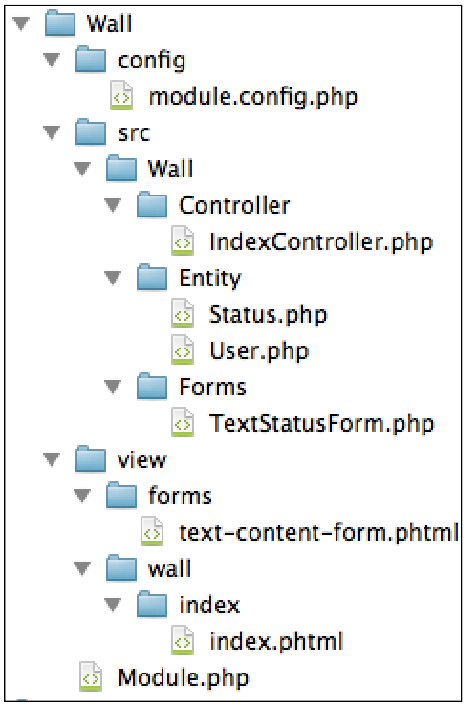
We need to update the `module.config.php` file of the `Users` module, in order to make the `UserStatusesTable` class available in the dependency injector. We just need to add the following line:

```
'Users\Model\UserStatusesTable' => 'Users\Model\UserStatusesTable'
```

Frontend

Now that we have the data coming from the API, we can modify the frontend to show the statuses of the user on the wall, and also create a form to allow the users to post new content. In this section, we will see how to include the statuses on the wall, but the important part is focused on how to create forms in ZF2, and how to attach filters and validations to forms.

Before going further, let's see the folder structure in the following screenshot with all the new files and folders we will add in this section of the chapter:



```
Wall
  config
    module.config.php
  src
    Wall
      Controller
        IndexController.php
      Entity
        Status.php
        User.php
      Forms
        TextStatusForm.php
  view
    forms
      text-content-form.phtml
    wall
      index
        index.phtml
  Module.php
```

Creating Status.php

Let's start by writing the code that will show the statuses on the wall. The first thing to do is create an entity for the status. Create a new file in the `Entity` folder inside the `src` folder of the `Wall` module and put the following code inside:

```
namespace Wall\Entity;

use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Factory as InputFactory;

class Status
{
    protected $id = null;
    protected $userId = null;
    protected $status = null;
    protected $createdAt = null;
    protected $updatedAt = null;

    public function setId($id)
```

```
        {
            $this->id = (int)$id;
        }

        public function setUserId($userId)
        {
            $this->userId = (int)$userId;
        }

        public function setStatus($status)
        {
            $this->status = $status;
        }

        public function setCreatedAt($createdAt)
        {
            $this->createdAt = new \DateTime($createdAt);
        }

        public function setUpdatedAt($updatedAt)
        {
            $this->updatedAt = new \DateTime($updatedAt);
        }

        public function getId()
        {
            return $this->id;
        }

        public function getUserId()
        {
            return $this->userId;
        }

        public function getStatus()
        {
            return $this->status;
        }

        public function getCreatedAt()
        {
            return $this->createdAt;
        }
        public function getUpdatedAt()
        {
            return $this->updatedAt;
        }

        public static function getInputFilter()
        {
            $inputFilter = new InputFilter();
            $factory = new InputFactory();

            $inputFilter->add($factory->createInput(array(
                'name'     => 'status',
                'required' => true,
                'filters'  => array(
                    array('name' => 'StripTags'),
                    array('name' => 'StringTrim'),
                ),
                'validators' => array(
                    array(
                        'name'    => 'StringLength',
                        'options' => array(
                            'encoding' => 'UTF-8',
                            'min'      => 1,
                            'max'      => 65535,
                        ),
                    ),
                ),
            )));

            return $inputFilter;
        }
    }
```

This is the full code for the class, as you can see everything comes down to a bunch of setters/getters for each property in the class. After that, we have the `getInputFilter()` method that returns a configured chain of filters and validators that will be used with the form we'll create later. Two small details while populating the `createdAt` and `updatedAt` properties. We are converting the data that comes from the API to `DateTime` PHP objects. This will allow us to use the dates in a better way instead of having plain strings. The other detail is that we are casting the numbers to integers in the `id` and `userId` properties.

Modifying User.php

The next step is to modify the `User` entity to be able to hold the related statuses.

As we want to store the `Status` object and also populate them from here, we need to declare the namespace of the object and also the `Hydrator` component we want to use, as shown in the following code:

```
use Zend\Stdlib\Hydrator\ClassMethods;
use Wall\Entity\Status;
```

We are going to add a new property called `$feed` that will contain an array of entries. For now they will be the `Statuses` object as follows:

```
protected $feed = array();
```

After this we know that while hydrating this object, the `ClassMethod` hydrator will try to call a method called `setFeed()` to store the array of entries on the wall. Let's create that method and process the data as follows:

```
public function setFeed($feed)
{
    $hydrator = new ClassMethods();

    foreach ($feed as $entry) {
        if (array_key_exists('status', $entry)) {
            $this->feed[] = $hydrator->hydrate(
                $entry, new Status()
            );
        }
    }
}
```

In this section of code, we are creating a new `ClassMethod` hydrator to populate the `Status` object based on the data we get from the API. After the object is populated, we just store it on the array we created before as a property.

The last modification on this class is a getter that will allow us to retrieve the array of statuses from the `User` object as shown in the following code:

```
public function getFeed()
{
    return $this->feed;
}
```

View

It's really hard to add the code of the view in here, because the code is usually verbose with all the HTML tags, classes, and so on. But we will now see a simplified version of the section that prints the statuses of a user. If you want to see the full code of the view, please refer to the code provided for this chapter.

```
<?php foreach ($profileData->getFeed() as $entry) : ?>
    <p><?php echo $status->getStatus() ?></p>
<?php endforeach; ?>
```

As you can see, we are iterating over the result of the `getFeed()` method, and for each element we are echoing the result of calling the `getStatus()` method. This will print a list of status text for the current user.

TextStatusForm.php

Now that we are able to see the statuses of a user, let's jump to the creation of the status. First of all we need the form to retrieve the data from the user. Forms have been completely refurbished on ZF2; they provide the best from ZF1, and also a lot of functionality and options that are completely new. As we progress in the book, we will see more complicated examples; but for now let's start with a simple and easy form to get the concept in place.

Following is the content of the file, `TextStatusForm.php`:

```
namespace Wall\Forms;

use Zend\Form\Element;
use Zend\Form\Form;

class TextStatusForm extends Form
{
    public function __construct($name = null)
    {
        parent::__construct('text-content');

        $this->setAttribute('method', 'post');
        $this->setAttribute('class', 'well input-append');

        $this->add(array(
            'name' => 'status',
            'type'  => 'Zend\Form\Element\Textarea',
            'attributes' => array(
                'class' => 'span11',
                'placeholder' => 'How are you?'
            ),
        ));
        $this->add(new Element\Csrf('csrf'));
        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type'  => 'submit',
                'value' => 'Submit',
                'class' => 'btn'
            ),
        ));
    }
}
```

As we know from other classes, the first thing is always the current namespace and the namespaces we want to use. When you create a form, usually we have to extend the base form class called `Form`. After that everything can be accomplished in a constructor.

The first thing is to initialize the parent class. For that we call the `__construct` method from the parent class passing the name of the form as the first parameter.

Right after that we start setting attributes of the form. In order to print the HTML that creates the form in the browser of the user, we will use view helpers. If we want to customize the attributes that are printed with the form, we should use the method, `setAttribute()`. In the second block of code, we set the method and the `css` class for the form.

Then, we arrive to the configuration of the form itself. We are going to add three fields: a text area element for the status, a submit button, and a hidden field to protect us from **Cross Site Request Forgery** (**CSRF**). ZF2 provides a way to protect us automatically from CSRF. We only need to add an element called `Csrf` to our form, and the framework will do the job of generating the token and validating it for us.

> CSRF is a type of malicious exploit of a website whereby unauthorized commands are transmitted from a user that the website trusts.

When we add elements to our form, usually we only have to pass an array with the element configuration. This array contains the name of the element, the type of the element that can be just a simple string that ZF2 will understand or the fully qualified class name of the element and some attributes that will be used while printing the HTML of the object.

text-content-form.html

We have a form, but we need to show it to the world. In order to show it, we usually use the view helpers that allow us to print the HTML for each field. To keep the code organized, we create a new partial and store the view code inside. Then, when we want to show the form on the view, we just need to include the partial and pass the form as a parameter itself.

Let's review the following code of the partial:

```
<?php $form = $this->form; ?>
<?php $form->prepare(); ?>
<?php echo $this->form()->openTag($form) ?>
<!-- Status input text -->
<?php echo $this->formElement($form->get('status')); ?>
<?php echo $this->formElementErrors($form->get('status')); ?>

<!-- Submit button -->
<?php echo $this->formElement($form->get('submit')); ?>

<!-- CSRF -->
<?php echo $this->formElement($form->get('csrf')); ?>
<?php echo $this->formElementErrors($form->get('csrf')); ?>

<?php echo $this->form()->closeTag() ?>
```

The first thing we do is call the method, `prepare()`, of the form. This will help us to do some wiring internally to prepare everything for the view helpers.

After that we start using some view helpers, such as `form()` or `formElement()`, to print the `form` tag or the `input` tag for the element we are passing as parameters. We also use a view helper called `formElementErrors()` that takes an element of the form as a parameter, takes care of checking if the validation has failed for that specific element, and prints out a list of error messages.

To add the form to the main view, we call the partial helper as shown in the following code:

```
<?php echo $this->partial(
    'forms/text-content-form.phtml',
    array('form' => $textContentForm));
?>
```

The partial helper allows us to include the contents of a separate view script on a specific point of another view. You can also pass variables and data to the view that is being included.

Modifying IndexController.php

As we handle the creation via a form, you already know forms have an action. In our case we need to point the form somewhere, receive the data, validate it, and show the wall again with the error messages in case of an error.

Now, as usual we should define new classes we want to use on this class. We need to add the form and the `Status` entity as follows:

```
use Wall\Forms\TextStatusForm;
use Wall\Entity\Status;
```

After that, we add the code that wires the form and does all the checks when it's submitted as shown in the following code:

```
    //Check if we are submitting content
    $request = $this->getRequest();
    $statusForm = new TextStatusForm;

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

        if (array_key_exists('status', $data)) {
            $result = $this->createStatus($statusForm, $user, $data);
        }

        switch (true) {
            case $result instanceOf TextStatusForm:
                $statusForm = $result;
                break;
            default:
                if ($result == true) {
                    $flashMessenger->addMessage(
                        'New content posted!'
                    );
                    return $this->redirect()->toRoute(
                        'wall',
                        array('username' => $user->getUsername())
                    );
                } else {
                    return $this->getResponse()->setStatusCode(500);
                }
                break;
        }
    }
```

The preceding code is responsible for checking the form, validating it, and calling the API to store the status. Let's analyze it section by section.

The first bit of code gets the `Request` object, creates a new instance of `TextStatusForm`, and checks if we are posting any data.

In case of a `POST`, we check the contents of the request to determine if we are creating a status entry. In that case, we pass the form, the user, and the data to the `createStatus()` method in order to create the status.

After calling the method, we check the returned value to see if the data was stored correctly or a form containing errors has been returned.

In case of success, we add a new message to the flash messenger and redirect back to the wall to show the new content.

In case of an error, we force a 500 status code. This is ugly and should be improved and handled properly in a production ready project.

The last thing we should modify in this action is the data we pass to the view and the configuration of the action of the form. This is really easy. Refer to the following code:

```
$form->setAttribute(
    'action',
    $this->url()->fromRoute(
        'wall',
        array('username' => $user->getUsername())
    )
);
```

```
$viewData['profileData'] = $user;
$viewData['textContentForm'] = $form;

if ($flashMessenger->hasMessages()) {
       $viewData['flashMessages'] = $flashMessenger->getMessages();
   }

return $viewData;
```

Here we use the `setAttribute()` method of the form to set the action we want for the form. After that we just add the form to the array we will return to the view.

After these changes on the `indexAction()` method, we need to create two more methods, `createStatus()` and `processSimpleForm()` as shown in the following code:

```
protected function createStatus($form, $user, array $data)
{
    $form->setInputFilter(Status::getInputFilter());
    return $this->processSimpleForm($form, $user, $data);
}
```

As you can see the `createStatus()` method is really simple and just configures the `InputFilter` component based on the configuration we have on the `Status` entity and then calls the `processSimpleForm()` method passing the form, the user, and the data again as follows:

```
protected function processSimpleForm($form, $user, array $data)
{
    $form->setData($data);

    if ($form->isValid()) {
        $data = $form->getData();
        $data['user_id'] = $user->getId();
        unset($data['submit']);
        unset($data['csrf']);

        $response = ApiClient::postWallContent(
            $user->getUsername(), $data
        );
        return $response['result'];
    }

    return $form;
}
```

The main responsibility of the `processSimpleForm()` method is to handle the creation of wall entries coming from forms, where we don't need to do a special treatment to the information. Basically, after validating the contents of the request against the filters and the validations are configured, we process the data to add and remove information we need or do not need. After that we call the method, `postWallContent()`, on `ApiClient` to send the data to the API.

Extending ApiClient.php

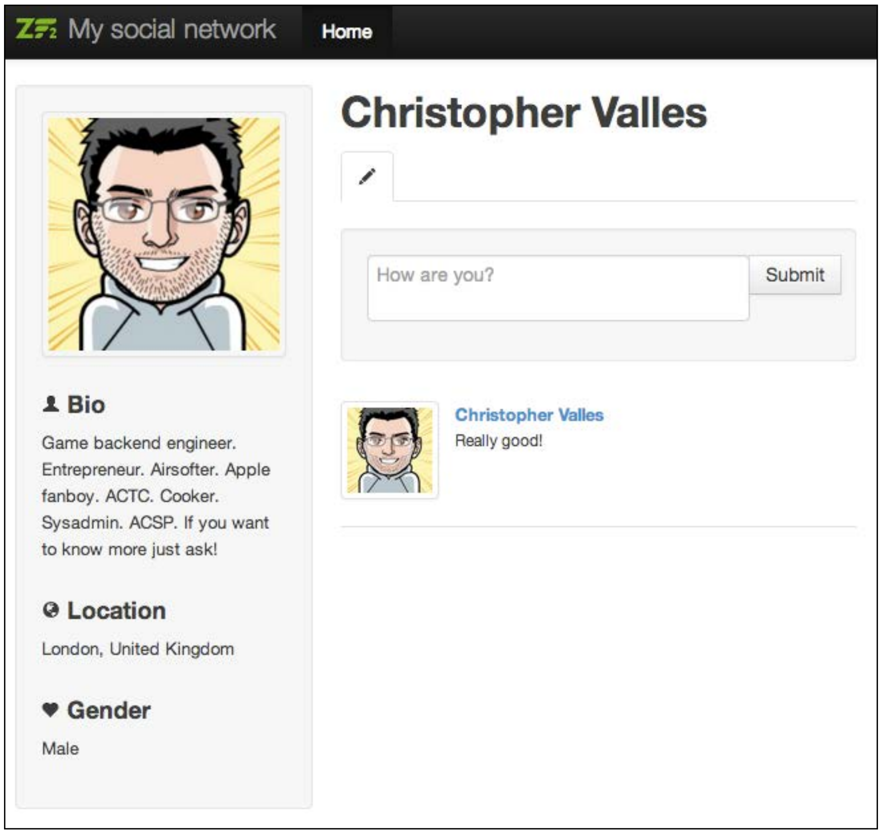
The last step is to get everything working. We need to add another method to `ApiClient` to be able to send data to the API as shown in the following code:

```
public static function postWallContent($username, $data)
{
    $url = self::$endpointHost . sprintf(
        self::$endpointWall, $username
    );
    return self::doRequest($url, $data, Request::METHOD_POST);
}
```

This method gets the data we want to send to the API, builds a URL with the corresponding parameters, and calls the method, `doRequest()`, passing the URL, the data and specifying `POST` as the HTTP method to use.

We already said that for now you have to trust me on how the `doRequest()` method works and to ignore the contents of it, but if you want you can start having a look at it.

The following screenshot shows how our wall looks like with our brand new form:



Summary

Phew, those were huge changes to the code we had! Now, users are able to send their statuses and they can see them on their walls.

In this chapter we covered how to use the filters and validators in standalone format and also attached them to a form. We also saw how to create simple forms, how to configure them, add elements to them, how to print them using view helpers, and how to handle them when they are submitted back from the browser. Finally, we extended the entities we had before to add the new content we get from the API requests.

In the next chapter it will be possible to publish images on the wall. Of course it will be built on top of what we have already seen here.

6
Working with Images – Publishing Pictures

In this chapter, we are going to add functionalities to the user wall, which will allow the users to add images. We will cover the API and frontend code as usual, and we will discover how to handle file uploads, how to validate files, and how to handle them.

By the end of this chapter, you will know:

- How to create forms that accept files
- How to add validation to those files
- How to handle the upload itself and the manipulation of an image

API development

Let's review the changes we should do on the API level to support file uploads. We will need a new table on the database to store the relations between the images uploaded and the users. As we are creating a new table on the database, we will also need a table gateway object to access the data. Finally, we are going to modify the `create()` method to handle the code for the status text and add the code for the image upload.

Requirements

The requirements for the API are the same in terms of HTTP methods. The only thing we are going to change is the way we process the data in the `create()` method. Essentially, when a request arrives to the `create()` method, we should try to detect if we are fulfilling a request to create a new status entry or are trying to post a new image.

The `create()` method will inspect the contents of the `$data` parameter, searching for an entry on the array. If we find the key status in the array, we will call it the `createStatus()` method, which contains the code we made in *Chapter 5, Handling Text Content – Posting Text*. If we find the key image, we will call the `createImage()` method that will handle the request.

Working with the database

Let's have a look at the new table we have to create. Essentially, it is the same as the one we created for the status; the only difference is that we are going to store the filename of the image instead of the status.

This table is called `users_images` and will hold the following data:

- Id
- User_id
- Filename
- Created at
- Updated at

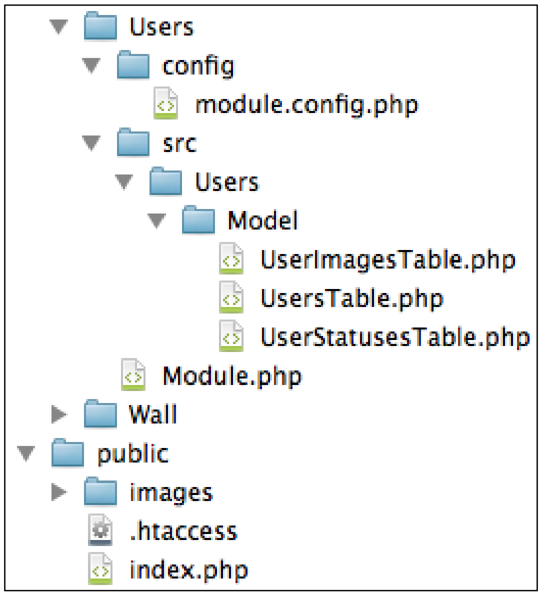
As usual, the table will also contain two columns to keep a track of the creation and update timestamps of each row.

By now, you should be pretty confident about how to connect to the database hosted on the VM, so we can just jump to see how the syntax to create the table looks , as shown in the following code:

```
CREATE TABLE `user_images` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(11) unsigned DEFAULT NULL,
  `filename` varchar(44) DEFAULT '',
  `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
```

Changes on the module structure

Now let's have a look at how the folder and file structure will like on the API side of the project. Notice that we added a new folder inside the `public` folder called `images`. This is where the uploaded images are going to be stored.



```
▼ Users
    ▼ config
        module.config.php
    ▼ src
        ▼ Users
            ▼ Model
                UserImagesTable.php
                UsersTable.php
                UserStatusesTable.php
        Module.php
▶ Wall
▼ public
    ▶ images
    .htaccess
    index.php
```

Creating the UserImagesTable.php file

We are not going to spend too much time on this file because it is a copy of `UserStatusesTable.php` and we are only changing a few lines. We will essentially copy the file and review the changes we should make.

```
class UserImagesTable extends AbstractTableGateway implements
AdapterAwareInterface
```

Of course, the first thing we should do is change the name of the class; in this case the class will be called `UserImagesTable`.

The second thing we should change is the name of the table we are going to query; the line will be as follows:

```
protected $table = 'user_images';
```

We will change one line in the `create()` method, but let's see the full method to have a clear understanding about where the change should be made:

```
public function create($userId, $filename)
{
    return $this->insert(array(
        'user_id' => $userId,
        'filename' => $filename,
        'created_at' => new Expression('NOW()'),
        'updated_at' => null
    ));
}
```

The last method, `getInputFilter()`,will remain more or less the same; the only thing we need to do is to remove the block of code that configures the validators and filters for the status field. This will leave the method with just the configuration for the `user_id` field.

Extending the IndexController.php file

Now that we have the table and the class to access data, let's see the changes we should make in the controller and the improvements we should do in the meantime.

When a user uploads an image, it should appear on the wall as a new entry, which means that we should return this data when the client requests the wall.

Earlier, it was easy as we just had the status updates; but now, we should somehow merge the status updates and the images, and order the entries by the creation date from the most recent to the ones in the past. Basically, that's the approach we are going to follow: first retrieve the data, then merge it, and finally order it by a common column; in this case, `created_at`.

```
    public function get($username)
    {
        $usersTable = $this->getUsersTable();
        $userStatusesTable = $this->getUserStatusesTable();
        $userImagesTable = $this->getUserImagesTable();

        $userData = $usersTable->getByUsername($username);
        $userStatuses = $userStatusesTable->getByUserId(
            $userData->id
        )->toArray();
        $userImages = $userImagesTable->getByUserId($userData->id)
            ->toArray();

        $wallData = $userData->getArrayCopy();
        $wallData['feed'] = array_merge($userStatuses, $userImages);

        usort($wallData['feed'], function($a, $b){
            $timestampA = strtotime($a['created_at']);
            $timestampB = strtotime($b['created_at']);
            if ($timestampA == $timestampB) {
                return 0;
            }

            return ($timestampA > $timestampB) ? -1 : 1;
        });

        if ($userData !== false) {
            return new JsonModel($wallData);
        } else {
            throw new \Exception('User not found', 404);
        }
    }
```

Ok, that's the whole method; let's review a couple of lines of this method .In the first block, we get instances of the tables. Then in the second block, we get the data based on `user_id`.

Next, we arrive at the merge phase; basically, we ask for an array version of the results we just got from the db and merge them under the key feed of the array.

After that, we find the order section. We basically use the PHP method, `usort()`, to sort an array based on an anonymous function we created. Inside the function, we just check if the `created_at` value of one item is equal, more, or lesser than the `created_at` value of the other.

Now, let's jump to the `create()` method and see how we are going to change it to accommodate the new code. The first thing we should do is try to detect the type of request and then based on that, forward the flow to a more specific method that will take care of the request.

```
    public function create($data)
    {
        if (array_key_exists('status', $data) &&
            !empty($data['status'])) {
            $result = $this->createStatus($data);
        }

        if (array_key_exists('image', $data) &&
            !empty($data['image'])) {
            $result = $this->createImage($data);
        }

        return $result;
    }
```

As you can see in this snippet of code, we just check if a certain key exists inside the data. The first block tries to determine if the request is for the creation of a status and to accomplish that, we check if the status key exists on the `$data` array.

The second block is related to the image a user might send. The code keeps checking for a key called image and will forward everything to the `createImage()` method if the conditions are satisfied.

Let's us now review the new method, `createImage()`; the method is a little bit long and is doing quite a lot of stuff, so we are going to see it in sections:

```
$userImagesTable = $this->getUserImagesTable();
```

The first line of the following code snippet just gets the table gateway for the images using the `getUserImagesTable()` method, which we will see later.

```
$filters = $userImagesTable->getInputFilter();
$filters->setData($data);
```

In the next block, we focus on the validation of the data. This step is the same as the one we use on the `createStatus()` method, the only difference is in where we get the input filter from.

```
if ($filters->isValid()) {
    $filename = sprintf(
        'public/images/%s.png',
        sha1(uniqid(time(), true))
    );
    $content = base64_decode($data['image']);
    $image = imagecreatefromstring($content);

    if (imagepng($image, $filename) === true) {
        $result = new JsonModel(array(
            'result' => $userImagesTable->create(
                $data['user_id'],
                basename($filename)
            )
        ));
    } else {
        $result = new JsonModel(array(
            'result' => false,
            'errors' => 'Error while storing the image'
        ));
    }
    imagedestroy($image);
} else {
    $result = new JsonModel(array(
        'result' => false,
        'errors' => $filters->getMessages()
    ));
}
```

This is the next block of code we will encounter in this method. First of all, we have a condition based on the validation of the data. In case it's wrong, we just return an error message to the frontend.

When the data is validated, we jump to handle the image. The image itself is sent by the browser along with the request encoded in base64. The job on the API side is to store that image in a file. To accomplish that, we first create a name for the file as we want to avoid collisions on the filenames, and then generate a random one as follows:

```
$filename = sprintf(
    'public/images/%s.png',
    sha1(uniqid(time(), true))
);
```

As you can see here, we generate a unique ID prepending the current time and activating the entropy flag of the `uniqid()` method. Then we just encode the result using the sha1 algorithm.

We do two things just after that. Firstly, we decode the image from base64 to the raw data and store it in a variable. Secondly, we load the raw data to an image with the help of `imagecreatefromstring()`, which takes a variable with the image data as a parameter.

The last thing we do here is store the image converted to PNG; we accomplish that using the `imagepng()` method and the filename we generated before. If the operation succeeds, we store the filename in the database linked to the `user_id` attribute and then return the rows affected to the client. If not, we just return an error.

The following is the full code for this method:

```
protected function createImage($data)
{
    $userImagesTable = $this->getUserImagesTable();

    $filters = $userImagesTable->getInputFilter();
    $filters->setData($data);
    $filters->isValid();
```

```
        if ($filters->isValid()) {
            $filename = sprintf(
                'public/images/%s.png',
                sha1(uniqid(time(), true))
            );
            $content = base64_decode($data['image']);
            $image = imagecreatefromstring($content);

            if (imagepng($image, $filename) === true) {
                $result = new JsonModel(array(
                    'result' => $userImagesTable->create(
                        $data['user_id'],
                        basename($filename)
                    )
                ));
            } else {
                $result = new JsonModel(array(
                    'result' => false,
                    'errors' => 'Error while storing the image'
                ));
            }
            imagedestroy($image);
        } else {
            $result = new JsonModel(array(
                'result' => false,
                'errors' => $filters->getMessages()
            ));
        }

        return $result;
    }
```

Finally, we have arrived on the last change we need to do on the controller. We are going to create a method that will create an instance of the table gateway for the new table we are using. The code doesn't need any explanation as it is almost the same as the one we saw before:

```
    protected function getUserImagesTable()
    {
        if (!$this->userImagesTable) {
            $sm = $this->getServiceLocator();
            $this->userImagesTable = $sm->get(
                'Users\Model\UserImagesTable'
            );
        }
        return $this->userImagesTable;
    }
```

As you also know, this method uses a property called `userImagesTable` that should be added to the class. The code is pretty simple and similar to the one used in the previous chapters:

```
protected $userImagesTable;
```

Changing the module.config.php file

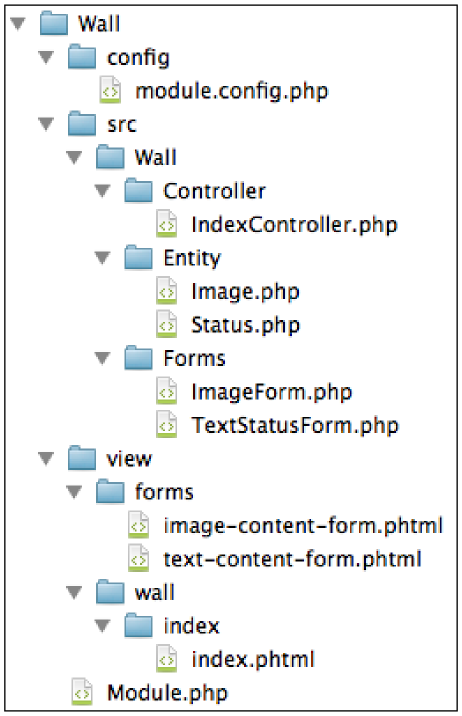
As we did in the previous chapter, we only have to tell **Dependency Injector** about the new table gateway we have. To do that, we just need to add the following line in the `module.config.php` file of the `Users` module:

```
'Users\Model\UserImagesTable' => 'Users\Model\UserImagesTable'
```

Frontend

Ok, now we have the API ready to start accepting images. We will need to add a new form on the client to accept the files from the user, validate the image, and upload it to the API.

The following screenshot shows how the folder structure has changed:



```
Wall
  config
    module.config.php
  src
    Wall
      Controller
        IndexController.php
      Entity
        Image.php
        Status.php
      Forms
        ImageForm.php
        TextStatusForm.php
  view
    forms
      image-content-form.phtml
      text-content-form.phtml
    wall
      index
        index.phtml
  Module.php
```

Creating the Image.php file

When we get the info from the API, we store it using entities and, of course, we need a new one for the images. In this case, we will do it a bit differently. We saw that we can store the input filter configuration in the entity; however, for now we will store it directly on the form instead of the entity. This class is more or less the same as the `Status` entity; we only have a couple of differences as follows:

```
public $domain = 'http://zf2-api/images/';
```

This property will be used in a method that generates the URL of the image. Notice that we are passing the image to the API and storing it on that side of the application, which means that the image should be accessible via a URL to be able to see it. That's why we store the images in a folder inside the public folder. If you were doing this in a proper project, you would usually serve the image from a **content distribution network** (**CDN**) or another cookieless domain, but for the purpose of this project we will keep it there as follows:

```
public function getUrl()
{
    return $this->domain . $this->getFilename();
}
```

This is the method that generates a URL for the current image. Basically, it creates a new string, gluing together the domain stored as a property in the class and the filename of the image.

The definition of CDN by Wikipedia is as follows:

A content delivery network or content distribution network (CDN) is a large distributed system of servers deployed in multiple data centers across the Internet. The goal of a CDN is to serve content to end users with high availability and high performance.

Adding the ImageForm.php file

Now we should create the form that the users will use to upload the images to the API. It's really simple as we only have one field. The difference from the other forms is that in this case, we are going to store the configuration of the input filters. In order to autoconfigure the form with config, we will implement a new interface called `InputFilterProviderInterface`, which will force us to implement a method called `getInputFilterSpecification()`, which in turn will return the filter configurations when called.

In order to implement the interface, we have to declare that we will use it first. This will be done by adding the following lines:

```
use Zend\InputFilter\InputFilterProviderInterface;

class ImageForm extends Form implements
    InputFilterProviderInterface
```

You will see that when we declare the class, we also say that we will implement the `InputFilterProviderInterface` interface.

Before we add the fields on the constructor, we will change them in this form just to show that we have alternative ways to do the same things, as follows:

```
public function __construct($name = null)
{
    parent::__construct('image-content');

    $this->setAttribute('method', 'post');
    $this->setAttribute('class', 'well input-append');

    $this->prepareElements();
}
```

This is the new constructor. You can see we call the parent constructor like we do on forms and then we set the attributes of the form, in this case, the method and class. Finally, we call a method called `prepareElements()`, which will add the elements to the form as shown in the following code:

```
public function prepareElements()
{
    $this->add(array(
        'name' => 'image',
        'type'  => 'Zend\Form\Element\File',
        'attributes' => array(
            'class' => 'span11',
        ),
    ));
    $this->add(new Element\Csrf('csrf'));
    $this->add(array(
        'name' => 'submit',
        'attributes' => array(
            'type'  => 'submit',
            'value' => 'Submit',
            'class' => 'btn'
        ),
    ));
}
```

This is the method we were talking about, with which we take care of the creation of the elements of the form. In our case, we create a new element specifying `Zend\Form\Element\File` as type, which will create an input of the type of file in the view. The other couple of elements should be familiar to you, as we already used them on other forms.

```
public function getInputFilterSpecification()
{
    return array(
        'image' => array(
            'required' => true,
        )
    );
}
```

This is the last method we will implement on the form and if you check, it is also the method we should create to implement the interface. As we have already mentioned, this method should return the input filter configuration array, so basically that's what we are doing here. If you check the validators, we just mark the field as required because we validate the contents of the uploaded file manually.

Showing the form in the image-content-form.phtml file

This is the view associated with the form we just created. As you will see, it is almost the same as the ones we created before; we just change the helper that we will use to render the file field, which means that instead of `formElement()`, we use `formFile()`, as shown in the following code:

```
<?php $form = $this->form; ?>
<?php $form->prepare(); ?>
<?php echo $this->form()->openTag($form) ?>
<!-- Status input text -->
<?php echo $this->formFile($form->get('image')); ?>
<?php echo $this->formElementErrors($form->get('image')); ?>

<!-- Submit button -->
<?php echo $this->formElement($form->get('submit')); ?>

<!-- CSRF -->
<?php echo $this->formElement($form->get('csrf')); ?>
<?php echo $this->formElementErrors($form->get('csrf')); ?>

<?php echo $this->form()->closeTag() ?>
```

Extending the User.php file

We already have the code required to fulfill the feed data in the `User` entity, but we need to extend it now in order to process the images posted on the wall.

```
public function setFeed($feed)
{
    $hydrator = new ClassMethods();

    foreach ($feed as $entry) {
        if (array_key_exists('status', $entry)) {
            $this->feed[] = $hydrator->hydrate(
                $entry, new Status()
            );
        } else if (array_key_exists('filename', $entry)) {
            $this->feed[] = $hydrator->hydrate(
                $entry, new Image()
            );
        }
    }
}
```

As you can see in the highlighted code, we added a new `else if` block to check if we are processing an image. We identify the image because the entry will contain a key called `filename`.

Now that the user is able to upload pictures, we can correctly process the user avatar as follows:

```
public function setAvatar($avatar)
{
    if (empty($avatar)) {
        $defaultImage = new Image();
        $defaultImage->setFilename('default.png');
        $this->avatar = $defaultImage;
    } else {
        $hydrator = new ClassMethods();
        $this->avatar = $hydrator->hydrate($avatar, new Image());
    }
}
```

This is the new content of the method; it entirely replaces what we had before. We are checking to see if the user has an avatar assigned in the database. If no avatar is assigned, we will show a default image. In order to do that, we create an `Image` object and manually set the filename of the avatar we want to show as default.

In case the user has an avatar assigned in the database, we just load the information using `Hydrator` into an `Image` entity.

Modifying the index.phtml file

Now that we know how we are going to store the data sent by the API, let's see the modifications we need to do on the view to show the new content. As is usual with views, when the HTML is mixed with the PHP code, the modifications will be verbose and large in number.

```
<?php if ($entry instanceOf \Wall\Entity\Status) : ?>
    <p><?php echo $entry->getStatus() ?></p>
<?php elseif ($entry instanceOf \Wall\Entity\Image) : ?>
    <p>
        <img src="<?php echo $entry->getUrl() ?>" />
    </p>
<?php endif; ?>
```

This is the main change; the other change is that instead of iterating through statuses, we iterate through the feed. As you can see, we basically check the type of the object and depending on the type, we show one type of content or the other.

Modifying the IndexController.php file

Let's jump to the changes we should do on the controller to be able to receive the file uploaded by the user, validate it, and send it to the API. Also,to be able to provide a feedback to the end user about the content posted.

Earlier, in the `indexAction()` method, we were detecting the submission of content by inspecting the request and creating the new status by calling the API. Let's change that to detect the type of content the user wants to create and redirect the flow of the app to the method that will take care of it.

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

    if (array_key_exists('status', $data)) {
        $result = $this->createStatus($statusForm, $user, $data);
    }

    if (!empty($request->getFiles()->image)) {
        $data = array_merge_recursive(
            $data,
            $request->getFiles()->toArray()
        );
        $result = $this->createImage($imageForm, $user, $data);
    }
```

```
        switch (true) {
            case $result instanceOf TextStatusForm:
                $statusForm = $result;
                break;
            case $result instanceOf ImageForm:
                $imageForm = $result;
                break;
            default:
                if ($result == true) {
                    $flashMessenger->addMessage(
                        'New content posted!'
                    );
                    return $this->redirect()->toRoute(
                        'wall',
                        array('username' => $user->getUsername())
                    );
                } else {
                    return $this->getResponse()->setStatusCode(500);
                }
                break;
        }
    }
```

We saw how we should deal with the creation of statuses in *Chapter 5, Handling Text Content – Posting Text*. The same concept applies to the image request, but instead of checking the contents of the array, we check if we have received any file within the request; in that case, we assume that the user has uploaded an image and jump to process it by calling the `createImage()` method. The way we handle the response of the method is exactly the same as the one we saw for the status content.

Let's review the `createImage()`method, which is a little complicated, as shown in the following code:

```
if ($data['image']['error'] != 0) {
    $data['image'] = null;
}

$form->setData($data);
```

These three lines basically check if there is an error while uploading the file. If we find an error, we set the value to `null` to force the form validation to fail as shown in the following code:

```
$size = new Size(array('max' => 2048000));
$isImage = new IsImage();
$filename = $data['image']['name'];
```

In the next block, we create an instance of two validators called `Size()` and `IsImage()` that basically check the size of a file and whether the file is an image or not. Of course, to be able to use these two new validators, we should tell PHP that we want to use them at the beginning of the class with the following lines:

```
use Zend\Validator\File\Size;
use Zend\Validator\File\IsImage;
```

After that, we need to use a class called `Zend\File\Transfer\Adapter\Http`, which will take care of receiving the file and validating it, using the validators we configured as follows:

```
$adapter = new \Zend\File\Transfer\Adapter\Http();
$adapter->setValidators(array($size, $isImage), $filename);
```

As you can see, we create an instance of the class and then we set the validators we configured before. As a second parameter, we are passing the name of the file we extracted on the previous block as shown in the following code:

```
if (!$adapter->isValid($filename)){
    $errors = array();
    foreach($adapter->getMessages() as $key => $row) {
        $errors[] = $row;
    }
    $form->setMessages(array('image' => $errors));
}
```

Next thing we do is validate the file itself calling the `isValid()` method on the adapter object. If we find errors, we will iterate over them to add them on the form as an error message.

Now we have arrived on the big chunk of code in this method. The one that actually receives the file, renames it to avoid filename collisions, and uploads it to the API. The reason why we rename the pictures while they are being uploaded is to avoid two users uploading images with the same filename and overwriting one with the other. Renaming the file when we receive it saves us from this problem. Notice that the following blocks of code are executed if the validation of the form is correct:

```
$destPath = 'data/tmp/';
$adapter->setDestination($destPath);
```

This is the first block in which we just define the path where we want to upload the file and set config on the adapter. This is just a temporary folder because once the file is uploaded to the API, we will remove it from the folder.

```
$fileinfo = $adapter->getFileInfo();
preg_match('/.+\/(.+)/', $fileinfo['image']['type'], $matches);
```

```
$extension = $matches[1];
$newFilename = sprintf(
    '%s.%s',
    sha1(uniqid(time(), true)),
    $extension
);
```

In the second block of code, we get information related to the image from the adapter, and then we do a regular expression to get the extension of the file. After that, we generate a new filename and if you take a look, we are doing it the same way as in the API. However, in this case, we are reusing the original extension of the file as follows:

```
$adapter->addFilter('File\Rename',
    array(
        'target' => $destPath . $newFilename,
        'overwrite' => true,
    )
);
```

Now in the third block of code, we add a filter to the file adapter in order to rename the file when we receive it. We set the target of the file passing the path and the filename, and then we force it to overwrite. In theory, we will never have to overwrite because the filenames are supposed to be unique. This is illustrated in the following code:

```
if ($adapter->receive($filename)) {
    $data = array();
    $data['image'] = base64_encode(
        file_get_contents(
            $destPath . $newFilename
        )
    );
    $data['user_id'] = $user->getId();

    unlink($destPath . $newFilename);

    $response = ApiClient::postWallContent(
        $user->getUsername(), $data
    );
    return $response->isSuccess();
}
```

This is the last block of code in this method. Here, we are asking the file adapter to receive the file and move it from the temp folder that PHP uses to the one we specified that will execute the filters we attached.

After that, we build an array that will be the container for the data. We read the contents of the file and then we encode it using base64 to store it as a string on the `$data` array; we also add the user ID on the params.

Then, we use `ApiClient` to send the data to our API. After making the request, we just remove the file from the `tmp` folder and return the response of the API.

Now that we have seen how to handle the file upload on the client, let's take a look at how to provide feedback to the user using flash messages.

ZF2 provides a controller plugin called `flashMessenger`, which takes care of storing the messages on the user session and removing them after certain hops or in the next request. The first thing we will do is get a copy of the plugin and store it on a variable inside the `indexAction()` method:

```
$flashMessenger = $this->flashMessenger();
```

Then, the only thing we should do is add a success message in each case: one for the creation of the status and another for the creation of the image. Let's see both the sections of code with the new line added:

```
if ($result === true) {
    $flashMessenger->addMessage('New status posted!');
    return $this->redirect()->toRoute(
        'wall',
        array('username' => $user->getUsername())
    );
} else {
    return $this->getResponse()->setStatusCode(500);
}
```

This one is the block related to the creation of a new status; let's review the following block of code related to the picture:

```
if ($result instanceOf ImageForm) {
    $imageForm = $result;
} else {
    if ($result === true) {
        $this->flashMessenger()->addSuccessMessage(
            'Your image has been posted!'
        );

        return $this->redirect()->toRoute(
            'wall',
            array('username' => $user->getUsername())
        );
    } else {
        return $this->getResponse()->setStatusCode(500);
    }
}
```

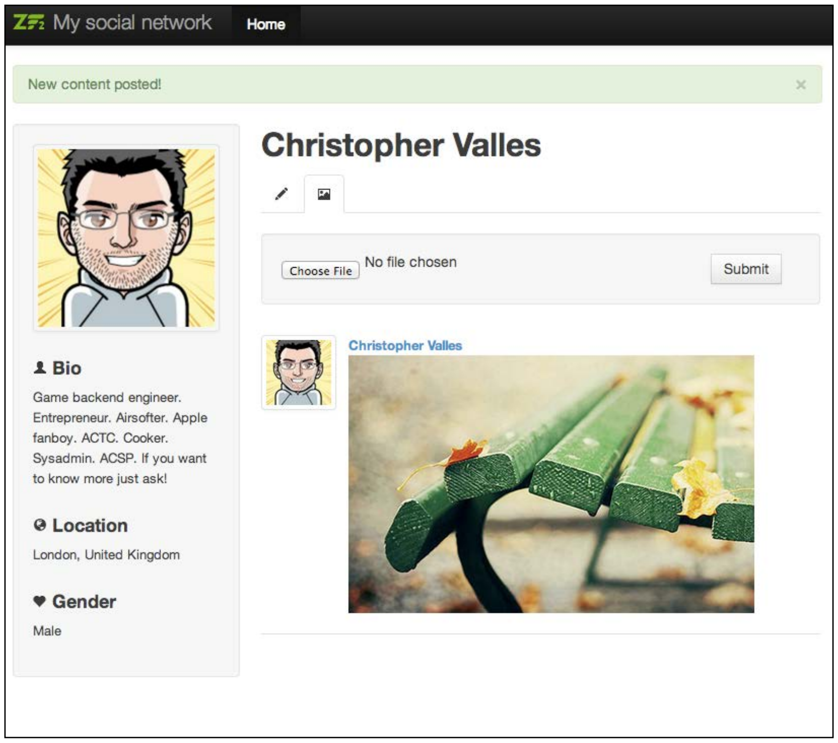
Modifying the layout.phtml file

We should modify the layout of the app to show the messages in a proper place on the website. The only thing we should do is call the `FlashMessenger` view helper to configure it and render the messages.

```
<div class="container wrapper">
    <?php $flash = $this->flashMessenger()->setMessageOpenFormat(
        '<div%s><button type="button" class="close"
        data-dismiss="alert" aria-hidden="true">&times;</button>'
        )->setMessageSeparatorString('<br />')
        ->setMessageCloseString('</div>'); ?>
    <?php echo $flash->render('error',
        array('alert', 'alert-dismissable', 'alert-danger'));
    ?>
    <?php echo $flash->render('info',
        array('alert', 'alert-dismissable', 'alert-info'));
    ?>
    <?php echo $flash->render('default',
        array('alert', 'alert-dismissable', 'alert-warning'));
    ?>
    <?php echo $flash->render('success',
        array('alert', 'alert-dismissable', 'alert-success'));
    ?>
    <?php echo $this->content; ?>
</div>
```

This is the code. As you can see, the first PHP block configures the view helper with the HTML we want to use to display the messages. After that, we tell the view helper to print the messages on each category.

The following screenshot shows how everything looks when completed:



Game backend engineer. Entrepreneur. Airsofter. Apple fanboy. ACTC. Cooker. Sysadmin. ACSP. If you want to know more just ask!

London, United Kingdom

Male

Summary

Wow! This chapter was vast, but we gave the users a new functionality to use on our social network. As people say, a picture is worth a thousand words—now they can upload pictures!

In this chapter, we covered how to detect a request type based on the contents of the request. We also refactored the way we were inserting the data in the database and the way we were retrieving it. We saw how to handle file uploads from the client and the API, and how to deal with filename collisions. We also covered how to validate file uploads, check the file type, and the size. Finally, we saw how to provide feedback to the user using flash messages and using the `FlashMessenger` view helper.

In the next chapter, we will allow our users to post URLs on their walls. Thanks to that, we will see how to deal with requests to external sites and how to process HTML using the `Zend\Dom` components.

7
Dealing with URLs – Posting Links

In this chapter, we will add the last bit of information a user can post to their wall. They will be able to post links. As an extra step, we will do a refactor of the code we have made so far in order to avoid the duplication of the code we have right now.

By the end of this chapter, you will know how to use `Zend\Http\Client` to get the HTML of a web page and then extract information, such as the title. We will also cover how to create a custom validator to use on our forms in order to validate URLs or anything you will need.

API development

We will follow the same approach as the last chapter to add this functionality. We will also need to create a new table in the database to add the link information. As usual, when we create a new table we should also create the table gateway that we will use to access the data. Finally, we need to create a new method that will be linked up inside the `create()` method and will be called based on the data inside the request.

Requirements

The requirements for this functionality will not change in terms of the HTTP methods we are exposing right now. The only difference is that now the `create()` method will check for a new key inside the `$data` array. If found, we will execute another specialized method called `createLink()` that will visit the web page, extract the data we want, and store it in the database. Finally, we will need to create a custom validator that we will re-use on the client code in order to validate a URL.

Working with the database

Now let's see the structure of the table we have to create to store the data related with the links that a user will post. Essentially, it is the same as the one we have already created, but we will add more fields to store the title and the URL itself.

This table is called `users_links` and will hold the following data:

- Id
- User_id
- URL
- Title
- Created at
- Updated at

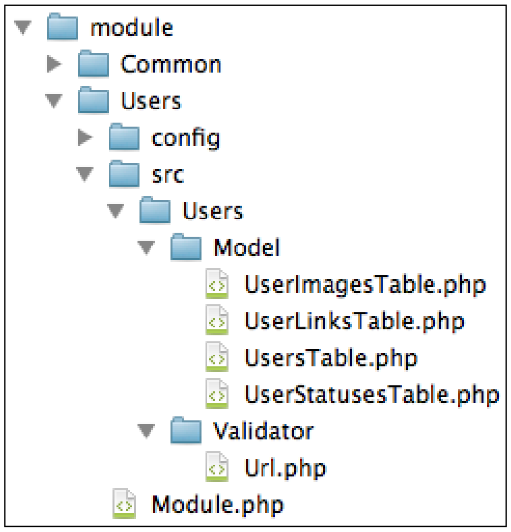
In order to store data related to the creation and update times we will also add two timestamps called `created_at` and `updated_at`.

The following is the statement to create the new table:

```
CREATE TABLE `user_links` (
  `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,
  `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;
```

Changes in the module structure

Now let's have a look at how the folder and file structure will look for the API side of the project:



Adding the UserLinksTable.php file

At the beginning this file is almost the same as the one we created before, but with a few changes and something important that we have to highlight. Let's see the contents of the file:

```
namespace Users\Model;

use Zend\Db\Adapter\Adapter;
use Zend\Db\TableGateway\AbstractTableGateway;
use Zend\Db\Adapter\AdapterAwareInterface;
use Zend\Db\Sql\Expression;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Factory as InputFactory;
use Users\Validator\Url;
```

As usual we start with the current namespace and declare all the namespaces we want to use in this class. If you review the list carefully you will see that the last one is a custom class that implements a validator. We will review that shortly.

```
class UserLinksTable extends AbstractTableGateway implements
AdapterAwareInterface
```

In the table gateways that we create on the API side, we should declare the name of our class, extend `AbtractTableGateway`, and implement `AdapterAwareInterface`.

```
protected $table = 'user_links';
```

Now we define the name of the table; in this case, as we already saw, the table is called `user_links`.

After that we will find the methods `setDbAdapter()` and `getByUserId()`; these two methods are exactly the same as the other table gateways that we have so we will just copy them over to the following file:

```
public function create($userId, $url, $title)
{
    return $this->insert(array(
        'user_id' => $userId,
        'url' => $url,
        'title' => $title,
        'created_at' => new Expression('NOW()'),
        'updated_at' => null
    ));
}
```

This is the `create()` method of this table gateway. As you can see, we just pass to the `insert()` method all the data that comes as parameters to the array. We also use the `Zend\Db\SQL\Expression` object to add the `NOW()` value to the `created_at` column.

Finally, let's review the method `getInputFilter()`. The first section should look familiar because essentially it is the same as the one we have in other table gateways, but the second part has something different, as shown in the following code:

```
public function getInputFilter()
{
    $inputFilter = new InputFilter();
    $factory = new InputFactory();

    $inputFilter->add($factory->createInput(array(
        'name'     => 'user_id',
        'required' => true,
        'filters'  => array(
            array('name' => 'StripTags'),
            array('name' => 'StringTrim'),
            array('name' => 'Int'),
        ),
        'validators' => array(
            array('name' => 'NotEmpty'),
            array('name' => 'Digits'),
            array(
                'name' => 'Zend\Validator\Db\RecordExists',
                'options' => array(
                    'table' => 'users',
                    'field' => 'id',
                    'adapter' => $this->adapter
                )
            )
        ),
    )));

    $inputFilter->add($factory->createInput(array(
        'name'     => 'url',
        'required' => true,
        'filters'  => array(
            array('name' => 'StripTags'),
            array('name' => 'StringTrim')
        ),
        'validators' => array(
            array('name' => 'NotEmpty'),
            array(
                'name' => 'StringLength',
                'options' => array(
                    'max' => 2048
                )
            ),
            array('name' => '\Users\Validator\Url'),
        ),
    )));

    return $inputFilter;
}
```

As you can see, in the URL input filter we are adding a validator called `\Users\Validator\Url`. This will include the custom validator in the validator chain, which we will create soon to validate URLs.

Modifying the IndexController.php file

Ok, we now have access to the new table we created in the database. Let's write the code that will inspect the HTML of the link sent by the user, extract the title of the page, and store the data in the database:

```
use Zend\Http\Client;
use Zend\Filter\FilterChain;
use Zend\Filter\StripTags;
use Zend\Filter\StringTrim;
use Zend\Filter\StripNewLines;
use Zend\Dom\Query;
```

These are the first lines we have to add to the controller next to the other classes we use in this class. In this controller we will visit the web page pointed by the URL provided by the user, then we will try to extract the title of the page from the HTML code using **XPath** and store it. That's why we will use `Zend\Http\Client` and a filter chain with `StripTags`, `StringTrim`, and `StripNewLines` filters.

```
protected $userLinksTable;
```

We will add this property to the class in order to store the table gateway instance that we will create in the `getUserLinksTable()` method.

```
public function create($data)
{
    if (array_key_exists('status', $data)
       && !empty($data['status'])) {
        $result = $this->createStatus($data);
    }

    if (array_key_exists('image', $data)
       && !empty($data['image'])) {
        $result = $this->createImage($data);
    }

    if (array_key_exists('url', $data)
       && !empty($data['url'])) {
        $result = $this->createLink($data);
    }

    return $result;
}
```

This is the modified version of the `create()` method. As you can see, we add a new block at the end checking if the key URL exists in the `$data` array. In that case we will execute the specialized method `createLink()`, which will take care of the rest.

Let's now review the new specialized method that will do the hard work. As the method is big, we will see it block by block and then the full code.

```
$userLinksTable = $this->getUserLinksTable();

$filters = $userLinksTable->getInputFilter();
$filters->setData($data);
```

This is the first block. We create an instance of the table and then we get the filters from the table gateway. After that we set the data in the filters.

```
if ($filters->isValid())
```

This is the line that actually tests if the data passed is valid or not. In case of failure we jump to the end of the method and return an error.

```
return new JsonModel(array(
    'result' => false,
    'errors' => $filters->getMessages()
));
```

In case the validation is successful, we proceed to create the HTTP client to get the contents of the website.

```
$data = $filters->getValues();

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

As you can see we get the data from the filters and then create an instance of `Zend\Http\Client`. After that we just configure the options for the client and issue the request.

```
if ($response->isSuccess())
```

As before, we test if the request was successful or not. In case of an error, we will jump again to the end of the method. Of course if you are actually doing a production grade application, you will handle the different reasons, for example, why it is not possible to get the contents of the link and the act based on that. But here, for the purpose of this project, returning a generic error is enough.

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

$dom = new Query($html);
$title = $dom->execute('title')->current()->nodeValue;
```

This is the next block of code. As the request was successful, we can get the contents of the website calling the `getBody()` method of the response object. After that we convert the encoding of the contents from UTF-8 con HTML-ENTITIES. This is done to avoid issues later on when we work with the HTML.

Now that the content is properly encoded we will extract the title from the HTML of the web page. This can be done in multiple ways, using regular expressions, just doing a search for the `<title>` tag and then processing the text, using a library that mimics the **jQuery CSS Selector** syntax, using XPath to select nodes in an XML document, or just by using the `Zend\Dom\Query` component and getting the tag by its name.

We will go with the last option that seems to be a simple one. That's what we accomplish on the last two lines of the block of code we saw earlier.

```
if (!empty($title)) {
    $filterChain = new FilterChain();
    $filterChain->attach(new StripTags());
    $filterChain->attach(new StringTrim());
    $filterChain->attach(new StripNewLines());

    $title = $filterChain->filter($title);
} else {
    $title = null;
}
```

This is the next section we will review. Basically we check the number of elements that match our selection and then process the contents; if we don't find any elements, we will set the title to null.

If we get more than zero `<title>` elements, it means that we potentially have the info we need. Keep in mind that if we find more than one, then something is wrong with that website but we will not worry about that for now and will just extract the data of the first element. Again, if you are doing this at a professional level you will have to take care about the content of the element, validate it, and so on.

Well, now that we have found the `<title>` element, we will prepare the filter chain, attaching the instances of the three filters we want to use. In this case we will remove any HTML tag found inside the text by using the `StripTags` filter, then we will remove any whitespace found at the beginning or end of the text by using the `StringTrim` filter; finally, we will remove any new lines we can find by using the filter `StripNewLines`.

After all the configuration of the chain, we just execute the `filter()` method passing a parameter, `nodeValue` of the first item we found, and that will give us the text we will use as a title for the URL.

We arrive now at the last block of this method where we actually store the data on the table we created using the table gateway and return the affected rows as a result.

```
return new JsonModel(array(
    'result' => $userLinksTable->create(
        $data['user_id'],
        $data['url'],
        $title
    )
));
```

The last change we should do on the controller is add the `getUserLinksTable()` method to create an instance of the table gateway and store it in the property we created at the beginning.

```
protected function getUserLinksTable()
{
    if (!$this->userLinksTable) {
        $sm = $this->getServiceLocator();
        $this->userLinksTable = $sm->get(
            'Users\Model\UserLinksTable'
        );
    }
    return $this->userLinksTable;
}
```

Ok, now that we have seen the code related to the storage side of the controller, let's review the small changes we should do on the `get()` method to be able to send to the client the links a user has on the wall.

We will add a line in the first block to get an instance of the table gateway, then in the second block we will retrieve the data based on the `used_id` sent on the request and store that in a variable called `$userLinks`. Finally, we will merge the `$userLinks` variable with the `$userStatuses` and `$userImages` we were fetching before.

The following is the final code of this method:

```
public function get($username)
{
    $usersTable = $this->getUsersTable();
    $userStatusesTable = $this->getUserStatusesTable();
    $userImagesTable = $this->getUserImagesTable();
    $userLinksTable = $this->getUserLinksTable();

    $userData = $usersTable->getByUsername($username);
    $userStatuses = $userStatusesTable->getByUserId(
        $userData->id
    );
    $userImages = $userImagesTable->getByUserId($userData->id);
    $userLinks = $userLinksTable->getByUserId($userData->id);

    $wallData = $userData->getArrayCopy();
    $wallData['feed'] = array_merge(
        $userStatuses->toArray(),
        $userImages->toArray(),
        $userLinks->toArray()
    );

    usort($wallData['feed'], function($a, $b){
        $timestampA = strtotime($a['created_at']);
        $timestampB = strtotime($b['created_at']);

        if ($timestampA == $timestampB) {
            return 0;
        }

        return ($timestampA > $timestampB) ? -1 : 1;
    });
```

```
        if ($userData !== false) {
            return new JsonModel($wallData);
        } else {
            throw new \Exception('User not found', 404);
        }
    }
```

Creating the Url.php file

This is the file that will contain the custom validator we are going to create in order to validate the URLs provided by the users. ZF2 already provides a validator for URLs called `Zend\Validator\Uri` that you can configure with different options, but this is the perfect excuse to learn how to create and use custom validators.

```
namespace Users\Validator;

use Zend\Validator\AbstractValidator;
```

The code starts as usual by declaring the current namespace and also, in this case, specifying that we want to use the component `AbstractValidator`. In order to create a custom validator, you should implement the `Zend\Validator\ValidatorInterface` interface or just extend `Zend\Validator\AbstractValidator` that already implements the interface we mentioned before and gives the benefit of using the validator in a validator chain or be used by `Zend\Filter\Input`.

```
class Url extends AbstractValidator
```

This is the class declaration on our file. After that we need to declare a constant that will be used by the message template to generate the errors while validating data.

```
const INVALID   = 'urlInvalid';

protected $messageTemplates = array(
    self::INVALID   => "Invalid url given"
);
```

This is the section related to error messages. For each error message you should define a constant and then inside `$messageTemplates` you should provide an error message associated with the error constant. This is a simple case, but if needed, you can create complex error messages that contain values or text based on the input given by the user, as follows:

```
public function isValid($value)
{
    if (!is_string($value)) {
        $this->error(self::INVALID);
        return false;
    }

    $this->setValue($value);
    if(!filter_var($value, FILTER_VALIDATE_URL)) {
        $this->error(self::INVALID);
        return false;
    }

    return true;
}
```

This is the `isValid()` method that will be called when we validate data. The method only has one parameter and will be populated with the value we want to validate. The first thing we check is the type of the value. In the case that we find something that is not a string, we record the error using the `error()` method of the class and then return `false`.

If the value is a string, we set the value in the object using the `setValue()` method and then leverage the actual check to the PHP method `filter_var()`, passing the value as the first parameter and the constant `FILTER_VALIDATE_URL` as the second parameter. We will test the value returned by this method; in case of failure we will record the error as we did before and return `false`. In case of success we just return `true`.

As you can see, it is really easy to create custom validators when needed and the complexity can go from simple checks like this one to complex things such as requesting things to DNS servers or connecting to other systems to check data.

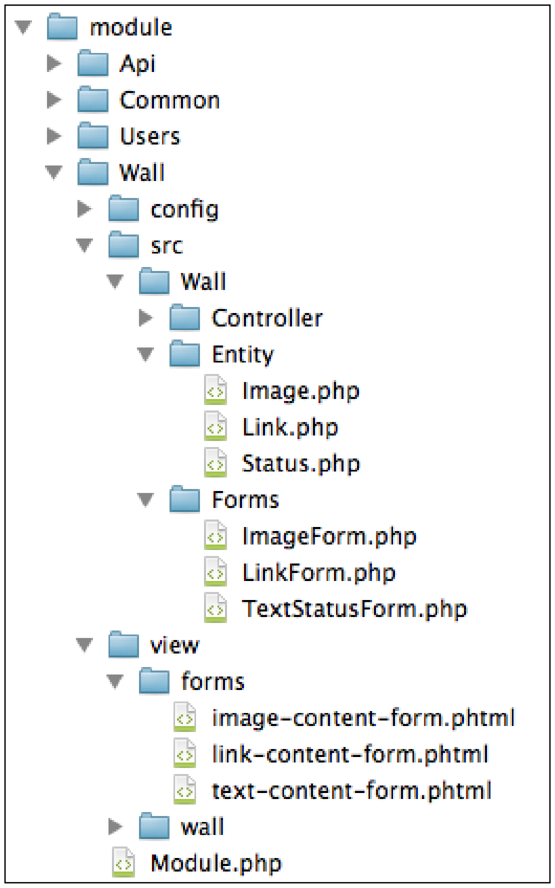
Modifying the module.config.php file

As we did in the previous chapters, we should add the following line to the dependency injector configuration of the `Users` module.

```
    'Users\Model\UserImagesTable' => 'Users\Model\UserImagesTable'
```

Frontend

Now we have all the required code on the API to accept URLs. Let's see the changes required on the frontend to be able to post the URL and show the result on the wall. As usual, let's have a look at a screenshot of the folder structure and new files.



```
module
  Api
  Common
  Users
  Wall
    config
    src
      Wall
        Controller
        Entity
          Image.php
          Link.php
          Status.php
        Forms
          ImageForm.php
          LinkForm.php
          TextStatusForm.php
    view
      forms
        image-content-form.phtml
        link-content-form.phtml
        text-content-form.phtml
      wall
    Module.php
```

Creating the link entity

Let's create a new entity to store the data related to a link. This class is really easy and just contains getters and setters for the column names. Let's see a couple of examples and the list of properties contained inside the class.

```
protected $id = null;
protected $userId = null;
protected $url = null;
protected $title = null;
   protected $createdAt = null;
protected $updatedAt = null;
```

Now let's see two simple getters and setters.

```
public function setTitle($title)
{
    $this->title = $title;
}

public function setUpdatedAt($updatedAt)
{
    $this->updatedAt = new \DateTime($updatedAt);
}

public function getId()
{
    return $this->id;
}

public function getUserId()
{
    return $this->userId;
}
```

Adding the LinkForm.php file

Of course if we want the users to use the new functionality, we should provide them with a way to provide us with the information we need. This form will allow them to specify the URL of the web page they want to share, as simple as that. Again, this form will have one special section, the field used for the URL information.

```
namespace Wall\Forms;

use Zend\Form\Element;
use Zend\Form\Form;
```

As usual we start with the following declaration of namespaces at the beginning of the class:

```
class LinkForm extends Form
```

The declaration of the class will follow the same approach as the other forms we already created, but in this case we are not implementing the `InputFilterProviderInterface` interface because the field we are going to use will automatically add the needed validators.

```
public function __construct($name = null)
{
    parent::__construct('link-content');
```

```
        $this->setAttribute('method', 'post');
        $this->setAttribute('class', 'well input-append');

        $this->prepareElements();
    }
```

The structure of the form will follow the approach of having specialized methods for each task. That leaves us with the following constructor:

```
    public function prepareElements()
    {
        $this->add(array(
            'name' => 'url',
            'type'  => 'Zend\Form\Element\Url,
            'attributes' => array(
                'class' => 'span11',
            ),
        ));
        $this->add(new Element\Csrf('csrf'));
        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type'  => 'submit',
                'value' => 'Submit',
                'class' => 'btn'
            ),
        ));
    }
```

This is how the `prepareElements()` method looks. As you can see we are adding the field URL to allow the users to insert the URL and is a `Url` element. This includes the new URL HTML5 element on the form and automatically adds the `Zend\Validate\Uri` validator. Remember that if we use the `Zend\Form\Element\Url` element we should use the `formUrl()` helper on the view.

Creating the image-content-form.phtml file

This file will take care of rendering the fields associated with the **LinkForm**. The content is similar to other files we saw before.

```
    <?php $form = $this->form; ?>
    <?php $form->prepare(); ?>
    <?php echo $this->form()->openTag($form) ?>
```

```
<!-- Url input text -->
<?php echo $this->formElement($form->get('url')); ?>
<?php echo $this->formElementErrors($form->get('url')); ?>

<!-- Submit button -->
<?php echo $this->formElement($form->get('submit')); ?>

<!-- CSRF -->
<?php echo $this->formElement($form->get('csrf')); ?>
<?php echo $this->formElementErrors($form->get('csrf')); ?>

<?php echo $this->form()->closeTag() ?>
```

Did you see it? Yes, we are not using the `formUrl()` helper, instead, we are using the usual `formElement()` helper. Why? Because the `formElement()` helper will introspect the field we are trying to print and will decide which is the best helper for the job and will use it. It's like a universal view helper for form elements.

Extending the index.phtml file

As usual we cannot put the entire code of the view because it's huge but we are going to see in this section how to include the new form on the view.

```
<div id="wall-link">
    <?php echo $this->partial(
        'forms/link-content-form.phtml',
        array('form' => $linkContentForm)
    ); ?>
</div>
```

As you can see, we are calling a partial that includes the elements of the form and we are passing the form itself as a parameter to that partial. When you render a partial you can pass all the variables you want and change its name using the second parameter of the `partial()` method. It accepts an array and the names you use on the keys will become the name of the variable that contains the value you pass inside the partial.

```
<?php elseif ($entry instanceOf \Wall\Entity\Link) : ?>
    <p>
        <a href="<?php echo $entry->getUrl() ?>">
            <?php echo $entry->getTitle() != ''?
                $entry->getTitle() : $entry->getUrl(); ?>
        </a>
    </p>
<?php endif; ?>
```

This is the code we will use to represent a link on the wall. It's a simple link pointing to the URL and using the title of the website as a text of the link. It's really simple but you can extend the base system to fetch the images of the web page and use one of them to represent the page similar to Facebook. You can also extract the content of the `description` meta tag and use it here to further explain the purpose of the website.

Changing the User.php file

As we did with the other content, we should adapt the `User` entity to create new entities for the links. This is really easy and we just need to add the following lines to the `setFeed()` method:

```
public function setFeed($feed)
{
    $hydrator = new ClassMethods();

    foreach ($feed as $entry) {
        if (array_key_exists('status', $entry)) {
            $this->feed[] = $hydrator->hydrate(
                $entry, new Status()
            );
        } else if (array_key_exists('filename', $entry)) {
            $this->feed[] = $hydrator->hydrate(
                $entry, new Image()
            );
        } else if (array_key_exists('url', $entry)) {
            $this->feed[] = $hydrator->hydrate(
                $entry, new Link()
            );
        }
    }
}
```

As you can see we are adding a new case to our `if` block and testing if the entry has a key called `url`, in which case we proceed to hydrate a new `Link` object.

Modifying the IndexController.php file

The following is the last file we have to modify in this chapter to make the new functionality work:

```
use Wall\Forms\LinkForm;
```

This is the first thing to do. We should add this line to be able to use the new form we created for the users.

Then we should make three changes in the `indexAction()` method in order to include and process the new form.

```
$linkForm = new LinkForm();
```

This line goes in the block where we create the other forms; we just need to create an instance of the new one.

```
if (array_key_exists('url', $data)) {
    $result = $this->createLink($linkForm, $user, $data);
}
```

This second block goes inside the `if` block that checks if we are dealing with a POST request. In here, we are checking the type of the request based on the keys found in the data and based on that call, the specialized method `createLink()`, which will take care of the rest. The rest of the code should look familiar because it is the same one we used on the two previous forms but we have just adapted the names of the variables to match the new forms.

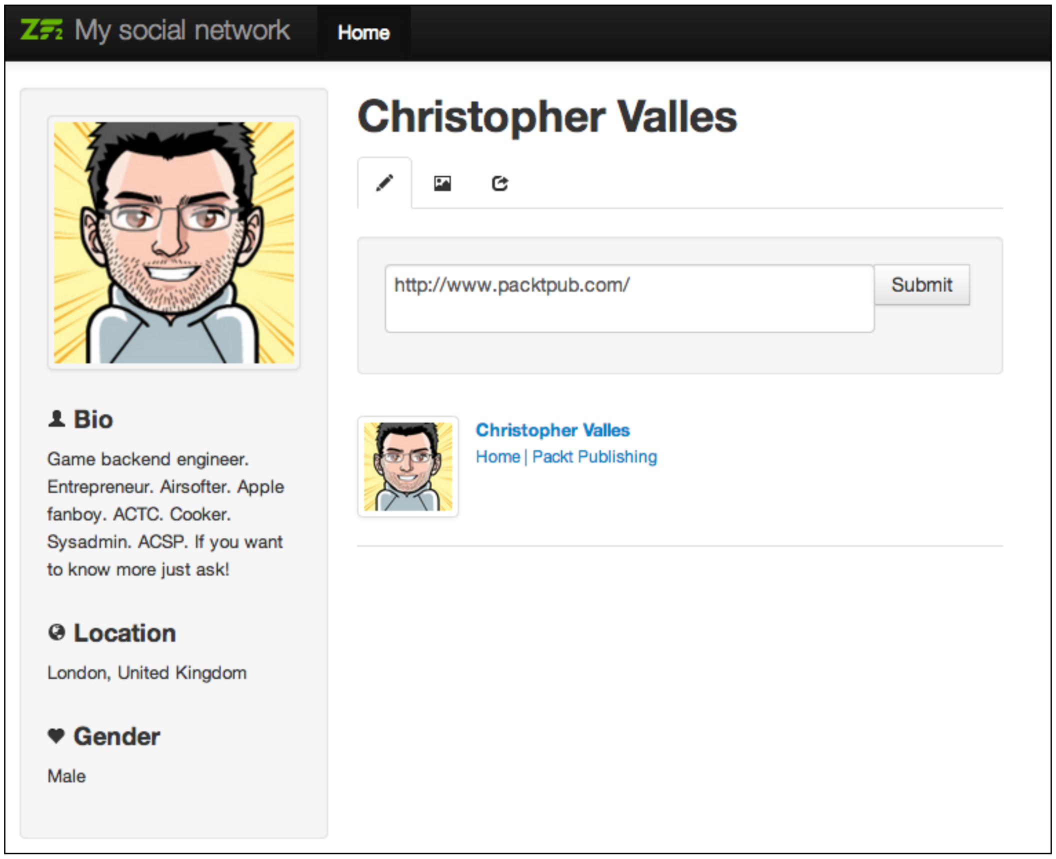
```
$linkForm->setAttribute(
    'action',
    $this->url()->fromRoute(
        'wall',
        array('username' => $user->getUsername())
    )
);
```

This is the last block of code we need in this method and we will add it at the end, next to the other two similar blocks of code we have. We are just setting the URL for the `action` parameter on the form.

Now let's review the new method `createLink()` that we have to add. It's really simple and looks like the other one we already added, specially with the `createStatus()` method that is basically the same.

```
protected function createLink($form, $user, array $data)
{
    return $this->processSimpleForm($form, $user, $data);
}
```

The following screenshot shows how our wall looks with the new functionality:



Christopher Valles

http://www.packtpub.com/

Christopher Valles
Home | Packt Publishing

Bio
Game backend engineer. Entrepreneur. Airsofter. Apple fanboy. ACTC. Cooker. Sysadmin. ACSP. If you want to know more just ask!

Location
London, United Kingdom

Gender
Male

Summary

Now users are able to share their favorite websites on their wall! That's the last type of content they can publish on their walls.

In this chapter we covered how to deal with URLs on forms using the new element provided by ZF2. We also saw how to create custom validators that can be used to validate anything you need!. Finally, we saw how to crawl other web pages in order to extract the content we need.

In the next chapter we will see how to allow users to add comments to the wall entries and how to deal with possible spam.

8
Dealing with Spam – Akismet to the Rescue

Before moving to another section of the social network, we still have something to add—comments. In this chapter, users will be able to comment on the entries made on the wall.

By the end of this chapter, you will know how to use the `ZendService` components to get extra functionality. We are going to use `ZendService\Askismet` to protect our service from spammers, and we will check the contents of each comment on the API side to validate it against the **Akismet** servers.

API development

In this code, we will need to follow the same approach used in all the chapters related with the user wall. We need to create a new table to hold the data, then create the table gateway to be able to access the data, and finally modify the controller to add the new type of content to the `create()` method.

Requirements

We are not going to modify the way API works or the way developers interact with the API. This means that we will keep using the same HTTP methods as before and will modify the `create()` method to detect new content we want to add, and redirect the job to the proper specialized method.

Working with the database

Let's take a look at the structure of the new table we have to create to store the comments of the users.

This table is named `user_comments`, and will hold the following data:

- Id
- User_id
- Type
- Entity_id
- Comment
- Created at
- Updated at

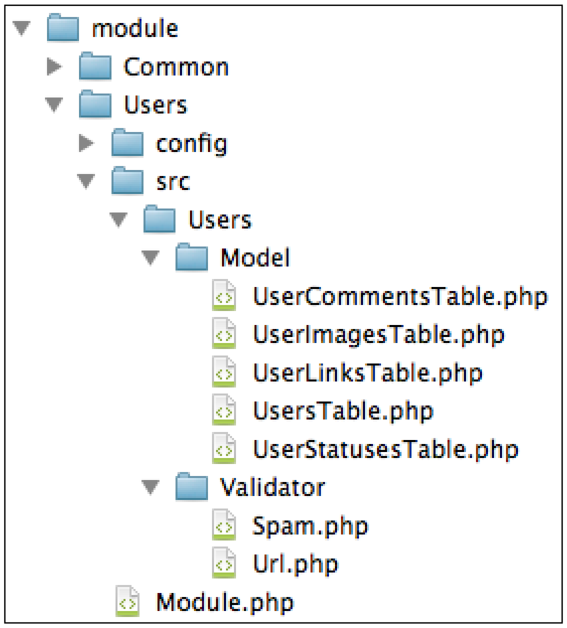
Also, in this table, as with all the ones we already created, we will add two columns to store the time where a record is created and updated.

The following is the statement to create the new table:

```
CREATE TABLE `user_comments` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(11) unsigned DEFAULT NULL,
  `type` tinyint(1) unsigned DEFAULT NULL,
  `entry_id` int(11) unsigned DEFAULT NULL,
  `comment` varchar(255) DEFAULT NULL,
  `created_at` timestamp NULL DEFAULT NULL,
  `updated_at` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_user_id` (`user_id`),
  KEY `idx_entry_id` (`entry_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Module structure

Before we start describing the code, let's also look at the following folder structure and the new files we will create to build this functionality:



```
module
  Common
  Users
    config
    src
      Users
        Model
          UserCommentsTable.php
          UserImagesTable.php
          UserLinksTable.php
          UsersTable.php
          UserStatusesTable.php
        Validator
          Spam.php
          Url.php
    Module.php
```

Installing Akismet service

In order to use the Akismet service we should install the dependencies in our code. As ZF2 uses composer, this will be an easy task. Let's see the changes we have to do in the `composer.json` file to install Akismet. Alternatively, you can also use the code provided for this chapter that has Akismet already installed.

```
"repositories": [
    {
        "type": "composer",
        "url": "https://packages.zendframework.com/"
    }
],
```

Here, we are specifying the repository where the packages for Zend Framework can be found. After that we have to specify the package we want by adding the following line in the required block:

```
"zendframework/zendservice-akismet": "2.*@dev"
```

After that, we should run the composer to update the dependencies that are accomplished executing the following command:

```
php composer.phar update
```

This command should be executed in the root folder of the API project.

Modifying the local.php file

Now that we have the Akismet service installed, we need to store our API key somewhere to be able to send it when we request it to check if a comment is spam or not. Let's add a new section in the `local.php` file to store this, as follows:

```
'akismet' => array(
    'apiKey' => 'YOUR API KEY HERE',
    'url' => 'THE URL OF YOUR SERVICE HERE
)
```

Adding the UserCommentsTable.php file

As usual when we create a new table, we should also create the table gateway object that will allow us to work with the data stored in the database. This time, the file will be bigger than usual because we will add the validators to check if the comment is spam or not. As we already created a few classes similar to this one, we will just review the queries we provided and the configuration of the filters. For the full source code refer to the code of this chapter.

```
public function create($userId, $type, $entryId, $comment)
{
    return $this->insert(array(
        'user_id' => $userId,
        'type' => $type,
        'entry_id' => $entryId,
        'comment' => $comment,
        'created_at' => new Expression('NOW()'),
        'updated_at' => null
    ));
}
```

This is the method that will allow us to insert new comments in the table; `entry_id` refers to the ID of the content we are commenting and the type refers to the type of content.

```
public function getByTypeAndEntryId($type, $entryId)
{
    $select = $this->sql->select()->where(array(
        'type' => $type,
        'entry_id' => $entryId)
    )->order('created_at ASC);
    return $this->selectWith($select);
}
```

This method will allow us to retrieve comments of a specific entry; we use the type of entry and `entry_id` to retrieve related comments, if any. Also, you can see that we order the rows by creation date with the new comments first.

Let's discuss now how to configure the validators for the comment. The three important checks we want to do are as follows:

- The user making the comments exists in the database
- The entry we are commenting on exists in the database
- The comment is not considered spam

We will do the first two checks against the database using the `RecordExists` validator; for the last check we will use a custom validator, which we will create.

Let's see the following validator and filter configuration for each field. As the method is big, refer to the code of this chapter to see the entire code:

```
    $inputFilter->add($factory->createInput(array(
        'name'     => 'user_id',
        'required' => true,
        'filters'  => array(
            array('name' => 'StripTags'),
            array('name' => 'StringTrim'),
            array('name' => 'Int'),
        ),
        'validators' => array(
            array('name' => 'NotEmpty'),
            array('name' => 'Digits'),
            array(
                'name' => 'Zend\Validator\Db\RecordExists',
                'options' => array(
                    'table' => 'users',
                    'field' => 'id',
                    'adapter' => $this->adapter
                )
            )
        ),
    )));
```

This block is the configuration for the `user_id` field as usual we remove the possible HTML tags, remove whitespaces at the beginning and end of the string, and we make sure it's a number. To validate the field we check that it is not empty, is a digit, and the record exists inside the `users` table.

```
$inputFilter->add($factory->createInput(array(
    'name'     => 'type',
    'required' => true,
    'filters'  => array(
        array('name' => 'StripTags'),
        array('name' => 'StringTrim')
    ),
    'validators' => array(
        array('name' => 'NotEmpty'),
        array('name' => 'Digits'),
    ),
)));
```

The type field is easier, as we just want to filter with the usual suspects, `StripTags` and `StringTrim`, and then we validate with `NotEmpty`, making sure we are dealing with a digit.

```
$inputFilter->add($factory->createInput(array(
    'name'     => 'entry_id',
    'required' => true,
    'filters'  => array(
        array('name' => 'StripTags'),
        array('name' => 'StringTrim')
    ),
    'validators' => array(
        array('name' => 'NotEmpty'),
        array('name' => 'Digits'),
        array(
            'name' => 'Zend\Validator\Db\RecordExists',
            'options' => array(
                'table' => $validatorTable,
                'field' => 'id',
                'adapter' => $this->adapter
            )
        )
    ),
)));
```

The configuration for `entry_id` is exactly the same as the one we created for the user with only one difference. The table we will use to check if the row exists is now a variable and will be configured based on the data we get from the client.

```
$inputFilter->add($factory->createInput(array(
    'name'     => 'comment',
    'required' => true,
    'validators' => array(
        array('name' => 'NotEmpty'),
        array(
            'name' => '\Wall\Validator\Spam',
            'options' => array(
                'apiKey' => $config['apiKey'],
                'url' => $config['url']
            )
        ),
    ),
)));
```

This is the last field we check. As you can see we do not filter the data, because it's an array created on the controller that will be validated directly with `Spam`.

Creating the Spam.php file

Now, let's create the validator that will use Akismet to check if a comment is considered spam or not. This time we will create a custom validator that will accept the configuration. With this knowledge, you will be able to create any validator that you may or may not need with the configuration.

```
namespace Users\Validator;

use Zend\Validator\AbstractValidator;
use ZendService\Akismet\Akismet;
```

We start, as usual, by defining the namespace and the components we want to use in this class; after that we define the actual class.

```
class Spam extends AbstractValidator
```

Now we need to define some constants, one for each error condition we can find on the validator.

```
const INVALID = 'invalid';
const SPAM = 'isSpam';
```

The first two constants refer to error conditions while validating the data, and the last two are errors that can appear when you configure the validator.

```
protected $messageTemplates = array(
    self::INVALID => "Invalid input",
    self::SPAM => "The text seems to be spam"
);
```

As you see, we also need to define an error message attached to each error constant we defined before.

```
protected $options = array(
    'apiKey' => null,
    'url' => null
);
```

This variable holds the available configurations for the validator. In here, we have to define all the possible variables the validator needs to do the job, and will be populated by the parent constructor using the setter methods.

```
public function __construct($options = array())
{
    if (!is_array($options)) {
        $options = func_get_args();
        $temp['apiKey'] = array_shift($options);
        if (!empty($options)) {
            $temp['url'] = array_shift($options);
        }

        $options = $temp;
    }

    parent::__construct($options);
}
```

This is the constructor of the validator. We check how the configuration data has been passed to the validator and act accordingly. At the end, we call the parent constructor passing the configuration data as an array.

```
public function getApiKey()
{
    return $this->options['apiKey'];
}

public function setApiKey($apiKey)
```

```
    {
        if (empty($apiKey)) {
            throw new \Exception('API key cannot be empty');
        }

        $this->options['apiKey'] = $apiKey;
        return $this;
    }
```

This is the setter and getter for the `apiKey` configuration option in the setter; we check that the value is not empty, as the two configuration values are mandatory.

```
    public function getUrl()
    {
        return $this->options['url'];
    }

    public function setUrl($url)
    {
        if (empty($url)) {
            throw new \Exception('The url cannot be empty');
        }

        $this->options['url'] = $url;
        return $this;
    }
```

This is a copy of the getter and setter shown before, but in this case for the `url` option.

Now let's see the `isValid()` method that will use the configuration to connect to Akismet using the new Akismet service we added, and validate the comment.

```
    public function isValid($value)
    {
        if (!is_array($value)) {
            $this->error(self::INVALID);
            return false;
        }

        $this->setValue($value);
        $akismet = new Akismet($this->getApiKey(), $this->getUrl());
        if (!$akismet->verifyKey($this->getApiKey())) {
            throw new \Exception('Invalid API key for Akismet');
        }
        if ($akismet->isSpam($value)) {
            $this->error(self::SPAM);
            return false;
        } else {
            return true;
        }
    }
```

This code will create a new instance of Akismet, then will verify the API key we provide, and finally check if the comment is considered spam or not.

Extending the IndexController.php file

In the controller, we have to modify two sections. The first is the one that processes an incoming request to save the comment and the second is the one that requests to get a user's wall. Let us see how to process the first request, validate that it is not spam, and store it.

We need to add a new check in the `create()` method to distinguish if the request is related with the creation of a comment.

At the end of the method we will add the following check:

```
if (array_key_exists('comment', $data) && !empty($data['comment'])) {
    $result = $this->createComment($data);
}
```

As you can see the check is similar to the ones we had before. Now we need to create the `createComment()` method that will take care of the actual processing of the comment.

The first block is as follows:

```
$userCommentsTable = $this->getUserCommentsTable();
$usersTable = $this->getUsersTable();
$user = $usersTable->getById($data['user_id']);
```

Here we are just getting two table gateways and retrieving the user who's commenting.

```
$data['comment'] = array(
    'user_ip' => $this->getRequest()->getServer('REMOTE_ADDR'),
    'user_agent' => $this->getRequest()->getServer(
        'HTTP_USER_AGENT'
    ),
    'comment_type' => 'comment',
    'comment_author' => sprintf(
        '%s %s',
        $user->name,
        $user->surname
    ),
    'comment_author_email' => $user->email,
    'comment_content' => $data['comment']
);
```

This is the array the Akismet service expects on their API level. Basically, we are providing them with the IP address of the user who's creating the comment, the user agent, a parameter to specify the type of content we are checking, the full name of the author of the comment, the e-mail, and the content itself.

```
    switch ($data['type']) {
        case \Users\Model\UserstatusesTable::COMMENT_TYPE_ID:
            $validatorTable =
                \Users\Model\UserstatusesTable::TABLE_NAME;
            break;
        case \Users\Model\UserImagesTable::COMMENT_TYPE_ID:
            $validatorTable =
                \Users\Model\UserImagesTable::TABLE_NAME;
            break;
        case \Users\Model\UserLinksTable::COMMENT_TYPE_ID:
            $validatorTable =
                \Users\Model\UserLinksTable::TABLE_NAME;
            break;
    }
```

Then we need to configure the validator of the entity to check that it exists in the database if you remember we were using a variable instead of the table name, and here is where we determine the table name that should be used. On the client side, we will send a type number that identifies the type of content on which we are commenting. We will use this value to determine what table to pass in the `getInputFilter()` function, which in turn will be used to validate that the status being commented on exists in the database.

```
    $config = $this->getServiceLocator()->get('Config');
    $filters = $userCommentsTable->getInputFilter(
        $validatorTable,
        $config['akismet']
    );
    $filters->setData($data);
```

Here we are getting the configuration where we have stored the details we need for the Akismet service. We are also passing the table name for the entry validation and the Akismet configuration to the method that returns the input filter. After that we set the data in the filters to be able to perform the validation.

```
if ($filters->isValid()) {
    $data = $filters->getValues();

    $result = new JsonModel(array(
        'result' => $userCommentsTable->create(
            $data['user_id'],
            $data['type'],
            $data['entry_id'],
            $data['comment']['comment_content']
        )
    ));
} else {
    $result = new JsonModel(array(
        'result' => false,
        'errors' => $filters->getMessages()
    ));
}

return $result;
```

This is the last block of the `createComment()` method and where we actually validate the data. In case of success, we retrieve the data from the filter, we then insert it in the database using the table gateway returning the results to the client. Otherwise, we return the error with an error message specifying what happened in the validation.

In order for this method to work, we also need to create a new property in this class, and a method called `getUserCommentsTable()` that will create and store a local copy of the table gateway.

```
protected $userCommentsTable;

protected function getUserCommentsTable()
{
    if (!$this->userCommentsTable) {
        $sm = $this->getServiceLocator();
        $this->userCommentsTable = $sm->get(
            'Users\Model\UserCommentsTable'
        );
    }
    return $this->userCommentsTable;
}
```

We also need to modify the table gateways and to assign to each one a type to be able to check the content on which we are commenting, but that will be done later. Now, let's review the changes we should make in the `get()` method to retrieve the content of the wall, including the comments for each one.

```
$userCommentsTable = $this->getUserCommentsTable();
```

We need to add this line to the first block to get an instance of the comments table gateway.

```
$userStatuses = $userStatusesTable->getByUserId($userData->id)
    ->toArray();
$userImages = $userImagesTable->getByUserId($userData->id)
    ->toArray();
$userLinks = $userLinksTable->getByUserId($userData->id)
    ->toArray();
```

We already had these lines before, the only change we made was adding a call at the end to retrieve the data as arrays. This in turn helped us while processing the entries to get the comments.

```
$allEntries = array(
    \Users\Model\UserstatusesTable::COMMENT_TYPE_ID =>
        $userStatuses,
    \Users\Model\UserImagesTable::COMMENT_TYPE_ID =>
        $userImages,
    \Users\Model\UserLinksTable::COMMENT_TYPE_ID =>
        $userLinks
);
```

This array will hold the entries of each content type, and then we will use this array to merge the contents in one array that will be ordered.

```
$cachedUsers = array();
foreach ($allEntries as $type => &$entries) {
    foreach ($entries as &$entry) {
        $comments = $userCommentsTable->getByTypeAndEntryId(
            $type,
            $entry['id']
        );

        if (count($comments) > 0) {
            foreach ($comments as $c) {
                if (array_key_exists($c->user_id, $cachedUsers)) {
                    $user = $cachedUsers[$c->user_id];
                } else {
                    $user = $usersTable->getById($c->user_id);
                    $cachedUsers[$c->user_id] = $user;
                }

                $entry['comments'][] = array(
                    'id' => $c->id,
                    'user' => $user,
                    'comment' => $c->comment
                );
            }
        }
    }
}
```

This block of code is responsible for getting comments of each content. As you can see we iterate through the array we just created, and then query the database. If we find comments, we iterate over them to get a copy of the data about the author of the comment to be able to display that data on the client. As we can have multiple comments from the same user, we create an array, of already retrieved users to avoid getting the same data from the database over and over again.

```
    $wallData = $userData->getArrayCopy();
    $wallData['feed'] = call_user_func_array(
        'array_merge',
        $allEntries
    );
```

These are the last two lines we need to modify. As we have the different entries under different keys in the `$allEntries` array we need to merge them to have one array with all the entries together.

Table gateways

As we mentioned before, in order to be able to detect the content type a user is commenting, we have to modify the table gateways to hold a unique type identifier on each one. Check the following table to see the relation between each type and the type assigned.

Type assigned	Content
1	Statuses (text)
2	Images
3	Links

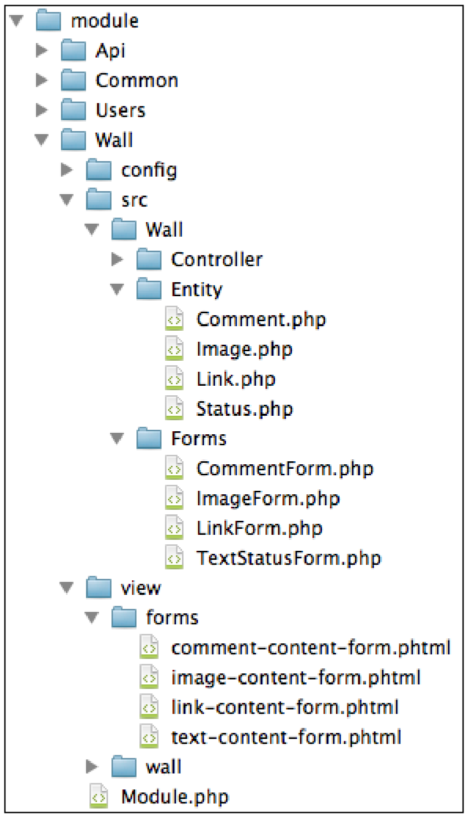
Now we have to modify the files to store this data let's see the following change we have to do:

```
const COMMENT_TYPE_ID = 1;
const TABLE_NAME = 'user_statuses';
```

These are the new lines for the `UserStatusesTable.php` file; the lines will be the same in `UserImagesTable.php` and `UserLinksTable.php`, the only difference will be the number stored in the constant `COMMENT_TYPE_ID` and the name of the table stored in `TABLE_NAME`.

Frontend

Now that we have our API sending back the comments for each entry on the user's wall, we need to adapt the client code to do two things. First we create a new form and put it beneath each entry so that users can comment on stuff. The second thing we have to do is show the comments made by users beneath each entry. Let's see the following folder structure for the client code:



```
module
  Api
  Common
  Users
  Wall
    config
    src
      Wall
        Controller
        Entity
          Comment.php
          Image.php
          Link.php
          Status.php
        Forms
          CommentForm.php
          ImageForm.php
          LinkForm.php
          TextStatusForm.php
    view
      forms
        comment-content-form.phtml
        image-content-form.phtml
        link-content-form.phtml
        text-content-form.phtml
      wall
    Module.php
```

Adding the Comment.php file

As usual we need to create an entity to store the data related with a comment. The entity itself is really simple, just a bunch of getters and setters. The only difference now is that we need to store a user entity inside the comment entity that will be the author data.

```
namespace Wall\Entity;

use Zend\Stdlib\Hydrator\ClassMethods;
use Users\Entity\User;
```

These two lines are the namespace and the declaration of the `ClassMethod` hydrator we are going to use to populate a `User` entity.

```
class Comment
{
    protected $id = null;
    protected $user = null;
    protected $comment = null;
    protected $createdAt = null;
    protected $updatedAt = null;

    public function setUser($user)
    {
        $hydrator = new ClassMethods();

        $this->user = $hydrator->hydrate($user, new User());
    }

    // MORE GETTERS AND SETTERS HERE
}
```

This is important code of the entity, for brevity the rest of the getters and setters are removed here, but you can check the code of this chapter if you want to see the full class.

The `setUser()` method is responsible for populating a `User` instance with the `ClassMethod` hydrator.

Adapting the Status.php file

Now we need to modify the entities we had for each content type to hold an array of possible comments. We are also going to store the type of the content itself to be able to send it to the API while commenting.

```
use Zend\Stdlib\Hydrator\ClassMethods;
```

Again, as we have to hydrate the comments we declare `ClassMethods`.

```
const COMMENT_TYPE_ID = 1;
```

This is the constant that will hold the type of content as you can see, it is exactly the same as the one we have on the API side.

```
protected $comments = null;
```

We need to add a new property in the class to be able to hold the array of comments provided by the API.

```
public function setComments($comments)
{
    $hydrator = new ClassMethods();

    foreach ($comments as $c) {
        $this->comments[] = $hydrator->hydrate($c, new Comment());
    }
}
```

This is the setter for the comments; as you can see we iterate over the comments, if any, and populate a new `Comment` instance appending it to the `$comments` property.

```
public function getComments()
{
    return $this->comments;
}

public function getType()
{
    return self::COMMENT_TYPE_ID;
}
```

Finally, we add two getters to the class; the first will return the comments and the second, `COMMENT_TYPE_ID`, will hold the type for this specific content.

Modifying the Image.php and Link.php file

The changes we have to do in these two entities are exactly the same as the ones we made on the status, the only difference being the number stored in the constant, `COMMENT_TYPE_ID`, which will be a two for `Image.php` and a three for `Link.php`.

As you can see, we are duplicating the code in the entities and it makes sense to make the entities to extend from a base entity and put the common code in the base entity. This task can be done by you as an exercise.

Adding the CommentForm.php file

The form we need to allow the users to add comments is really simple. We are going to have a textbox where users will put the comment on which itself. We will also have a couple of hidden fields that will hold the type of content we are commenting, and the ID of the entry we are commenting. Thanks to the hidden fields, the API will be able to associate the comment with the corresponding entry.

The structure of the file is the same as the one we have on other forms, so we will jump straight to the code we have on the constructor to see the fields we have.

```
parent::__construct('comment-content');

$this->setAttribute('method', 'post');
$this->setAttribute('class', 'well input-append');

$this->add(array(
    'name' => 'comment',
    'type'  => 'Zend\Form\Element\Text',
    'attributes' => array(
        'class' => 'span11',
        'placeholder' => 'Write a comment here...'
    ),
));
$this->add(array(
    'name' => 'type',
    'type'  => 'Zend\Form\Element\Hidden',
));
$this->add(array(
    'name' => 'entry_id',
    'type'  => 'Zend\Form\Element\Hidden',
));
$this->add(array(
    'type' => 'Zend\Form\Element\Csrf'
));
$this->add(array(
    'name' => 'submit',
    'attributes' => array(
        'type'  => 'submit',
        'value' => 'Submit',
        'class' => 'btn'
    ),
));
```

As you can see, the form is really simple, and we just add the fields we mentioned before with the usual **Cross Site Request Forgery** (**CSRF**) protection and the submit button.

Adding the comment-content-form.phtml file

The content of the view for the form is similar to the one we saw before; you can see the following code of the partial view that will be inserted in `index.phtml` view to print the form:

```
<?php $form = $this->form; ?>
<?php $form->prepare(); ?>
<?php echo $this->form()->openTag($form) ?>
<!-- Comment input text -->
<?php echo $this->formElement($form->get('comment')); ?>

<!-- Submit button -->
<?php echo $this->formElement(
    $form->get('type')->setValue($type)
); ?>
<?php echo $this->formElement(
    $form->get('entry_id')->setValue($entryId)
); ?>
<?php echo $this->formElement($form->get('submit')); ?>

<!-- Error message -->
<?php echo $this->formElementErrors($form->get('comment')); ?>

<!-- CSRF -->
<?php echo $this->formElement($form->get('csrf')); ?>
<?php echo $this->formElementErrors($form->get('csrf')); ?>

<?php echo $this->form()->closeTag() ?>
```

Notice that we are setting the values of the hidden fields based in two variables, `$type` and `$entryId`, that we expect.

Modifying the IndexController.php file

The changes we have to do on the controller are simple and short. Let's review the changes in the `indexAction()` method.

```
use Wall\Forms\CommentForm;
```

First, we need to declare the usage of `CommentForm` by adding the following line at the top:

```
$commentForm = new CommentForm();
```

Then we need to create an instance of the form; we will do that in the block of code where we create the other forms.

```
if (array_key_exists('comment', $data)) {
    $result = $this->createComment($commentForm, $user, $data);
}
```

Now, in the section where we inspect the POST request, we need to add a new block to check if the user is trying to create a comment, and then redirect the request to the proper method.

```
case $result instanceOf CommentForm:
    $commentForm = $result;
    break;
```

When we process the form results, we need to add a new case in the `switch` statement to check the return value of the `createComment()` method.

```
$commentForm->setAttribute(
    'action',
    $this->url()->fromRoute(
        'wall',
        array('username' => $user->getUsername())
    )
);
```

This piece of code should be added next to the block where we set the attributes to the other forms; it is exactly the same but of different form.

```
$viewData['commentContentForm'] = $commentForm;
```

This is the final line we have to add in this method to pass the form to the view and be able to print it.

As this form is really simple, the `createComment()` method will call the `processSimpleForm()` parent method.

```
protected function createComment($form, $user, array $data)
{
    return $this->processSimpleForm($form, $user, $data);
}
```

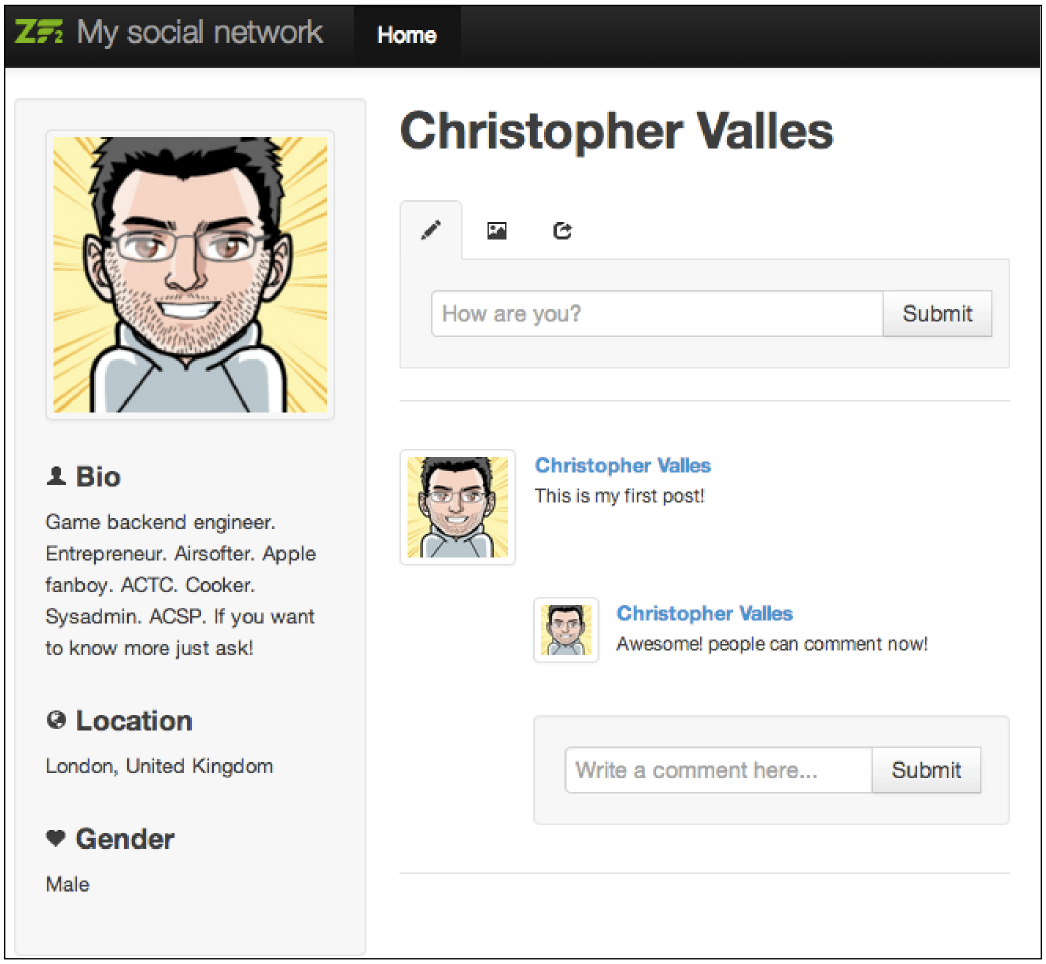
Extending the index.phtml file

This is the last file we have to modify to make everything work. We are going to add a block of code right after printing the content of each entry to print the comments, if any, and also to insert the comment form.

```
<?php if ($entry->getComments() !== NULL) : ?>
    <ul class="post-list">
        <?php foreach ($entry->getComments() as $i => $c) : ?>
            <li class="post">
                <section>
                    <a href="#" class="pull-left thumbnail">
                        <img src="#" />
                    </a>
                    <div class="post-body">
                        <h4 class="post-heading">
                            <a href="#">NAME OF USER</a>
                        </h4>
                        <p><?php echo $c->getComment() ?></p>
                    </div>
                </section>
            </li>
        <?php endforeach; ?>
    </ul>
<?php endif; ?>
<?php echo $this->partial(
    'forms/comment-content-form.phtml',
    array(
        'form' => $commentContentForm,
        'type' => $entry->getType(),
        'entryId' => $entry->getId())
); ?>
```

The code you can see here is a simplified version of the real one, otherwise it will not be readable. As you can see, if there are comments, we iterate over them and print each one putting the avatar of the user, hardcoded for now, and also the name of the user and the content of the comment. Finally, we call the partial that will render the comment form and pass the type of entry and the entry ID to populate it on the hidden fields.

The following screenshot shows how the wall looks like now:



Christopher Valles

How are you?

Bio

Game backend engineer. Entrepreneur. Airsofter. Apple fanboy. ACTC. Cooker. Sysadmin. ACSP. If you want to know more just ask!

Location

London, United Kingdom

Gender

Male

Christopher Valles
This is my first post!

Christopher Valles
Awesome! people can comment now!

Summary

We have finished the journey of the wall functionality; congratulations!

In this chapter we saw how to use external web services, such as Akismet, to further extend the possibilities of our system; in our case, creating a custom validator that uses the Akismet service to validate comments and determine if they are spam or not.

In the next chapter we will jump to adding external content to our website using RSS feeds and we will learn how to integrate them and also how to cache the requests.

9
Let's Read Feeds – A News Reader

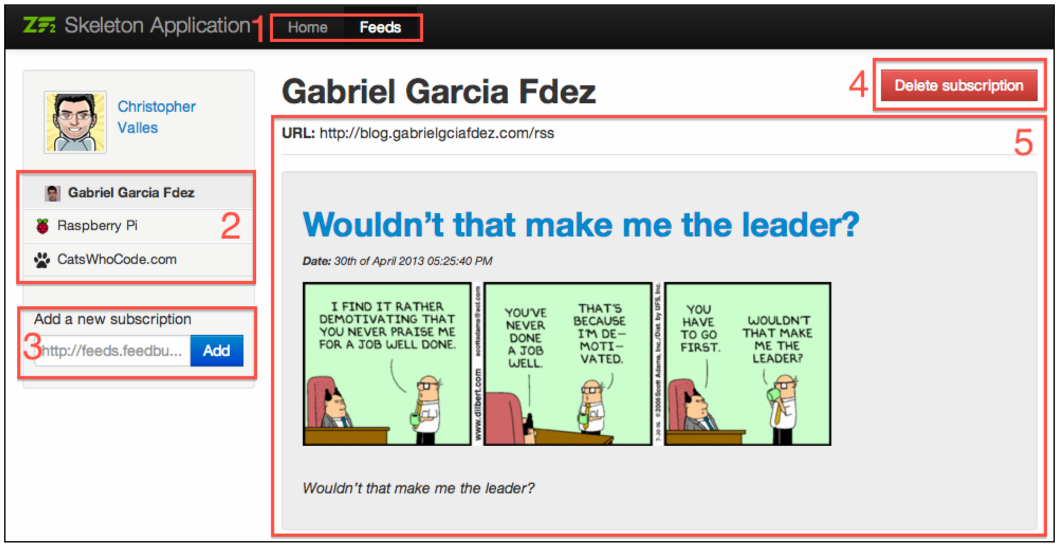
It's time to add a new, big functionality on our social network. Right now, users can post a variety of content and comment on each other's content. Of course, they cannot register themselves or log in to the page, but this problem will be fixed in the next few chapters. From the point of view of the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

Overview

On the application, the functionality will look similar to the following screenshot:



Gabriel Garcia Fdez

URL: http://blog.gabrielgciafdez.com/rss

Wouldn't that make me the leader?

Date: 30th of April 2013 05:25:40 PM

Wouldn't that make me the leader?

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:

HTTP method	**Controller method**	**Parameters**	**Functionality**

GET	`get()`
	`getList()`
		None	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.

POST	`create()`	data	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.
PUT	`update()`	ID	This method is not allowed

HTTP method	Controller method	Parameters	Functionality
DELETE	`delete()`	ID	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.

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:

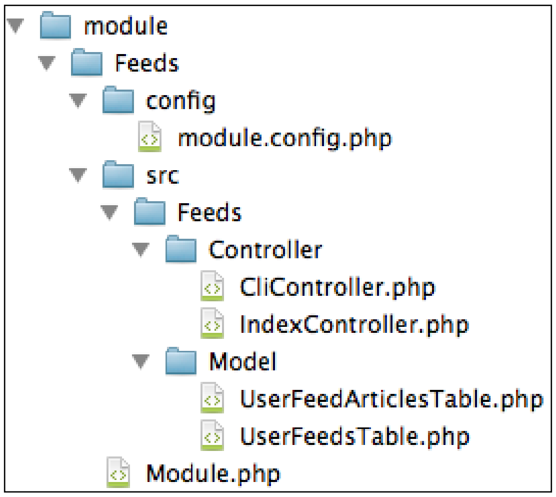
```
CREATE TABLE `user_feeds` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(11) unsigned DEFAULT NULL,
  `url` varchar(2048) DEFAULT NULL,
  `title` varchar(512) DEFAULT NULL,
  `icon` varchar(2048) DEFAULT NULL,
  `created_at` timestamp NULL DEFAULT NULL,
  `updated_at` timestamp 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(
                    '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
```

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:

```
'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.

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

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:

```
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`:

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

Additionally, we also use the following components:

```
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:

```
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:

```
$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:

```
$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:

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

```
$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 retriev 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();
```

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.

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

XPath is a query language that allow 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.

```
$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:

```
$rss = Reader::import($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:

```
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.

```
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.

```
$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.

```
$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:

```
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.

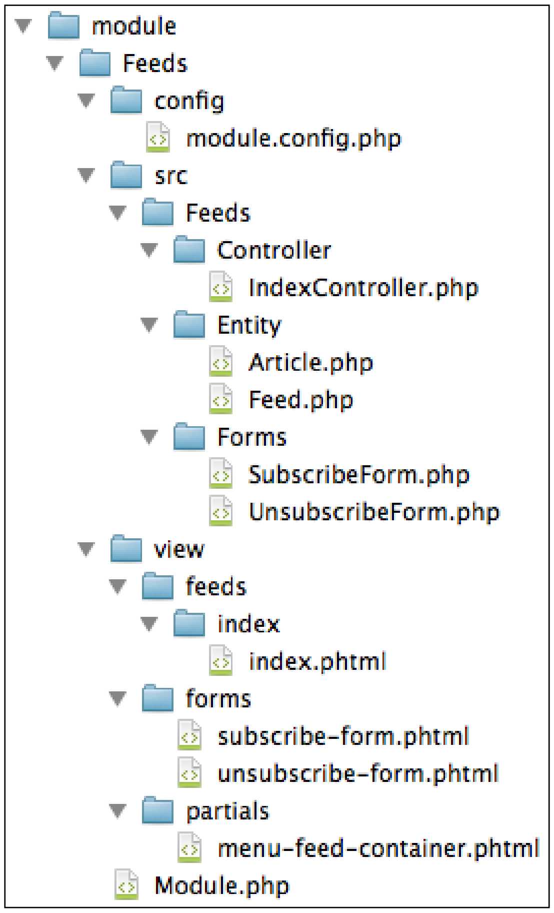
The ApiErrorListener.php file

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.

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

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:

```
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:

```
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:

```
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:

```
'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(
                    '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:

```
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:

```
$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;
}
```

```
$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) {
    $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;
    }
```

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:

```
$unsubscribeForm->get('feed_id')->setValue($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.

```
$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);
    }
```

```
        return $this->redirect()->toRoute(
            'feeds', array('username' => $username)
        );
    }
```

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:

```
$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:

```
<ul id="subscription-list">
    <?php foreach ($container->getPages() as $p) : ?>
        <li>
            <a href="<?php echo $p->getHref() ?>"
            class="<?php echo $p->isActive()? 'active' : '' ?>">
                <?php if ($p->get('icon') !== null) : ?>
                    <img src="<?php echo $p->get('icon') ?>" />
                <?php endif; ?>
                <?php echo $p->getTitle() ?>
            </a>
        </li>
    <?php endforeach; ?>
</ul>
```

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 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:

```
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.

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 $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.

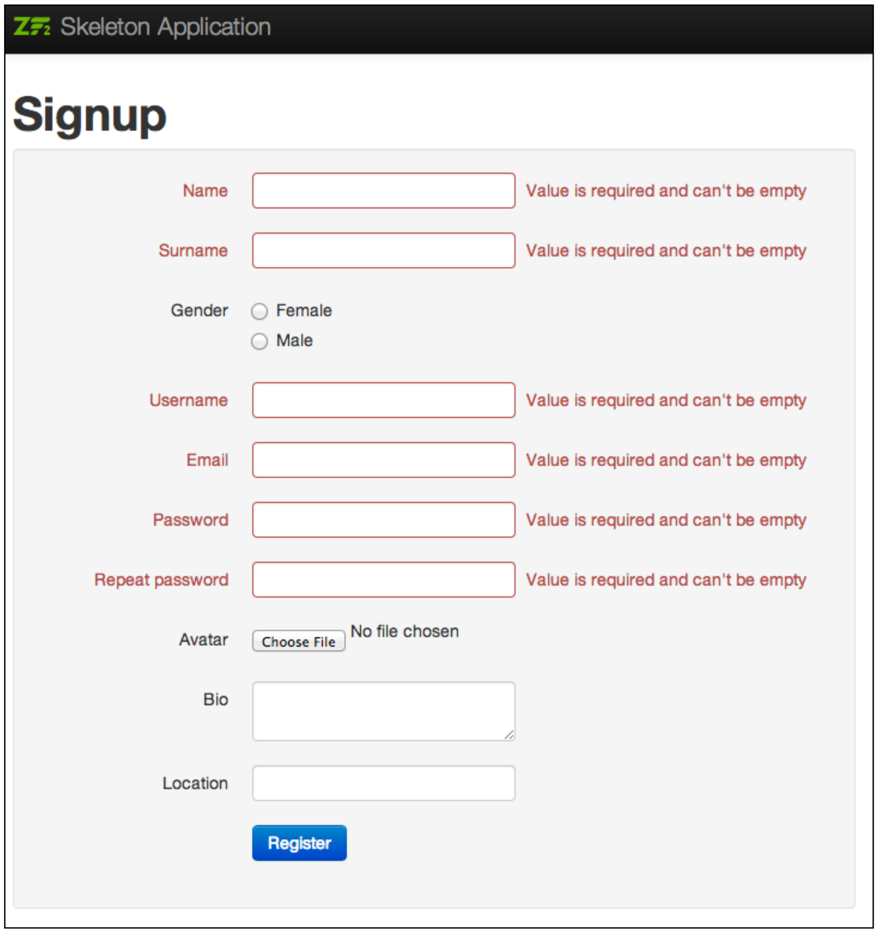
10
Sign Up

Now that we have all the basic features on our product, we need users to use them! In order to do that, we need to allow them to register and have an account on the social network.

Until now, all the forms we created are fairly simple, but by the end of this chapter, you will know how to deal with complex forms with the validation for each field, custom error messages, and its proper handling at the frontend. You will also learn how to use the `Zend\Crypt` component in order to encrypt the user's password in a really secure way. For the signup form, we will also see how to switch the default layout to another one when needed.

Overview

The signup form looks similar to the following screenshot, including the error messages:



Name		Value is required and can't be empty
Surname		Value is required and can't be empty
Gender	○ Female ○ Male	
Username		Value is required and can't be empty
Email		Value is required and can't be empty
Password		Value is required and can't be empty
Repeat password		Value is required and can't be empty
Avatar	No file chosen	
Bio		
Location		

API development

The tasks we have to do on this side of the project are very simple. We just need to accept the data sent at the frontend, validate it, and create the user on the database by first encrypting the password of the user. As we are going to allow the users to upload an avatar, we will also handle the processing of the image in order to show it everywhere.

Requirements

The requirements in the API level are as follows:

- Accept the `POST` request at the `/api/users[/:id]` endpoint
- Process the image and store it on the server
- Encrypt the password of the user before storing the data

Working with the database

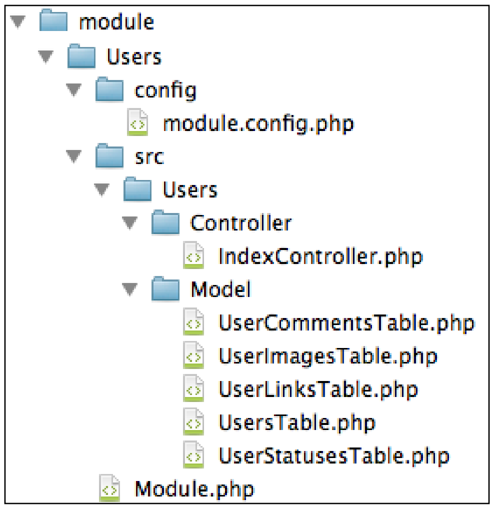
As we already have the `users` table, there is no need to create a table or modify it; we are going to use the same one. But before jumping to the next section and start coding, we need to talk about the `password` column in the `users` table.

If you notice, the `password` column is a `BINARY(60)` value. This is because the component we are going to use to encrypt the password is based on bcrypt, and will generate a 60-character binary string for each password we store.

bcrypt is a cipher based on the Blowfish cipher that contains a salt to protect against rainbow table attacks. Read more about it at: `http://en.wikipedia.org/wiki/Bcrypt`

The module structure

We are going to work in the `Users` module created in the first few chapters of the book, so we don't need to create new folders and files. We just need to modify the ones we already have. The folder structure is given in the following screenshot:



```
module
  Users
    config
      module.config.php
    src
      Users
        Controller
          IndexController.php
        Model
          UserCommentsTable.php
          UserImagesTable.php
          UserLinksTable.php
          UsersTable.php
          UserStatusesTable.php
    Module.php
```

Extending UsersTable.php

We need to extend this table gateway and add three new methods called `create()`, `updateAvatar()`, and `getInputFilter()`. Let's review them one by one.
The `create()` method is as follows:

```
public function create($userData)
{
    $userData['created_at'] = new Expression('NOW()');

    return $this->insert($userData);
}
```

If you noticed, we changed the definition of the `create()` method. Earlier, we were passing all the values we needed to create the row as a parameters to the method, and then assembling an array specifying all the data while performing the insert.

The signup form will contain a lot of fields and data. Having all the data as parameters will be impractical, and that's why in this method, we will receive an array containing the data needed to create the table already filtered and validated (this should include only the fields without any extra information; if not, the insert would fail). As you can see, the only modification we did is that we added the `created_at` value.

This gives us another way to manage the `create()` methods. In fact, we can move this generic code to a parent class and delete all the redundant code, allowing us to clean up and keep everything simple. At this point, feel free to modify the code of the project in order to achieve this behavior, if desired. The `updateAvatar()` method is as follows:

```
public function updateAvatar($imageId, $userId)
{
    return $this->update(array(
        'avatar_id' => $imageId
    ), array(
        'id' => $userId)
    );
}
```

The `updateAvatar()` method is the second method we have to create, which will help us in assigning an avatar to a user. The idea is that we need to create the user row first, and then upload the image specifying the `user_id` parameter, and then update the user to reflect the `avatar_id` parameter based on the image we just inserted.

This method essentially sets the `avatar_id` value of a row by first filtering the `id` attribute of the user.

Finally, we need to create the `getInputFilter()` method. Because the method is long and repetitive, I will just highlight the areas of interest for us and things we didn't cover before.

```
array(
    'name' => 'Zend\Validator\Db\NoRecordExists',
    'options' => array(
        'table' => 'users',
        'field' => 'username',
        'adapter' => $this->adapter
    )
)
```

The following code defines the validator we are going to use in the `username` field to validate that there is no other user with the same username.

```
array(
    'name' => 'EmailAddress',
),
array(
    'name' => 'Zend\Validator\Db\NoRecordExists',
    'options' => array(
        'table' => 'users',
        'field' => 'email',
        'adapter' => $this->adapter
    )
)
```

There are two validators that are used on the `email` address: the first one checks the format and validity of the e-mail address; the second one is used to confirm that there is no other user with the same e-mail address.

```
array(
    'name' => 'StringLength',
    'options' => array(
        'min' => 1,
        'max' => 50
    )
),
```

You will see the following type of validator everywhere, and it's used to check the minimum and maximum size of the data:

```
'validators' => array(
    array(
        'name' => 'InArray',
        'options' => array(
            'haystack' => array('0', '1')
        )
    ),
),
```

This is the last validator we are going to comment. This is used to check that the value provided belongs to one of the options we have set. In this case, the options are `0` or `1` (male, female), and the data sent by the client should only be `1` or `0`.

The IndexController.php file

In the controller, we are going to add a new method called `create()`, which will take care of getting the information from the client, storing the image when the user uploads an avatar, and creating the user account.

```
$usersTable = $this->getUsersTable();

$filters = $usersTable->getInputFilter();
$filters->setData($unfilteredData);

if ($filters->isValid()) {
    ...
} else {
    $result = new JsonModel(array(
        'result' => false,
        'errors' => $filters->getMessages()
    ));
}

return $result;
```

This is the skeleton of the method. As you can see, we get an instance of the users table gateway, and then we configure the filters and validators to test the data that comes as a parameter to the `create()` method. If the data is invalid, we return an error message with the specific reason about why it failed. If the data is correct, we proceed inside the `if` block, as given in the following code:

```
$data = $filters->getValues();

$avatarContent = array_key_exists('avatar', $unfilteredData) ?
```

```
        $unfilteredData['avatar'] : NULL;
    $bcrypt = new Bcrypt();
    $data['password'] = $bcrypt->create($data['password']);
```

The following is the first section of code executed after checking if the data is correct or not. What we do here is retrieve the filtered data, extract the avatar, and encrypt the password using the `Bcrypt` component:

```
    $user = $usersTable->getByUsername($data['username']);
    if ($usersTable->create($data)) {
        if (!empty($avatarContent)) {
            $userImagesTable = $this->getUserImagesTable();

            $filename = sprintf(
                'public/images/%s.png',
                sha1(uniqid(time(), TRUE))
            );
            $content = base64_decode($avatarContent);
            $image = imagecreatefromstring($content);

            if (imagepng($image, $filename) === TRUE) {
                $userImagesTable->create(
                    $user['id'], basename($filename)
                );
            }
            imagedestroy($image);

            $image = $userImagesTable->getByFilename(
                 basename($filename)
            );
            $usersTable->updateAvatar($image['id'], $user['id']);
        }

        $result = new JsonModel(array(
            'result' => true
        ));
    } else {
        $result = new JsonModel(array(
            'result' => false
        ));
    }
```

The first few lines retrieve the information from the filter and the possible image from the original data. The `$unfilteredData` array comes from the client and is passed to the `create()` method as the first parameter. After that, we encrypt the password of the user using the `Bcrypt` component of `Zend\Crypt\Password`. Encrypting something using the `Bcrypt` component is as easy as creating an instance and calling the `create()` method on the object, passing the data you want to encrypt as the first parameter. You may notice that right now, we are passing the data of the user to the server in plain text, which is, of course, a security breach. If you are creating a production-ready application, you may need to encrypt the data before sending it to the server and/or use the HTTPS protocol while sending sensitive data.

Once we have the password encrypted, we proceed to create the user account on the database and based on the success of that operation, we jump to process the avatar if we need to.

To process the image, we follow the same procedure as when the user posts an image on the wall. Of course, we are duplicating code here, and you can optimize it by calling a shared method with the wall controller to avoid duplication of code.

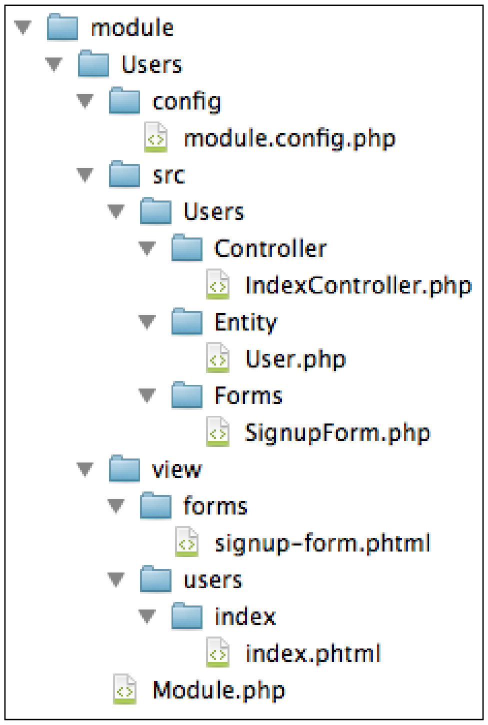
When the image is saved on the server and the data is stored on the database, we retrieve its ID from the database in order to update the `avatar_id` field on the user table for this specific user.

Finally, when the avatar has been updated as expected on the user account, we return `true` to the client.

Frontend

Let's review the work we need to do at the client side of the app. We need to create a new layout in order to remove the header menu. After that, we have to create the signup form and its view. Then, we should modify the controller to use the new layout and process the signup form. We also need to modify the controller view to show the signup form and finally adapt `ApiClient` to be able to send the data to the API.

The following screenshot shows the structure of the module:



```
module
  Users
    config
      module.config.php
    src
      Users
        Controller
          IndexController.php
        Entity
          User.php
        Forms
          SignupForm.php
    view
      forms
        signup-form.phtml
      users
        index
          index.phtml
    Module.php
```

The signup.phtml file

Let's start with the layout. The file is not shown in the screenshot, but it should be placed in the `Common` module next to the `layout.phtml` file. The contents of the file are the same as that of the `layout.phtml` file. The only difference is that we removed the following lines of code:

```
<?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 objective we want to achieve is a signup form without the navigation menu at the top. You can do this by using another layout or by creating a view helper that checks the route and based on that, shows or hides the menu.

The SignupForm.php file

The `SignupForm.php file` file is the file in which we are going to add all the fields for the signup form. Because the file is huge, we will look at the following relevant parts:

Field	**Type**
Username	`Text`
Email	`Text`
Password	`Password`
Repeat password	`Password`
Avatar	`File`
Name	`Text`
Surname	`Text`
Bio	`Textarea`
Location	`Text`
Gender	`Radio`

This is the list of fields we have on the form and their types. We need to highlight that we are also providing the `label` attribute and customizing the `label` class, as follows:

```
$this->add(array(
    'name' => 'location',
    'type'  => 'Zend\Form\Element\Text',
    'options' => array(
        'label' => 'Location',
        'label_attributes' => array(
            'class' => 'control-label'
        )
    )
));
```

This is the location field code. As you can see, the `label` attribute and class are specified inside the `options` key. Almost all the form view helpers allow us to specify their options in the following section of the field configuration:

```
$this->add(array(
    'name' => 'gender',
    'type'  => 'Zend\Form\Element\Radio',
    'options' => array(
        'label' => 'Gender',
        'label_attributes' => array(
            'class' => 'radio'
        ),
        'value_options' => array(
            0 => 'Female',
            1 => 'Male'
        )
    )
));
```

The preceding code is the code for the `Gender` field. As you can see, we specified the options for the radio buttons in the `value_options` key. This will also configure an `InArray` validator based on the values provided.

For the complete code, do not hesitate to jump to the file and see it yourself.

The signup-form.phtml file

As we are trying to make a most useful and customized form, we are going to review a few things on this file.

The way of creating a view is the same as the ones we saw before; the difference is that we now also add the HTML code in between and customize some HTML tag attributes based on the data provided by the PHP code and the fields.

The following structure is followed by all the fields:

```
    <div class="control-group <?php echo count($form->get('name')-
    >getMessages()) > 0 ? 'error' : '' ?>">
        <?php echo $this->formLabel($form->get('name')); ?>
        <div class="controls">
            <?php echo $this->formElement($form->get('name')); ?>
            <?php echo $this->formElementErrors()
                ->setMessageOpenFormat('<div class="help-inline">')
                ->setMessageSeparatorString(
                    '</div><div class="help-inline">'
                )
                ->setMessageCloseString('</div>')
                ->render($form->get('name')); ?>
        </div>
    </div>
```

I know that is difficult to visualize, but that's what happens usually with HTML views. The code is verbose and usually doesn't fit on one line.

The first thing we do is customize the class of the `div` element that contains the field. We check if there are any error messages for a specific field; in case there are errors, we add the `error` class to the class attribute of the `div` element.

After that, we proceed to print the label of the field. We accomplish that by using the `formLabel()` helper that receives the field as a parameter. This label will contain the `control-label` class, because we specified it while configuring the field in the form file.

Then, we have more HTML to wrap the field itself and right after that, we print the field and the error messages.

The error messages are printed using the `formElementErrors()` method, which receives the field as a parameter; but in this case, we are also customizing the way the helper will render the error messages. We do that by using the provided methods, such as `setMessageOpenFormat()`, `setMessageSeparatorString()`, and `setMessageCloseString()`. As you can see, we customize the helper the first time we use it, and then it will remember the options we set for the next fields.

User.php

In this file, we need to define the filters and validators we are going to use to validate the data we will store. When we work with filters, we need to add the following lines below the namespace to declare the dependencies:

```
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Factory as InputFactory;
```

After that, we can jump to creating the `getInputFilter()` method, which will contain the filters and validators for the data. As we did with the form, we are going to show the relevant sections because the method is vast.

```
$inputFilter->add($factory->createInput(array(
    'name'     => 'username',
    'required' => true,
    'filters'  => array(
        array('name' => 'StripTags'),
        array('name' => 'StringTrim'),
    ),
    'validators' => array(
        array(
            'name' => 'NotEmpty',
            'break_chain_on_failure' => true
        ),
        array(
            'name' => 'StringLength',
            'options' => array(
                'min' => 1,
                'max' => 50
            ),
        ),
    ),
)));
```

This is the first example we are going to examine. Let's focus on the first validation. If you take a closer look, you will notice that it contains an option called `break_chain_on_failure` set to `true`. This means if the validator fails, the validation will not continue and will be stopped there for this specific field. This is useful if you want to avoid the user getting a lot of error messages, or things that don't make sense, or are evident; for example, an error message saying that the value cannot be empty, and then another error message saying that the length should be a minimum of one character.

```
    array(
        'name' => 'EmailAddress',
        'options' => array(
            'messages' => array(
                \Zend\Validator\EmailAddress::INVALID_FORMAT =>
                    'The input is not a valid email address',
            )
        )
    ),
```

This validator belongs to the **Email** field. What I want to highlight here is how we can customize error messages. What we have to do is define them on the `options` key with the name `messages` and using it as a key of the array, use the same constant on the validator to define the original error message. Doing this, we are overwriting the error message for that specific error.

```
    array(
        'name' => 'InArray',
        'options' => array(
            'haystack' => array('0', '1')
        )
    ),
```

Remember we said that when you use a `MultiCheck` element, it adds an `InArray` validator by default. Well, this is how you configure the `InArray` validator manually for a specific element.

The contents of the IndexController.php file

We need to fill the contents of the `indexAction()` method, which we left empty in the previous chapters. Let's review the following lines of code step-by-step:

```
$this->layout('layout/signup');

$viewData = array();
$signupForm = new SignupForm();
$signupForm->setAttribute('action', $this->url()->fromRoute(
    'users-signup'
));
```

The first thing we do is set a new layout. After that, we create the instance of our brand new form and set the `action` attribute to point to the following method:

```
$request = $this->getRequest();
if ($request->isPost()) {
    ...
}

$viewData['signupForm'] = $signupForm;
return $viewData;
```

After that, we check if the request is a POST request; if it isn't, we just pass the form to the view in order to render it if it is, we will proceed with the processing of the form using the following code:

```
$data = $request->getPost()->toArray();

$signupForm->setInputFilter(User::getInputFilter());
$signupForm->setData($data);

if ($signupForm->isValid()) {
    ...
}
```

Here, we are getting the data from the request, configuring the filters and validators on the form based on the configuration we made on the `User` entity, and after that checking if the data sent compiles with what we expect.

```
$files = $request->getFiles()->toArray();
$data = $signupForm->getData();
$data['avatar'] = $files['avatar']['name'] != '' ?
    $files['avatar']['name'] : null;
```

After that, we proceed to process the data itself. By using the following lines, we will get the data filtered by the form and the possible image uploaded by the user, and merge them together:

```
$size = new Size(array('max' => 2048000));
$isImage = new IsImage();
$filename = $data['avatar']['name'];

$adapter = new \Zend\File\Transfer\Adapter\Http();
$adapter->setValidators(array($size, $isImage), $filename);

if (!$adapter->isValid($filename)){
    $errors = array();
    foreach($adapter->getMessages() as $key => $row) {
        $errors[] = $row;
    }
    $signupForm->setMessages(array('avatar' => $errors));
}
```

Notice that you can specify the size of the file using the International System of Units. The accepted SI notation units are: KB, MB, GB, TB, PB, and EB. All these units are converted using 1024 as the base value.

In case the user has uploaded an image, we proceed to validate the size; and in case of an error, we set it on the form using the following code:

```
$destPath = 'data/tmp/';
$adapter->setDestination($destPath);

$fileinfo = $adapter->getFileInfo();
preg_match('/.+\/(.+)/', $fileinfo['avatar']['type'], $matches);
$newFilename = sprintf('%s.%s', sha1(uniqid(time(), true)),
$matches[1]);

$adapter->addFilter('File\Rename',
    array(
        'target' => $destPath . $newFilename,
        'overwrite' => true,
    )
);
```

```
if ($adapter->receive($filename)) {
    $data['avatar'] = base64_encode(
        file_get_contents(
            $destPath . $newFilename
        )
    );

    if (file_exists($destPath . $newFilename)) {
        unlink($destPath . $newFilename);
    }
}
```

The following are the lines that move the sent data from the existing `tmp` folder to our `tmp` folder, then renames the file to avoid collisions and reads the contents of the file. Once the contents have been read, we remove the file and proceed to create the account using the following code:

```
unset($data['repeat_password']);
unset($data['csrf']);
unset($data['register']);

$response = ApiClient::registerUser($data);

if ($response['result'] == true) {
    $this->flashMessenger()->addMessage('Account created!');
    return $this->redirect()->toRoute(
        'wall', array('username' => $data['username'])
    );
}
```

If you remember, we expect to have all the data as one parameter of the `create()` method on the API side. That is why we are getting rid of the `repeat_password`, `csrf`, and `register` values. Right after that, we use the `ApiClient` interface to call the API and then if the response is successful, we redirect the user to the new wall.

The index.phtml file

The `index.phtml` file is really simple, and we are just calling the partial that will render the signup form, which is accomplished using the following lines of code:

```
<?php echo $this->partial(
    'forms/signup-form.phtml', array('form' => $signupForm)
); ?>
```

Changes in the ApiClient.php file

In this file, we just need to add a new method called `registerUser()`, which contains the following lines:

```
$url = self::$endpointHost . self::$endpointUsers;
return self::doRequest($url, $postData, Request::METHOD_POST);
```

Also, we need to adapt the endpoints we have for the users as follows:

```
protected static $endpointUsers = '/api/users';
protected static $endpointGetUser = '/api/users/%s';
```

Summary

We now have a really neat and complete form with which the users can get an account on our social network!

In this chapter, we saw how to secure passwords using the bcrypt encryption provided by the `Zend\Crypt\Password` component. We also saw how to create a complex form with validation for each field, custom error messages, and the proper handling at the frontend using custom classes that will turn the field to red.

In the next chapter, we will complete the cycle of the register/login functionality, allowing the users to log in and log out from the social network with their credentials.

11
Log in

In the previous chapter, we allowed users to create an account on our systems and now we should give them the ability to log in and use our product.

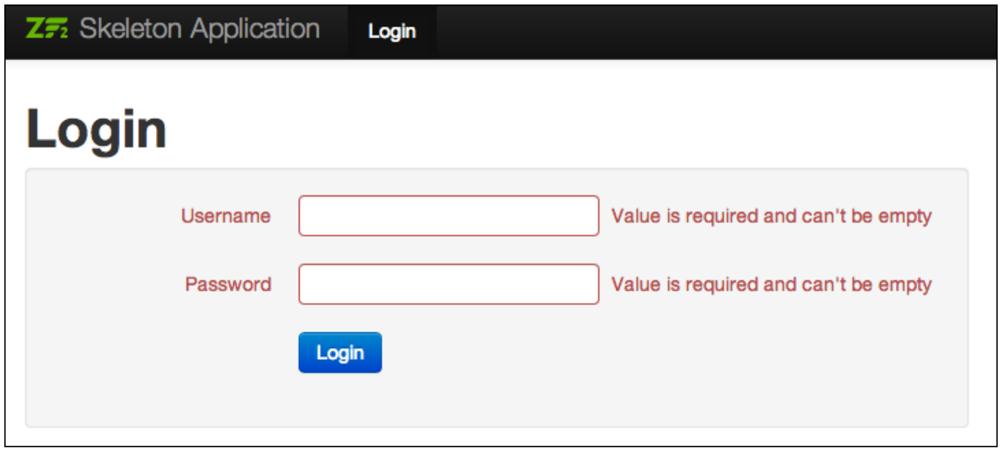
This chapter will be really simple in terms of the API; remember we said that our API is stateless, which means that we need not keep track of sessions. Because of that, the changes we have to do are easy, and we just need to validate the credentials of the user.

This is completely different at the frontend, where we need to put more efforts and code all the session-handling systems, integrate with the existing code and functionalities, and create a form for logging in.

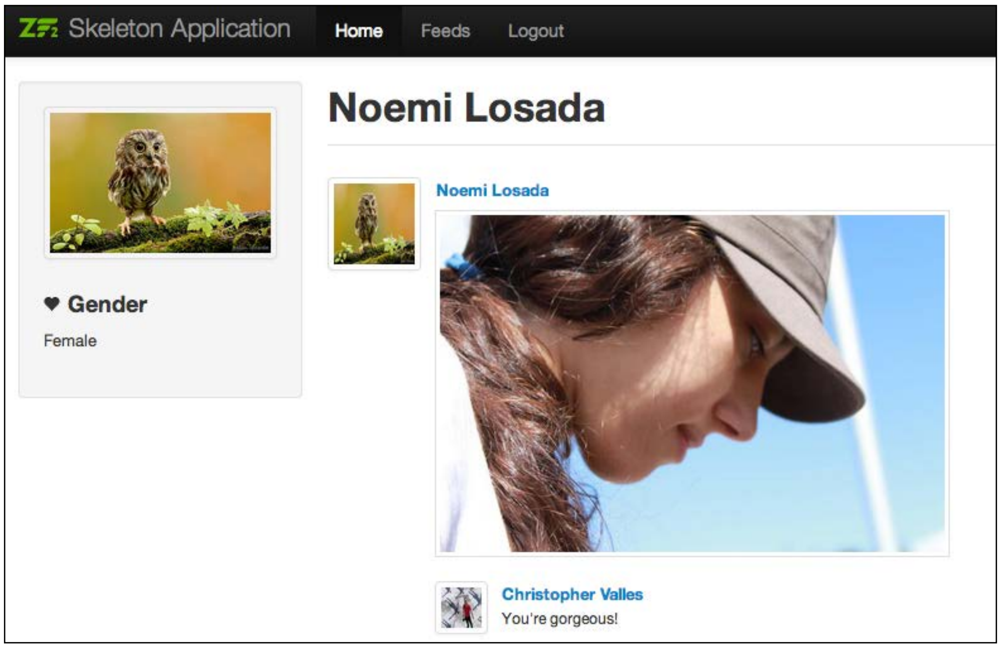
By the end of this chapter, you will know how to deal with authentication and authorization using `AuthenticationService` and the `Acl` components. You will see how to configure the permissions of each group of users and also allow them to log out.

Overview

The following screenshot shows us how the login form looks and how the wall would look when users start commenting on others' posts:


Skeleton Application
Login
Login
Username
Value is required and can't be empty
Password
Value is required and can't be empty
Login

This is how the login form will look. After logging in, the system will recognize us and allow us to comment on the entries made by other users as given in the following screenshot:



API development

As we already discussed, we need to extend the API to be able to validate the credentials of a user and return a proper response to the client.

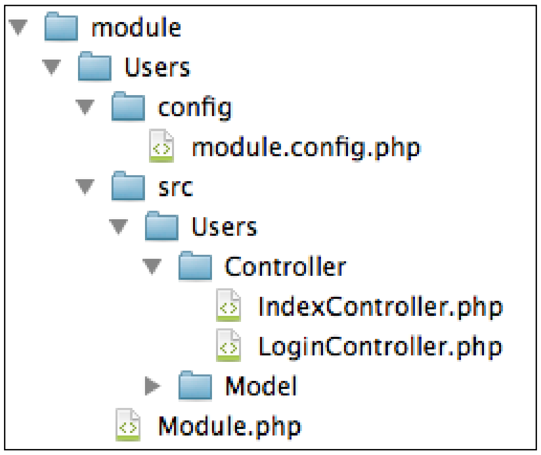
Requirements

The requirements at the API level are as follows:

- Add a new endpoint called `/api/users/login`
- Process the information posted there to check the credentials using bcrypt, which we already discussed in *Chapter 10*, *Sign Up*.

Overview of the module structure

As we did in the previous chapter, we are going to work on the `Users` module, and we will add a new controller. The following screenshot shows us how the folders are organized:



```
module
  Users
    config
      module.config.php
    src
      Users
        Controller
          IndexController.php
          LoginController.php
        Model
    Module.php
```

The module.config.php file

We need to do a couple of changes in this file to receive requests, with the following code:

```
'login' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route' => '/api/users/login',
        'defaults' => array(
            'controller' => 'Users\Controller\Login'
        ),
    ),
),
```

This is the new route we need to declare in order to fulfill the login requests. Because we are adding a new controller, we should also specify the name of the controller in the `controllers/invokable` section as follows:

```
    'controllers' => array(
        'invokables' => array(
            'Users\Controller\Index' =>
                'Users\Controller\IndexController',
            'Users\Controller\Login' =>
                'Users\Controller\LoginController',
        ),
    ),
```

Creating the LoginController.php file

In this controller, we are going accept the credentials as post parameters. Because we are doing everything by extending `AbstractRestfulController`, we should place our code in the `create()` method, even if we are not exactly creating something.

As we need to verify the password using the bcrypt component, we need to add the dependency at the top of the file after defining the following namespace:

```
    use Zend\Crypt\Password\Bcrypt;
```

After we declare the class, we also need to declare a new property, which will hold the user's table and also the method to load the table gateway. At this point of time, you should be able to complete those two tasks by yourself, but if you have to struggle to do it, do not hesitate to check the following source code:

```
    public function create($data)
    {
        $usersTable = $this->getUsersTable();
        $user = $usersTable->getByUsername($data['username']);

        $bcrypt = new Bcrypt();
        if (!empty($user) &&$bcrypt->verify(
        $data['password'],
        $user->password)
        ) {
            $result = new JsonModel(array(
                'result' => true,
                'errors' => null
            ));
        } else {
            $result = new JsonModel(array(
                'result' => false,
                'errors' => 'Invalid Username or password'
            ));
        }

        return $result;
    }
```

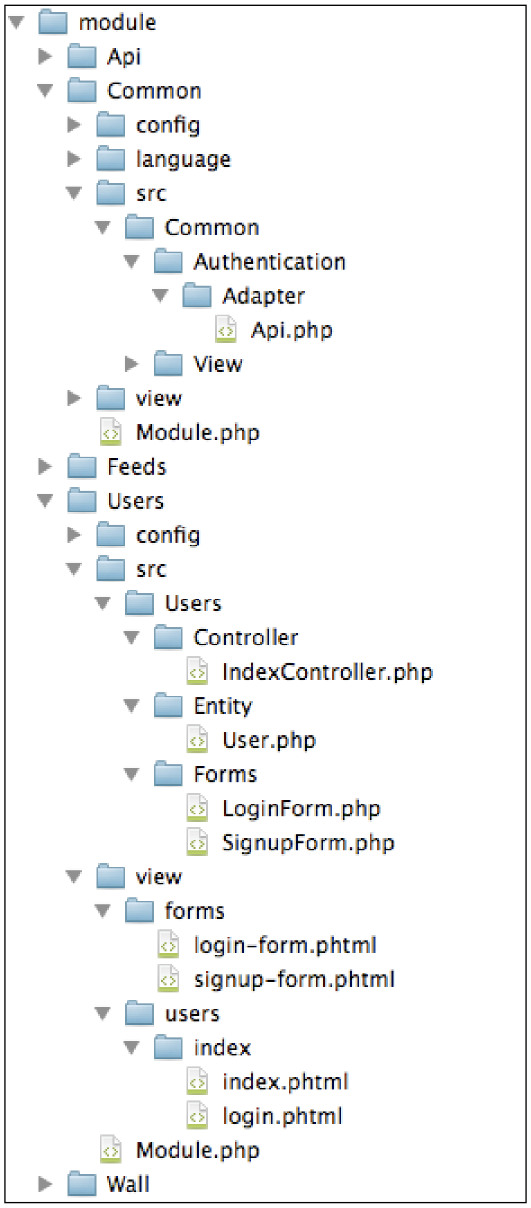
These are the contents of the `create()` method. As you can see, we get the data of the user from the database. If the user exists, we verify the password sent by the client against the password stored in the database. Each time you encrypt a password with bcrypt, the hash will be different, and that's why you cannot just do a simple comparison; you must use the `verify()` method from the `Bcrypt` component, which will take care of checking if the passwords match or not and return a `true`/`false` result. Then, it is just a matter of sending a `true` or `false` value to the client.

As we are again using sensible information, we should do this under the HTTPS protocol in a production-ready project to ensure that the plain text password is encrypted by the HTTPS protocol and kept safe when sending it to the server.

Frontend

Let's review the work we need to do on the client side of the app. We need to create a new layout in order to remove the header menu, create the signup form and its view, modify the controller to use the new layout and process the sign-up form, modify the controller view to show the sign-up form, and finally adapt `ApiClient` to be able to send the data to the API side.

The following is the structure of the module:



```
module
  Api
  Common
    config
    language
    src
      Common
        Authentication
          Adapter
            Api.php
          View
    view
    Module.php
  Feeds
  Users
    config
    src
      Users
        Controller
          IndexController.php
        Entity
          User.php
        Forms
          LoginForm.php
          SignupForm.php
    view
      forms
        login-form.phtml
        signup-form.phtml
      users
        index
          index.phtml
          login.phtml
    Module.php
  Wall
```

The login form

In the login form, we will ask the user for the username and the password. So, the form will be very simple. Also, when we work with forms, we need to also create the form view and the filter configuration. I'm pretty sure that you are totally capable of creating these files by yourself, so give it a try! Remember that you can always check the source code provided in the book.

Changes in the ApiClient.php file

We need to add a new method named `authenticate()` in this object in order to be able to authenticate the user. As you can imagine, the method is really simple and we just need to post the data to the API as follows:

```
protected static $endpointUserLogin = '/api/users/login';
```

We need to add this line at the top of the file to define the new endpoint, and then add the following method:

```
public static function authenticate($postData)
{
    $url = self::$endpointHost . self::$endpointUserLogin;
    return self::doRequest($url, $postData, Request::METHOD_POST);
}
```

Creating a new authentication adapter

Usually, when you use the authentication components from ZF2, you use the `DbTable` adapter that help you in verifying the credentials against a database, specifying a table name, a username column, and a password column. In our case, as we have taken the API-centric approach, we cannot do this directly from the client, and we need to interface it with the API.

As usual, the ZF2 components provide multiple ways to extend them or modify their behavior to fit our needs, and that's what we are going to do now. Create a new adapter for the authentication component that will use the API we created to validate the credentials. Because this is something totally new, let's review it together line by line.

We will start by creating the `Api.php` file at the same location as in the preceding screenshot:

```
namespace Common\Authentication\Adapter;

use Zend\Authentication\Adapter\AdapterInterface;
use Zend\Authentication\Result;
use Api\Client\ApiClient;
use Users\Entity\User;
use Zend\Stdlib\Hydrator\ClassMethods;
```

These are the namespaces and components we are going to use in this class. As you can see, we have also included the `User` entity and the `ClassMethods` hydrator; you will soon understand why.

The following code defines the class that extends from the base interface for the adapters:

```
class Api implements AdapterInterface
```

The following two properties are going to store the credentials a user sends via the form:

```
private $username = null;
private $password = null;
```

These properties will be populated in the constructor as given in the following code:

```
public function __construct($username, $password)
{
    $this->username = $username;
    $this->password = $password;
}
```

We will create an instance of the adapter manually and specify the username and password at that point for later usage:

```
public function authenticate()
{
    $result = ApiClient::authenticate(array(
        'username' => $this->username,
        'password' => $this->password
    ));

    if ($result['result'] === true) {
        $hydrator = new ClassMethods();
        $user = $hydrator->hydrate(
            ApiClient::getUser($this->username), new User()
        );

        $response = new Result(
            Result::SUCCESS,
            $user,
            array('Authentication successful.')
        );
    } else {
        $response = new Result(
            Result::FAILURE,
            NULL,
            array('Invalid credentials.')
        );
    }

    return $response;
}
```

This is the only method specified on the adapter interface that must be implemented in our class. We have to implement the code, which will check the credentials here and return the result as an instance of `Zend\Authentication\Result`. In order to create the result object, we need to pass the result/failure of the authentication to the instance at construction time, using the constants provided for the class, identity data, and message.

The identity message is a payload we store on the session to identify the user; you can store the ID of the user or the whole object. In our case, we will store an instance of the `User` entity so that we can also access the data and related info.

As you can see on the first lines of the method, we use the `ApiClient` method to check with the API if the credentials are fine. Based on the result from the API, we hydrate an instance of the `User` entity to store it as an identity data or we generate an error. Finally, no matter what the result is, we generate the specified `Result` object with the corresponding information in it.

The IndexController.php file

We need to do four changes on this controller. In the `indexAction()` method, we should check if the user is logged in at the beginning. In that case, we do not need to show the signup form to the user, we will redirect it to the user wall. This is accomplished by adding the following code at the top of the method:

```
    $auth = new AuthenticationService();
    $loggedInUser = $auth->getIdentity();

    if ($loggedInUser !== null) {
        return $this->redirect()->toRoute(
            'wall',
            array('username' => $loggedInUser->getUsername())
        );
    }
```

Now, the second change we need to do is at the end of the method. We want to allow the user to log in as soon as his or her account is created. Because we are going to authenticate the user against our API, we need to declare the components we want to use by adding the following code at the top of the file:

```
use Zend\Authentication\AuthenticationService;
use Common\Authentication\Adapter\Api as AuthAdapter;
```

We need to add the following code before redirecting the user to the wall to log in:

```
$auth = new AuthenticationService();
$authAdapter = new AuthAdapter(
    $data['username'],
    $data['password']
);
$auth->authenticate($authAdapter);
```

In the preceding code snippet, we created an instance of `AuthenticationService`, and then created an instance of our adapter aliased here as `AuthAdapter`. We provided the username and the password to the constructor, and finally, we called the `authenticate` method on the `$auth` object, passing the auth adapter.

The third change we have to do in this file is adding the `loginAction()` method in order to show the login form, and also to process the login. As we are going to create an instance of the login form, we need to import the following code at the top of the file:

```
use Users\Forms\LoginForm;
```

Now let's review the code we are going to add on the newly created method:

```
$viewData = array();
$flashMessenger = $this->flashMessenger();

$loginForm = new LoginForm();
$loginForm->setAttribute(
    'action', $this->url()->fromRoute('users-login')
);
```

The following are the first lines. As you can see, we get an instance of the flash messenger and also create an instance of `LoginForm`, to set its action:

```
$request = $this->getRequest();
if ($request->isPost()) {
    ...
}

$viewData['loginForm'] = $loginForm;

return $viewData;
```

This is the structure of the new block of code. As you can see, we get the request and check if we are processing the form or just showing it. To show it, we assign it to the view and at the end, we also check for any messages available on the flash messenger, and pass them to the view.

The following is the code we will execute when we are processing the form. This code will fit on the suspension points, as we have already seen in the skeleton.

```
$data = $request->getPost()->toArray();

$loginForm->setInputFilter(User::getLoginInputFilter());
$loginForm->setData($data);

if ($loginForm->isValid()) {
    $data = $loginForm->getData();

    $auth = new AuthenticationService();
    $authAdapter = new AuthAdapter(
        $data['username'], $data['password']
    );
    $result = $auth->authenticate($authAdapter);

    if (!$result->isValid()) {
        foreach ($result->getMessages() as $msg) {
            $flashMessenger->addErrorMessage($msg);
        }
    } else {
        return $this->redirect()->toRoute(
            'wall', array('username' => $data['username'])
        );
    }
}
```

The code gets the data from the request and configures the filter from the `User` entity to validate and filter the sent data. If the form is valid, we continue getting the data filtered from the login form, and then we set up the authentication object, as we did in the `indexAction()` method. In case the login is successful, we redirect the user to the wall; in case of a failure, we will show the error message sent by the API using the flash messenger.

If we allow the users to log in to our product, we also need to provide a way for them to log out and that's the last change we have to do in this controller, as given in the following code:

```
$auth = new AuthenticationService();
if ($auth->hasIdentity()) {
    $auth->clearIdentity();
}

return $this->redirect()->toRoute('users-login');
```

These are the contents of the `logoutAction()` method. As you can see, we checked if the user has an identity or not. In case he or she has, we just clear the stored identity so that we are not able to identify who the user is. After that, we redirect them to the login page.

The login.phtml file

This is the view that will show the form to the user, it's really simple. As we already saw how to add forms to the view, we will try to define the contents of this file on our own. Remember that you can always check the source code if needed.

Changes in module.config.php

In this chapter, we are adding the ability to log in and log out of our application, and we already completed the methods needed on the controller. The last step is to create the routes so that the methods can be accessed from a browser.

```
'users-login' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route'    => '/login',
        'defaults' => array(
            'controller' => 'Users\Controller\Index',
            'action' => 'login'
        ),
    ),
),
'users-logout' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route'    => '/logout',
        'defaults' => array(
            'controller' => 'Users\Controller\Index',
            'action' => 'logout'
        ),
    ),
),
```

These are the two routes you need to add to point the URLs to an appropriate method in `IndexController`.

The global.php file

Now that we have two routes available to use, we need to add the links that will actually allow the users to use the log in and log out methods. They can always remember the URLs, but that's not always possible. To add new items on the header menu, we need to modify this file and add the configuration of the new links, as given in the following code:

```
    array(
        'label' => 'Logout',
        'route' => 'users-logout',
        'resource' => 'users-logout'
    ),
    array(
        'label' => 'Login',
        'route' => 'users-login',
        'resource' => 'users-login'
    ),
```

If you pay attention to the preceding code, you will notice that there is a new key on the configuration called `resource`. We will talk more about it later, but for now we can say that the data stored there will control if the item should be shown on the menu or not.

The layout.phtml file

As of now, the users can log in and log out, but we need to show or hide the links on the menu properly. The `navigation` helper can help us on this task; however, in order to be able to do it, we need to provide some information. We need to replace the following lines of code:

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

The following code should replace the preceding code:

```
<?php if ($this->loggedInUser !== null) : ?>
<?php $this->navigation('navigation')->findBy('route', 'wall')
        ->setParams(
            array('username' =>
                $this->loggedInUser->getUsername()
            )
        );
    ?>
<?php $this->navigation('navigation')
        ->findBy('route', 'feeds')
        ->setParams(array('username' =>
            $this->loggedInUser->getUsername())
        );
    ?>
<?phpendif; ?>
<?php echo $this->navigation('navigation')
    ->menu()->setUlClass('nav')->setAcl($this->acl)
    ->setRole($this->userRole)->renderMenu(); ?>
```

As usual, the code of the view is not as neat as we want it to be and is kind of hard to read. But what we are essentially doing here is customizing one of the items of the menu to point to the user wall when the user is logged in, and then providing the `navigation` helper with the data needed to show or hide the items. In the following sections, we will talk about what that information is, how to create it, and how to work with it.

ACLs

ZF2 provides two ways of handling the permissions of an application. The first one and the one which we use here is the ACLs, the second one is the Rbac. The difference between them is that ACLs is focused on resources, which means short modules, controllers, and actions. The Rbac model focusses more on roles and their permissions, allowing you to create permissions that are not attached to any specific resource. If you need something simple, you can go with Rbac; if you need something more complex, you can go with ACLs. But you usually do almost the same with both the models and they both more or less have the same "glue" code.

In our product, we will use ACLs; the concept behind the Access Control List is allowing roles' request access to resources. A role is an object that needs access somewhere and a resource is an object whose access is protected and controlled. In our case, we will create a role called guest for the users not logged in, and another one called member for the logged-in users. After that, we will create resources to protect the access to all the functionalities we have in place. This sounds a little bit complicated, but I'm sure that you will understand everything after implementing it.

As we are developing everything under specific modules, we are going to provide each module with the ACL configuration that affects that module directly. Then we will load it when ZF2 reads the module.

The ACL implementation on ZF2 is very flexible and allows you to store the configuration of the roles and resources in any way you want. Actually, it's the developer's responsibility to come up with a storage and loading mechanism. We will now see how we are going to do it on our product, but this is surely not the only way to do it and probably not the best one:

```
<?php

return array(
    'roles' => array(
        'guest',
        'member'
    ),
    'permissions' => array(
        'member' => array(
            'feeds',
            'feeds-subscribe',
            'feeds-unsubscribe'
        )
    )
);
```

The preceding code is the content of the `module.acl.php` file stored in the `config` folder of the `Feeds` module. As you can see, the configuration consists of an array, inside which we specify the roles that interact with this module and the permissions assigned to each role. In order for a role to access a resource, the resource should be added in here. If you look closely at our resources, you may notice that we are using the names of our routes. In this case, the route is tied to a module, controller, and action and the route will conform to our resource.

In this configuration, we are allowing the members to access the feeds list, the subscription action, and the unsubscribe functionality. The guest users are not allowed to see anything and that's why there is no key for them under the `permission` key.

The following is the ACL configuration of the `Users` module and `Wall` module respectively:

```
return array(
    'roles' => array(
        'guest',
        'member'
    ),
    'permissions' => array(
        'guest' => array(
            'users-signup',
            'users-login'
        ),
        'member' => array(
            'users-logout'
        )
    )
);

return array(
    'roles' => array(
        'guest',
        'member'
    ),
    'permissions' => array(
        'member' => array(
            'wall'
        )
    )
);
```

As you can see in the first block, the guest user is allowed to access the login and the signup form, but not the logout functionality. Also, the member is not allowed to go to the login form because he or she is already logged in, and also it is not allowed to go to the signup.

In terms of the `Wall` module, the only user role allowed is the `member`.

Modules

Now, let's see how to inform ZF2 about our ACL configuration and how to create them in terms of objects.

On every `Module.php` file, we need to load the ACL configuration using bootstrap, and then add an event listener to the `route` event to check if the user has the rights to access the resource.

The objects that conforms the ACL on ZF2 are the `Acl` class, `GenericRole` class, and `GenericResource` class. In order to import them, you should add the following lines of code at the top of each `Module.php` file that has something to do with ACLs:

```
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;
use Zend\Permissions\Acl\Resource\GenericResource as Resource;
```

We need to modify the `onBootstrap()` method of every module that has ACLs, to load the configuration in every `Module.php` file. This is accomplished by adding the following line at the top of the method:

```
$this->initAcl($e);
```

After that change, we need to implement the `initAcl()` method as follows:

```
public function initAcl(MvcEvent $e)
{
    if ($e->getViewModel()->acl == null) {
        $acl = new Acl;
    } else {
        $acl = $e->getViewModel()->acl;
    }

    $aclConfig = include __DIR__ . '/config/module.acl.php';
    $allResources = array();

foreach ($aclConfig['roles'] as $role) {
```

```
            if (!$acl->hasRole($role)) {
                $role = new Role($role);
                $acl->addRole($role);
            } else {
                $role = $acl->getRole($role);
            }

            if (array_key_exists(
                $role->getRoleId(), $aclConfig['permissions'])
            ) {
                foreach (
                    $aclConfig['permissions'][$role->getRoleId()] as
                        $resource) {
                    if (!$acl->hasResource($resource)) {
                        $acl->addResource(new Resource($resource));
                    }
                    $acl->allow($role, $resource);
                }
            }
        }

        $e->getViewModel()->acl = $acl;
    }
```

Let's review what are we doing here. In order to share the object that represents the ACL, we need to store it somewhere and be able to access it from the views and the `Module.php` files. This is why the first line checks if there is something stored in the view model. In case nothing is found, we create an instance of the `Acl` object.

Right after that, we load the `module.acl.php` config file to a variable and start iterating over the roles. We then check if each role is already created on the ACL storage. If not, we create it using a new instance of `Role`. In case it's defined, we just get its instance to use later.

After that, we check if the module has permissions defined; in case we have them, we iterate over them by adding the resources using the instances of `Resource` and then allow the role stored on the `$role` variable to access the resource we just created.

This will configure the `Acl` object with all the roles and permissions of the `Wall`, `Users`, and `Feeds` modules. Also, remember that `Role` and `Resource` are just aliases of the classes we are really using.

Finally, after the configuration of the ACLs, we need to check if the user has access to a specific route while loading it. In order to do it, we need to add or register a method with the `route` event and then insert the code on the `Common` module.

At the end of the `onBootstrap()` method, we have to add the following lines of code to register the listener for the event:

```
$e->getApplication()->getEventManager()->attach(
'route', array($this, 'checkAcl')
);
```

This will execute the `checkAcl()` method every time the `route` event is fired:

```
public function checkAcl(MvcEvent $e) {
    $route = $e->getRouteMatch()->getMatchedRouteName();
    $routerParams = $e->getRouteMatch()->getParams();
    $auth = new AuthenticationService();

    $userRole = 'guest';
    if ($auth->hasIdentity()) {
        $userRole = 'member';
        $loggedInUser = $auth->getIdentity();
        $e->getViewModel()->loggedInUser = $loggedInUser;
    }

    $e->getViewModel()->userRole = $userRole;

    if (substr($route, 0, 5) == 'feeds' &&
        $loggedInUser->getUsername() != $routerParams['username'])
    {
        $response = $e->getResponse();
        $response->setStatusCode(404);
        return;
    }

    if (!$e->getViewModel()->acl->isAllowed($userRole, $route)) {
        $response = $e->getResponse();
        $response->setStatusCode(404);
        return;
    }
}
```

This is the `checkAcl()` method. What we do in the first few lines is get the information about the route, we process the data about the identity of the user and the role. Because we only have two roles, by default the role is `guest`, unless the user is logged in. In that case, we set the role manually to `member`.

Products with more roles is a good idea to store the role of the user on the database and load it on the `User` entity, as we are storing the `User` entity as the identity of the user on the session. You will be able to easily check the role associated with a user and avoid assigning it manually.

After gathering the data, we have two big checks. The first one is a specific case for the feeds functionality. As users are members by default, they are allowed to see the feeds, modify the URLs, change the username for the other feed, and access the data of another user. In order to avoid this, we have several options. The first one we implement here is determine if we are working with feeds routes, and then check if the username the user is trying to access matches with his username. If they do not match, we redirect the user to a 404 page. The second option is not implemented here, but you definitely need to implement it in a ready-for-production application, checking the owner of the data you are accessing against the username trying to access it on all the methods, and of course, never trusting what is coming from the client.

The second big check we do here is to check if the role of the user is actually allowed to access the route we are processing. In case it is not allowed, we will redirect it to a 404 page.

The index.phtml file

Finally, to finish with the changes of this chapter, we need to modify the `index.phtml` page of the `Wall` module. As we are going to visit other people's walls, we don't want to allow the user to publish content on other's walls (they can only post comments).

What we have to do is check in the view if we are on our own wall or on another user's wall. We do this by using the following line of code:

```
<?php if ($isMyWall) : ?>
```

Of course, the `$isMyWall` variable should be populated from the controller. We do that by adding the following lines of code:

```
$viewData['isMyWall'] = !empty($loggedInUser)?
$loggedInUser->getUsername() == $username : false;
```

Wall IndexController.php

As we already mentioned, you need to do changes in the `IndexController.php` file to determine if the wall the user is accessing is his or her own or another person's wall. Another thing you need to change in this file is the owner of the posted comments. Until now, all the comments were assigned to the current user; however, because the users now have the ability to log in, the comment posted should be credited to the right author. Again, this change is pretty straightforward and we will experiment with it and try to get it working.

Summary

Phew! We learned a lot of concepts in this chapter.

We saw how to use the authentication components on ZF2 to allow users to log in and log out on our application, and also to create custom adaptors to connect to different backends. After that, we saw how ACLs work, and how to store them and recreate them according to our needs. We also saw how to check if a specific user has appropriate rights to access a specific functionality.

In the next chapter, we will see how to send e-mails to users and implement the forgot password functionality to allow them to retrieve a password using the e-mail we send to them.

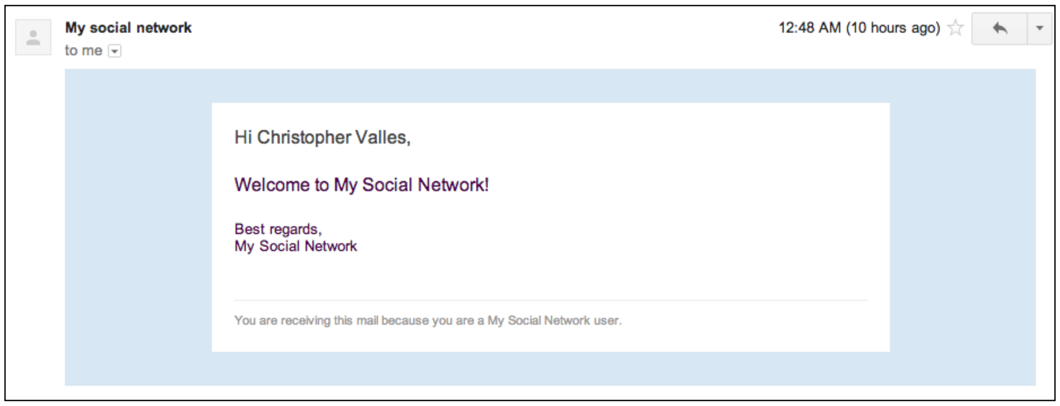
12
Sending E-mails

Our social network has a lot of functionality and the first version is almost complete. In order to notify the users about what happens while they are away we will send them e-mails. Specifically, they will receive a welcome e-mail when they create an account and after that they will receive an e-mail each time someone posts a comment on their wall.

By the end of this chapter, you will know how to send e-mails using the `Zend\Mail` components. We will also see how to organize the HTML content of the e-mails, and how to compose them using templates and layouts.

Overview

A welcome e-mail will look similar to the following screenshot; the text is lame but as an example will work.



My social network
to me
12:48 AM (10 hours ago)

Hi Christopher Valles,

Welcome to My Social Network!

Best regards,
My Social Network

You are receiving this mail because you are a My Social Network user.

API development

All the work on this chapter will be done on the API side of the project and no changes are required on the client. In this way, we can manage exactly what we are sending, the recipient, and so on.

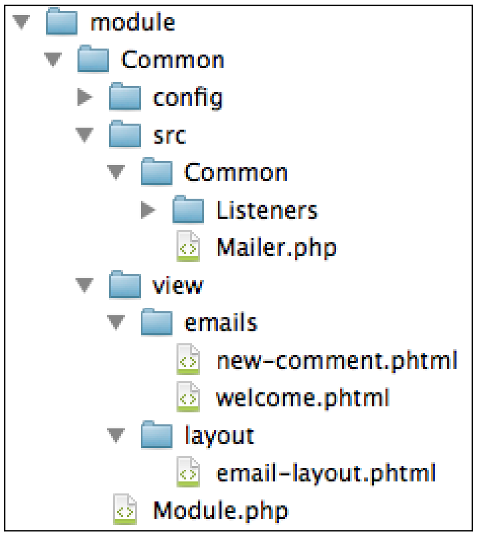
Requirements

We have a few requirements for this section of the API, which are as follows:

- We need to centralize all the e-mail logic on one class
- The e-mails must support HTML
- The HTML must not be mixed with the PHP code and they have to be placed on a template
- The e-mails must use a common layout to simplify the changes on the skeleton of the e-mail.

The module structure

As the code will be used in different types of controllers, we are going to add the code to the `Common` module. The following screenshot shows how the folder structure will look, along with the new files:



```
module
  Common
    config
    src
      Common
        Listeners
        Mailer.php
    view
      emails
        new-comment.phtml
        welcome.phtml
      layout
        email-layout.phtml
    Module.php
```

Adding the mailer.php file

This is the class where we are going to centralize all the code related to the e-mail sending functionality and this is also the class that we have to change every time we want to add a new e-mail to the system. The structure is very simple, we will have a bunch of specialized methods that will take care of the information of a single e-mail type. Then we will have a method to initialize some common code, and finally a method that will take care of sending the actual e-mail. The following are the contents of the `mailer.php` file:

```
namespace Common;

use Zend\Mail\Message;
use Zend\Mail\Transport\Sendmail;
use Zend\View\Renderer\PhpRenderer;
use Zend\View\Resolver\TemplateMapResolver;
use Zend\View\Model\ViewModel;
use Zend\Mime\Message as MimeMessage;
use Zend\Mime\Part as MimePart;
```

These are the initial lines of the class. As usual, we have declared the namespace and imported the components we need to use. The first three `use` statements refer to the `Mail` components.

There are different ways of sending e-mails you can deliver them locally if the e-mail accounts are stored on the same server on which your app is running, you can relay them to another server to send it, or you can send it from your server using some sort of mail server. In our case, the virtual machine provided with the book has `postfix` installed that allows us to send e-mails using a program called `sendmail`. ZF2 provides different transport objects that do the actual job of interfacing the PHP code with the sending program to send the e-mail; that's why, in this case, we import the `Zend\Mail\Transport\Sendmail` component.

The next three lines are used to import the `View` components we are going to use to process the HTML code of the e-mails.

The final two lines are used together with the `Message` component to be able to specify the HTML content of the e-mail, otherwise the contents of the e-mail will be treated as plain text, and the e-mail programs will not read the HTML code. We can define a class using the following code:

```
class Mailer
```

As you can see, the class declaration is really simple; in this case we are not extending from another class or implementing an interface.

The following is the first specialized method we are going to see:

```
public static function sendContentNotificationEmail(
    $recipientAddress, $recipientName, $authorName
){
    $subject = 'New comment on your wall!';
    $templateVars = array(
        'recipientName' => $recipientName,
        'authorName' => $authorName
    );

    self::send(
        $recipientAddress,
        $recipientName,
        $subject,
        'NewComment',
        $templateVars
    );
}
```

In this case, the `sendContentNotificationEmail()` method takes care only of the content notification e-mail. This e-mail is sent every time a person posts a comment on the user's wall.

As you can see, we expect the name and e-mail address of the recipient and the name of the author of the comment. After that, we store the subject of the e-mail on a variable and we build an array with the variables we want to pass to the template of the e-mail.

Finally, we call another method called `send()`, which will take care of sending the e-mail. The parameters we need to pass to the `send()` method are the recipient's address, the recipient's name, the subject of the e-mail, the name of the template we want to use, and the variables we want to pass to the template. Later, we will see how the name of the template is mapped to the actual filename of the template. The following is the `sendWelcomeEmail()` method:

```
public static function sendWelcomeEmail(
    $recipientAddress, $recipientName
){
    $subject = 'Welcome to My Social Network';
    $templateVars = array(
        'recipientName' => $recipientName
    );

    self::send(
        $recipientAddress,
        $recipientName,
        $subject,
        'WelcomeTemplate',
        $templateVars
    );
}
```

The `sendWelcomeEmail` method will take care of sending the welcome e-mail to the user when they create an account. As you can see, this is very similar to the one we saw before, and expects the e-mail address and the name of the recipient. The `initResolver()` method is as follows:

```
    protected static function initResolver()
    {
        $resolver = new TemplateMapResolver;
        $resolver->setMap(array(
            'MailLayout'
                 => __DIR__ . '/../../view/layout/email-layout.phtml',
            'WelcomeTemplate'
                 => __DIR__ . '/../../view/emails/welcome.phtml',
            'NewComment'
                 => __DIR__ . '/../../view/emails/new-comment.phtml',
        ));

        return $resolver;
    }
```

The preceding code takes care of initializing a component we will use on the `send()` method. The code itself initializes a `TemplateMapResolver` object that maps keys to the path of a file. It is in this code that we are mapping the names of our templates with the actual path and filename of the template, and this allows us to avoid inserting paths everywhere to specify the template we want to use. As you can see in the preceding code, we also specify the layout we are going to use on the e-mails.

If, in the future, you need to add some more e-mails to the social network, you will need to map here the name of the new template with the path of the file containing the HTML message.

Now, let's see the contents of the last method of this class. It is in the `send()` method, where we are going to glue all the pieces together in order to send the e-mail to the recipient's inbox. The `send()` method is as follows:

```
protected static function send(
    $toAddress,
    $toName,
    $subject,
    $templateName,
    $templateVars = array()
){
    $view = new PhpRenderer;
    $view->setResolver(self::initResolver());

    $viewModel = new ViewModel;
    $viewModel
        ->setTemplate($templateName)
        ->setVariables($templateVars);
    $content = $view->render($viewModel);

    $viewLayout = new ViewModel;
    $viewLayout->setTemplate('MailLayout')->setVariables(array(
        'content' => $content,
    ));

    $html = new MimePart($view->render($viewLayout));
    $html->type = "text/html";

    $body = new MimeMessage();
    $body->setParts(array($html));

    $mail = new Message;
    $mail->setBody($body);
    $mail->setFrom('no-reply@example.com', 'My social network');
    $mail->addTo($toAddress, $toName);
    $mail->setSubject($subject);

    $transport = new Sendmail;
    $transport->send($mail);
}
```

As you can see, the preceding method expects the parameters we listed before. The first two lines of code instantiates a `PhpRenderer` component. This is used because our HTML templates will include the PHP code in order to insert some useful information, such as the name of the recipient or the name of the author who posted the content on the wall.

After that, we create a `ViewModel` instance and we set the template of the model to the template name passed as a parameter, and we also pass the template variables to replace them in the HTML code. Right after that, we will render the code using the `PhpRenderer` object and store it in a file. At this point, `PhpRenderer` is using the `TemplateMapResolver` object we configured before in order to translate the template name passed as a parameter to the actual path of the file.

The next section of code is going to render the layout of the e-mail passing the content of the e-mail as the `content` template variable.

Now we have a `ViewModel` object that contains the layout and the injected HTML code from the e-mail message creating a longer HTML code.

We now jump to create the e-mail itself. Nowadays, e-mail messages are of the `multipart` type, which means that we can use a part of the `text/plain` type to specify the contents of the e-mail as plain text and another part of the `text/html` type to specify the HTML version of the message.

As we are dealing with HTML, we need to specify the `text/html` part on the e-mail and that's what we are doing in the next section. The content of the `MimePart` object is the HTML we had stored on the `$viewLayout` variable we created before. Remember that this variable contains not only the layout, but also the contents of the e-mail.

As this is a message that will contain the HTML code, we need to create a `Message` class that supports the feature we are trying to use and that's why we create an `MimeMessage` object.

After setting the HTML part on the `MimeMessage` object, we will create the final e-mail object instantiating the `Message` class. We set the body, the sender's name and address, the recipient's name and address, and the subject of the e-mail.

Finally, we will create an instance of the `Sendmail` transport and use it to send the e-mail to the Internet.

The email-layout.phtml file

As usual, we are not going to paste the HTML code here because it is long, verbose, and will not be readable at all; instead, we are just going to highlight one thing. As you saw before, we will inject the content of the e-mail to the layout using a variable called `$content`.

As this file will be parsed by the `PhpRender` object, we can use the PHP code and that's how we are going to merge the HTML code of this file with the HTML code injected in the `send()` method. The specific line has no mystery, it is similar to the following one:

```
<?php echo $content ?>
```

This is how we are usually going to insert the content on the e-mail. In this case, it is the whole message, but we are going to see in the next section how we can insert the name of the user.

The view welcome.php file

The `view welcome.php` file contains the code we use when we send a message to the user who just created an account. The text is pretty lame and we just say hello. However, as an example, it is enough.

The e-mail itself is customized with the name of the user and this is as simple as echoing the variable sent by the method that handles the following e-mail message, if you want to try to add more contents on the e-mail and insert more data of the user and make it more friendly:

```
Hi <?php echo $recipientName ?>
```

The view new-comment.phtml file

When a user posts a comment, the owner of the wall receives an e-mail notifying about what just happened. This e-mail message is still lame but it contains the author name of the comment. The message will contain a line similar to the following one:

```
Hi <?php echo $recipientName ?>,
<?php echo $authorName ?> has posted a new comment on your wall!
```

Modifying the IndexController.php file

Now that we have configured the mail system in place, we need to start calling the system on specific situations to send a proper e-mail.

We are going to add just one line in the `Users` module of the `IndexController` file in order to send an e-mail when the account of the user is created. The code must be placed right after the block of code where we know for sure that the account has been created without problems.

```
Mailer::sendWelcomeEmail($user['email'], $user['name']);
```

The line should look similar to the preceding line. As we created static methods on the `Mailer` class, we don't need to create an instance. Remember, that in order to use this class, you must import it at the beginning of the controller.

The IndexController.php file on the Wall module

The code we need to add to this controller to notify the user about the new content on his wall is a little bit trickier due to the structure of the tables and the way in which the data is stored.

When a request is made to this API endpoint, we know the user ID of the current author but not the user ID of the previous author who made the original post on which is being commented. As our e-mail has to go to the owner of the original content, we need to find out who that user is.

To make everything a little bit uglier, we can comment on three different types of contents, which means that we need to create an instance of a table gateway based on the type of the contents that is being commented.

In the method we have a `switch` structure that we use to set the `$validatorTable` value, and this is where we are going to extract the original entry based on the type of content. The line we have to add on each case of the `switch` statement looks similar to the following one:

```
$table = $this->getUserStatusesTable();
```

Of course, we will change the method call on each case to create an instance of the corresponding table gateway.

After that we can safely get the entry and the author of the original entry using the following code:

```
$entry = $table->getById($data['entry_id']);
$recipient = $usersTable->getById($entry['user_id']);
```

As you can see, we use the table gateway to get a row based on the ID. You will notice that two of the three table gateways do not have the `getById()` method but they are pretty simple to implement. At this point of the app, because some methods on the table gateways are duplicated on multiple files, is a good moment to create a base table gateway file and move the common code to the parent class and extend the children from the new base table gateway. You should be confident enough to jump and do it in case if you didn't refactor the table gateways before.

Finally, after we know that the comment was created successfully, we can safely send the message to the user using the following lines of code:

```
if($creationResult && $recipient['id'] != $user['id']) {
    Mailer::sendContentNotificationEmail(
        $recipient['email'], $recipient['name'], $user['name']
    );
}
```

Can you explain why we are limiting the sending of the e-mail, where the recipient ID and the user ID are different?

The challenge

We are near to the end of the book, and we saw a lot of things, we learned about most of the ZF2 components and we created a basic social network from scratch. At this point, I want to test your skills and measure how much you have learned so far by challenging you. Of course, this will also help you to revise any part of the book to reinforce your knowledge and master ZF2.

I want you to implement a whole new functionality on the social network that will touch almost all the topics we saw so far. The users can now register and log in to the social network, they can post content and read their favorite RSS feeds, but what happens if they forget the password? How can they recover it?

In order to help them, the challenge is to build the forgotten password system we will usually find on almost all the applications out there. The following are the requirements for this:

- A new link should be placed on the login screen to allow the user to go to the recover password page.
- The recover password page should show to the user a new form where they can enter their e-mail address.
- The system should send an e-mail to the user with a unique token assigned to their account and a link pointing them to the recover page; the link should contain their token.
- After clicking on the link in the e-mail, the system must show the user a new form to allow them to change their password by specifying the new password twice.

- After the submission of the form, the clients have to contact the API to notify about the change of the password. Of course, the API will require a new endpoint for this and the new password should be encrypted using bcrypt.
- Once the password has been changed, the unique token for this user should be removed and the user should be redirected to the login page.

Summary

In this chapter we learned how to send HTML e-mails from ZF2 and how to organize the template to have a cleaner code and to avoid having HTML code mixed with PHP. We also created a centralized point for sending the e-mails by simplifying the integration by just adding a line of code at each point where an e-mail has to be sent. Finally, we also modified the API in order to send the two e-mails we created.

The next chapter is the last one, in which we will complete our product functionality by implementing a security layer on the API level. We will use a simpler version of OAuth 2.0 and we will modify the client and the server to communicate using this protocol.

13
OAuth 2.0 Protocol Securing our API

The MVP of our product is almost finished. Due to the nature of the architecture chosen, we have an exposed API. As we don't want people to mess around with our data and users, we need to protect it somehow, and to accomplish that we can implement different mechanisms to authenticate clients, but in this chapter we will implement a simple version of OAuth 2.0. Later on, with a few modifications, we can extend the code that we are going to put in place to allow third-party apps to use our API.

By the end of this chapter we will know how to implement the two-legged OAuth 2.0 variant in a Zend Framework 2 application and secure our APIs.

Overview

Most of the work is going to be done at the API level. The client will just implement a few calls to the API to get the corresponding access token, and then we will be able to query our API.

In ZF2 you can find an OAuth 2.0 component, but unfortunately the component only acts as a client for APIs that support OAuth 2.0. There is nothing done yet on the other side of the problem. If you want to protect your API, nothing on ZF2 is going to help you now.

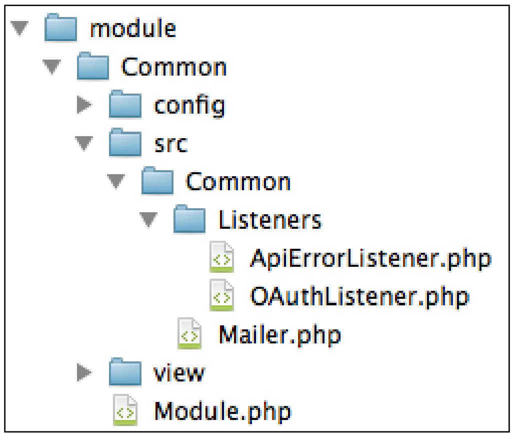
To accomplish the objective of this chapter, we are going to integrate a fabulous OAuth 2.0 server component developed by *Brent Shaffer* under the MIT License. This component will allow us to forget about all the implementations of the OAuth 2.0 protocol and make it work without too much effort.

API development

On this side of the project, we need to integrate the OAuth 2.0 component in order to generate access tokens. OAuth 2.0 gives the possibility of using different grant types. The simplest one is the Client Credentials Grant. In this scenario, the client can request an access token by using only its client credentials. Using this type of grant, we can skip the authorization step and after verifying the credentials of the user we can generate an access token.

The module structure

For the functionality we are implementing, we will add a new module named `OAuth`, and we will also create a new listener on the `Common` module. The folder structure will look similar to the following screenshot:



```
▼ module
  ▼ Common
    ▶ config
    ▼ src
      ▼ Common
        ▼ Listeners
            ApiErrorListener.php
            OAuthListener.php
          Mailer.php
    ▶ view
      Module.php
```

Installing the OAuth 2.0 component

The first thing we need to do is install the OAuth2.0 component we mentioned before on our project. The task is very easy and will be handled by a composer. Open the `composer.json` file and place the following code in the `require` section:

```
"bshaffer/oauth2-server-php": "v1.0"
```

After that, you need to inform the composer to update everything. This can be accomplished by running the following command on a terminal window inside the root folder of the project:

```
phpcomposer.phar update
```

This will update all the dependencies and libraries to the previous version and will also install the new component under the `vendor` folder.

This component will be using the database, so we need to create a few tables. The following are the queries, which we need to run in order to create new tables. These queries are extracted directly from the documentation of the component we are using.

```
CREATE TABLE oauth_clients (client_id TEXT, client_secret TEXT,
redirect_uri TEXT);

CREATE TABLE oauth_access_tokens (access_token TEXT, client_id TEXT,
user_id TEXT, expires TIMESTAMP, scope TEXT);

CREATE TABLE oauth_authorization_codes (authorization_code TEXT,
client_id TEXT, user_id TEXT, redirect_uri TEXT, expires TIMESTAMP,
scope TEXT);

CREATE TABLE oauth_refresh_tokens (refresh_token TEXT, client_id TEXT,
user_id TEXT, expires TIMESTAMP, scope TEXT);
```

Modifying LoginController.php

In this controller, we need to check if the user has sent the correct username and password in order to create an access token. By using the access token, the user will be able to issue calls to the API. Let's review the changes we have to do in the `create()` method.

```
use OAuth2\Storage\Pdo;
use OAuth2\Server;
use OAuth2\GrantType\ClientCredentials;
use OAuth2\Request;
use OAuth2\Response;
```

These are the lines of code we need to add at the beginning. As we are using an external library to handle all the OAuth 2.0 operations, we need to define the namespaces we want to use.

After checking the credentials of the user and being sure that they match, we need to create an access token. We will do that by adding the following code right after the verification of the username and password:

```
$storage = new Pdo(
    $usersTable
        ->adapter
```

```
                ->getDriver()
                ->getConnection()
                ->getConnectionParameters()
        );
        $server = new Server($storage);
        $server->addGrantType(new ClientCredentials($storage));
        $response = $server->handleTokenRequest(
        Request::createFromGlobals()
        );
        if (!$response->isSuccessful()) {
            $result = new JsonModel(array(
                'result' => false,
                'errors' => 'Invalid oauth'
            ));
        }

        return new JsonModel($response->getParameters());
```

Instead of just returning `true` to the client we need to generate an access token.

First, we will create the `PDO` storage object using the connection parameters of a table gateway.

Then, we will create the server and specify that we are working with client credentials in order to skip the authorization step.

After that we will call the `handleTokenRequest()` method on the server by passing a request. This method will take care of generating the access code.

Finally, we will check if everything went fine on the OAuth2.0 side, and we will send a `JsonModel` object to the client containing the access code, the expiration time, and the token type.

Adding OAuthListener.php to the Common module

Now that we have the authorization mechanism in place, we need to protect the API and check if the access token provided in the request is valid. Of course, there is one endpoint that doesn't need to be protected by the OAuth2.0 protocol.

Now, we are going to create a new listener that will inspect all the requests and check the access token. This will be attached to the dispatch event, which we will see later.

The structure of the file is similar to the `ApiErrorListener.php` file that we already have, so feel free to copy and paste it and do the following changes:

```
use OAuth2\Storage\Pdo;
use OAuth2\Server;
use OAuth2\Request;
use OAuth2\Response;
```

The preceding are lines we need to add to the top of the `OAuthListener.php` file, because we are going to use these components.

The following are the contents of the `attach()` method. As you can see, the listener is attached to the dispatch event and will call a method named `onDispatch()`:

```
public function attach(EventManagerInterface $events)
{
    $this->listeners[] = $events->attach(
        MvcEvent::EVENT_DISPATCH, __CLASS__ . '::onDispatch', 1000
    );
}
```

Let's take a look at the contents of the `onDispatch()` method:

```
if ($e->getRequest() instanceOf \Zend\Console\Request) {
    return;
}

if ($e->getRouteMatch()->getMatchedRouteName() == 'login' ||
$e->getRouteMatch()->getMatchedRouteName() == 'users') {
    return;
}

$sm = $e->getApplication()->getServiceManager();
$usersTable = $sm->get('Users\Model\UsersTable');

$storage = new Pdo(
    $usersTable
        ->adapter
        ->getDriver()
        ->getConnection()
        ->getConnectionParameters()
);
$server = new Server($storage);
if (!$server->verifyResourceRequest(Request::createFromGlobals)){
    $model = new JsonModel(array(
        'errorCode' => $server->getResponse()->getStatusCode(),
        'errorMsg' => $server->getResponse()->getStatusText()
    ));

    $response = $e->getResponse();
    $response->setContent($model->serialize());
    $response->getHeaders()->addHeaderLine(
        'Content-Type', 'application/json'
    );
    $response->setStatusCode(
        $server->getResponse()->getStatusCode()
    );

    return $response;
}
```

If you remember, in one of the previous chapters, we created the RSS reader and a CLI interface to process the feeds. This is the reason why the first thing we do here is check that we are not processing a request made through the CLI.

After that, we check the URL that doesn't have to be protected by OAuth 2.0. The route that we don't need to protect is called `users`. If we protect that endpoint, the user will not be able to log in to the application.

After that, we again create the `PDO` object and the server component based on the connection parameters.

Then we proceed to verify if the access token is valid for this resource, and this is accomplished by calling the `verifyResourceRequest()` method on the server component passing the request and response as parameters.

In case of an error, we will return a JSON response to the client specifying the issue and also specifying the 401 HTTP code. Otherwise, we will just allow the request to continue.

Other changes in the Common module

Now, because the request will continue with the `ApiErrorListener` object, we need to avoid processing those requests marked as 401 by the OAuth 2.0 listener. In order to change that behavior, we need to modify the first check on the `onRender()` method in the `ApiErrorListener.php` file. The new check will look as follows:

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

```
        ->getStatusCode() == Response::STATUS_CODE_401
) {
    return;
}
```

After making this change, we need to inform the `Common` module about the new listener we just created. We need to add a listener to the module configuration as follows:

```
return array(
    'service_manager' =>array(
        'invokables' => array(
            'Common\Listeners\ApiErrorListener' =>
                'Common\Listeners\ApiErrorListener',
            'Common\Listeners\OAuthListener' =>
                'Common\Listeners\OAuthListener'
        )
    )
);
```

As you can see, we just added a new entry specifying the path for the new listener.

Finally, we need to attach the listener to the bootstrap event on the module. This will be done by adding the following line next to the `ApiErrorListener` initialization on the `onBootstrap()` method in the `Module.php` file:

```
$events->attach($sm->get('Common\Listeners\OAuthListener'));
```

This will attach the listener to the dispatch event every time the module is bootstrapped.

Frontend

Now that we have all the work done at the API level we need to adjust the frontend to store the access token once we log in to the app and pass the access token along with the request while accessing the API. We will also set up the expiration on the containers of the session because the token has expiration.

Modifying ApiClient.php

As this is the centralized point for all the API requests, this is the only place where we need to send the access token to the API. We need to first import the `Container` component by using the following code to access the sessions we are using to store the access token:

```
useZend\Session\Container;
```

After that, we will create a new property using the following code to store the session object to avoid creating it over and over again:

```
protected static $session = null;
```

Then we need to create the following method to retrieve the `Container` object:

```
public static function getSession()
{
    if (self::$session === null) {
        self::$session = new Container('oauth_session');
    }

    return self::$session;
}
```

Now that we have this code in place to access the session, we need to modify the `authenticate()` method to pass the OAuth 2.0 data along with the request.

```
public static function authenticate($postData)
{
    $postData['grant_type'] = 'client_credentials';
    $postData['redirect_uri'] = 'http://example.com';
    $postData['client_id'] = 'zf2-client';
    $postData['client_secret'] = 'mysupersecretpass';

    $url = self::$endpointHost . self::$endpointUserLogin;
    return self::doRequest($url, $postData, Request::METHOD_POST);
}
```

As you can see, we have hardcoded the data in the method. We can move it to the configuration file for convenience, but for this example you will see it clearly here.

First, we need to specify the grant type. As we are working with client credentials, we should pass `client_credentials` on the `grant_type` variable. Then, we also need to send a client ID and a client secret. These two values will be similar to the consumer key and the consumer secret found in OAuth 1.0 and basically identifies each app that has access to OAuth 2.0 with a unique ID and a secret shared between the server and the app.

If you are going to develop a third-party access, you will need to generate these two values for each app you want to integrate and provide them to the developer of the app.

The last change we need to do is located in the `doRequest()` method and will take care of sending the access token to the API on each request. As the backend already ignores the access token for those URLs that don't have to be protected, we can always securely send it.

Before setting the parameters to the client, we need to add the following lines of code:

```
if ($postData === null) {
    $postData = array();
}

$postData['access_token'] = self::getSession()->accessToken;
```

As you can see, the preceding code adds the access token to the array used to set the parameters of the request.

Modifying Api.php

This is the last file we need to modify on the client in order to use OAuth 2.0. While authenticating a user, we must store the access token returned by the server. Let's see the code we need to add at the top of the `authenticate()` method:

```
if (array_key_exists('access_token', $result) &&
!empty($result['access_token'])) {
    $hydrator = new ClassMethods();
    $user = $hydrator->hydrate(
    ApiClient::getUser($this->username), new User()
    );

    $session = new Container('oauth_session');
    $session->setExpirationSeconds($result['expires_in']);
    $session->accessToken = $result['access_token'];

    $response = new Result(
        Result::SUCCESS,
        $user,
        array('Authentication successful.')
    );
} else {
    $response = new Result(
        Result::FAILURE, NULL , array('Invalid credentials.')
    );
}
```

Right after calling the `authenticate()` method on the `ApiClient` object, we need to examine the response to see if it was successful and the API returned an access token; otherwise, we will return an error to the client. If the API returns an access token, we need to create a new container to store the data there and we will set up the expiration accordingly to the expiration sent by the API.

As you can see, we hydrate the user as before, but after that we create the container and we store the data inside.

Following the OAuth 2.0 flow

To illustrate the responses we get from the OAuth 2.0 protocol a little bit, let's take a look at the following request made on the command line:

```
curl -XGET http://zf2-api/api/wall/tbhot3ww
```

If we try to get the wall of a user without specifying the OAuth 2.0 access token as we did in the preceding `curl` request, we will receive the following response:

```
    {"errorCode":401,"errorMsg":"Unauthorized"}
```

Now let's try to follow the OAuth 2.0 flow in order to make a successful request. The first thing we need to do is log in to the system and get an access token as follows:

```
curl -XPOST http://zf2-api/api/users/login
--data-urlencode "username=tbhot3ww"
--data-urlencode "password=111111"
--data-urlencode "grant_type=client_credentials"
--data-urlencode "client_id=zf2-client"
--data-urlencode "client_secret=mysupersecretpass"
```

As you can see, we are sending the `username`, `password`, `grant_type`, `client_id`, and the `client_secret` parameters in the `curl` request. On our PHP code, the `ApiClient` object is taking care of this.

```
    {
        "access_token":"3c90975e915d84b444d420703b5d7aca73a4f101",
        "expires_in":3600,
        "token_type":"bearer",
        "scope":null
    }
```

This is how the response of the API will look. As you can see the response includes an access token, which can be used for future requests.

Finally, as we have the access token, we can perform another `curl` request including it to receive the wall data.

```
curl http://zf2-api/api/wall/tbhot3ww?access_token=3c90975e915d84b444d420
703b5d7aca73a4f101
```

This is how the request looks like with the access token. If you are following this example, remember that you will get a different access token than mine and you will have to update the `curl` request accordingly.

After issuing the request, you will receive a response that looks as follows:

```
    {"feed":[
        {
            "id":"2",
            "user_id":"1",
            "url":"http:\/\/www.google.es",
            "title":"Google",
            "created_at":"2013-06-0122:22:55",
            "updated_at":null
        },
        {
            "id":"2",
            "user_id":"1",
            ...
```

The challenge

In this chapter, we implemented a two-legged version of the OAuth 2.0 protocol between our API and the client. I will leave the implementation of the third-party access to the API using the components we have in place at the API level as an exercise.

Summary

Now our API is protected against access from unauthorized requests. ZF2 doesn't provide anything yet to help us in implementing an OAuth 2.0 server, but in this chapter we saw how easily we can import components from other sources.This means that we can also use components from other frameworks and libraries.

Congratulations! You made it! You arrived at the end of the last chapter, and now you have an impressive MVP of a social network. I hope you have learned, during this journey, as much as I did while writing it and I hope you enjoyed reading it and coding the examples.

Index