Learn by doing: less theory, more results

Ext GWT 2.0

Take the user experience of your website to a new level with Ext GWT

Beginner's Guide

Daniel Vaughan

[PACKT] PUBLISHING open source* community experience distilled

Ext GWT 2.0

Beginner's Guide

Take the user experience of your website to a new level with Ext GWT

Daniel Vaughan

[PACKT] PUBLISHING open source community experience distilled

Ext GWT 2.0

Beginner's Guide

First published: November 2010

Production Reference: 1191110

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849511-84-1

`www.packtpub.com`

Cover Image by John M. Quick (`john.m.quick@gmail.com`)

Credits

Author
Daniel Vaughan

Reviewers
Michal Kozik
Yiwen Ng (Tony)
Carl Pritchett

Acquisition Editor
Usha Iyer

Development Editor
Wilson D'souza

Technical Editors
Dayan Hyames
Pooja Pande

Copy Editors
Leonard D'Silva
Lakshmi Menon

Indexers
Hemangini Bari
Monica Ajmera Mehta

Editorial Team Leader
Aanchal Kumar

Project Team Leader
Ashwin Shetty

Project Coordinator
Zainab Bagasrawala

Proofreader
Mario Cecere

Graphics
Nilesh R. Mohite

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Daniel Vaughan has worked with enterprise web applications for over 12 years. He is currently a software architect for a UK financial institution. An experienced Java developer, Daniel first started working with Google Web Toolkit soon after it was released in 2006 and loved the power and simplicity it bought to web application development. When Ext GWT came along, he was an early adopter and he has used it as a part of several large projects.

Daniel currently splits his time between the beautiful tranquility of the Cotswold, England and the fast-moving city state of Singapore. He enjoys traveling, scuba diving, and learning new ideas.

I would like to thank Jason Brown, Bob Twiddy, Wayne Harris, Kirsty Harper, and Gwendolyn Regina Tan for their advice and encouragement while writing this book. I would also especially like to thank Lindy Wai and my family for all their support. Finally, I would like to remember my grandmother, Mary Vaughan, who died during the writing of this book. She would not have understood a word but would have been very proud.

About the Reviewers

Michal Kozik is currently working in Inofonova GmbH as a Senior Technology Analyst, developing applications for Telco companies. His area of expertise includes Java Standard Edition, Java Enterprise Edition, and Web Services.

Michal has received a Master's degree in Teleinformatics Systems from Cracow University of Technology in Cracow. During his spare time, he enjoys playing basketball and snowboarding.

Yiwen Ng (Tony) is a Java software developer with over 7 years of commercial application development and consulting experience. Fringe passions involve agile methodology, mobile development, web enterprise development, configuration management, and security. If cornered, he may actually admit to knowing Java's latest technologies and pair programming.

He's easily amused by programming language design and collaborative applications. Yiwen has also developed a few android mobile applications and RIA GWT-based web applications. Occasionally, he works as a consultant on a contractor basis. Yiwen can be reached directly via e-mail at `ttony@mjsoft.com.au`.

Currently, he is employed at Tullett Prebon in Singapore as a Senior Software Developer.

Carl Pritchett is an avid software developer with over 11 years of industry experience. He has worked on integration and pure software development projects for many companies including Novell, Insurance Australia Group Limited, and currently develops a financial research platform for Calibre Financial Technology. With extensive experience in the financial and insurance sectors, Carl's primary focus is on Java, JEE, and GWT with a healthy knowledge of C# and other technologies and products that help get the job done. He enjoys working with motivated people in agile/lean environments.

I'd like to acknowledge my workplace and my wife for their support in providing the time to review this book.

Table of Contents

Preface

Ext GWT 2.0: Beginner's Guide is a practical book that teaches you how to use the Ext GWT library to its full potential. It provides a thorough, no-nonsense explanation of the Ext GWT library, what it offers, and how to use it through practical examples. This book provides clear, step-by-step instructions for getting the most out of Ext GWT and offers practical examples and techniques that can be used for building your own applications in Ext GWT.

This book gets you up and running instantly to build powerful Rich Internet Applications (RIA) with Ext GWT. It then takes you through all the interface-building widgets and components of Ext GWT using practical examples to demonstrate when, where, and how to use each of them. Layouts, forms, panels, grids, trees, toolbars, menus, and many other components are covered in the many examples packed in this book. You will also learn to present your data in a better way with templates and use some of the most sought-after features of Ext GWT in your web applications such as drag-and-drop and charts. Throughout the book, a real application is built step-by-step using Ext GWT and deployed to Google App Engine.

Imagine how great you'll feel when you're able to create great-looking desktop-style user interfaces for your web applications with Ext GWT!

What this book covers

Chapter 1, *Getting Started with Ext GWT*, introduces Ext GWT and explains where it fits into GWT. It then moves on to show how to get up and running with Ext GWT by creating your first project.

Chapter 2, *The Building Blocks*, starts by looking at the explorer demo application. It then introduces the world of GXT components, beginning with some key concepts, and quickly moves on to practically working with an example application.

Chapter 3, *Forms and Windows*, explores GXT's form features. It looks at the form components that GXT provides and demonstrates how to put them to use. It also introduces the GXT Registry and shows how it can be used across the application.

Chapter 4, *Data-backed Components*, explains how GXT facilitates working with data. It looks at the components available for retrieving, manipulating, and processing data and then moves on to work with the built-in data-backed display components.

Chapter 5, *More Components*, introduces more advanced data-backed components and the extensions that build on the components covered in the previous chapter. It then moves on to cover additional advanced components—specifically menus, toolbars, and tabs.

Chapter 6, *Templates*, looks at templates and how they can be used to easily format and display data in a highly customizable way. It also introduces the more powerful features of XTemplates.

Chapter 7, *Model View Controller*, explains GXT's Model View Controller framework and demonstrates how it can allow components to communicate in larger applications.

Chapter 8, *Portal and Drag-and-Drop*, covers the portal and drag-and-drop features of GXT. It starts by showing how to turn out existing components into portlets and then moves on to *practically make use of GXT's drag-and-drop features to move data between them.*

Chapter 9, *Charts*, covers GXT's charting plugin. It explores the wide range of charts available, shows how to avoid the pitfalls of the plugin, and demonstrates how charts can be used with existing data.

Chapter 10, *Putting it all together*, shows how to publish the example application to the world using the Google App Engine. It then moves on to look at how to take development with GXT further and other resources that can be turned to after this book.

What you need for this book

1. Sun JDK version 6u21 available at `http://java.sun.com/javase/downloads/widget/jdk6.jsp`
2. Eclipse IDE for Java EE Developers version 3.6 available at `http://www.eclipse.org/downloads/`
3. Ext GWT SDK version 2.2.0 available at `http://www.sencha.com/products/gwt/download.php`
4. Google Plugin for Eclipse version 3.6 available at `http://code.google.com/eclipse/`
5. Google Web Toolkit version 2.1.0 available at `http://code.google.com/webtoolkit/download.html`
6. Google App Engine Java SDK version 1.3.8 available at `http://code.google.com/appengine/downloads.html`

Who this book is for

If you are a Java developer aspiring to build intuitive web applications with Ext GWT, then this book is for you. It assumes that you are familiar with HTML and CSS. Developers who wish to add an RIA look to their existing GWT applications with Ext GWT will find this book extremely useful.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also 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: "The `FirstGxtApp` class modifies the default GWT application to use GXT controls instead of the GWT equivalents."

A block of code is set as follows:

```
LayoutContainer layoutContainer = new LayoutContainer();
Button button = new Button("Click me");
layoutContainer.add(button);
RootPanel.get().add(layoutContainer);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
RootPanel.get().add(layoutContainer);
Button anotherButton = new Button("Click me too");
layoutContainer.add(anotherButton);
layoutContainer.layout();
```

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: " We would like to take advantage of our example application to pop up a small form for entering an URL when the user clicks on the **Link feed** button".

> 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 book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

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.

> **Downloading the example code for this book**
>
> 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/support`, 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 to address it.

1
Getting Started with Ext GWT

In this chapter, we introduce Ext GWT and explain where it fits into GWT. We then move on to show you how to get up and running with Ext GWT by creating your first project.

In this chapter, we will cover:

- Installing Ext GWT
- Creating a new GWT project
- Preparing the GWT project to use Ext GWT
- Adapting the GWT example application to use Ext GWT components

What is GWT missing?

The Google Web Toolkit is a great way for Java developers to create AJAX-based rich Internet applications without requiring in-depth knowledge of JavaScript or having to deal with the quirks of different browsers. However, it is a toolkit as opposed to a full development framework, and for most projects, it forms the part of a solution rather than the whole solution.

Out-of-the-box GWT comes with only a basic set of widgets and lacks a framework to enable the developers to structure larger applications. Fortunately, GWT is both open and extensible and as a result, a range of complementary projects have grown up around it. Ext GWT is one of those projects.

What does Ext GWT offer?

Ext GWT sets out to build upon the strengths of GWT by enabling the developers to give their users an experience more akin to that of a desktop application.

Ext GWT provides the GWT developer with a comprehensive component library similar to that used when developing for desktop environments. In addition to being a component library, powerful features for working with local and remote data are provided. It also features a model view controller framework, which can be used to structure larger applications.

How is Ext GWT licensed?

Licensing is always an important consideration when choosing technology to use in a project. At the time of writing, Ext GWT is offered with a dual license.

The first license is an open source license compatible with the GNU GPL license v3. If you wish to use this license, you do not have to pay a fee for using Ext GWT, but in return you have to make your source code available under an open source license. This means you have to contribute all the source code of your project to the open source community and give everyone the right to modify or redistribute it.

If you cannot meet the obligations of the open source license, for example, you are producing a commercial product or simply do not want to share your source code, you have to purchase a commercial license for Ext GWT.

It is a good idea to check the current licensing requirements on the Sencha website, `http://www.sencha.com`, and take that into account when planning your project.

Alternatives to Ext GWT

Ext GWT is one of the many products produced by the company Sencha. Sencha was previously named Ext JS and started off developing a JavaScript library by the same name. Ext GWT is closely related to the Ext JS product in terms of functionality. Both Ext GWT and Ext JS also share the same look and feel as well as a similar API structure. However, Ext GWT is a native GWT implementation, written almost entirely in Java rather than a wrapper, the JavaScript-based Ext JS.

GWT-Ext

Before Ext GWT, there was GWT-Ext: `http://code.google.com/p/gwt-ext/`. This library was developed by Sanjiv Jeevan as a GWT wrapper around an earlier, 2.0.2 version of Ext JS. Being based on Ext JS, it has a very similar look and feel to Ext GWT. However, after the license of Ext JS changed from LGPL to GPL in 2008, active development came to an end.

Apart from no longer being developed or supported, developing with GWT-Ext is more difficult than with Ext GWT. This is because the library is a wrapper around JavaScript and the Java debugger cannot help when there is a problem in the JavaScript code. Manual debugging is required.

Smart GWT

When development of GWT-Ext came to an end, Sanjiv Jeevan started a new project named Smart GWT: `http://www.smartclient.com/smartgwt/`. This is a LGPL framework that wraps the Smart Client JavaScript library in a similar way that GWT-Ext wraps Ext JS. Smart GWT has the advantage that it is still being actively developed. Being LGPL-licensed, it also can be used commercially without the need to pay the license fee that is required for Ext GWT. Smart GWT still has the debugging problems of GWT-Ext and the components are often regarded not as visually pleasing as Ext GWT. This could be down to personal taste of course.

Vaadin

Vaadin, `http://vaadin.com`, is a third alternative to Ext GWT. **Vaadin** is a server-side framework that uses a set of precompiled GWT components. Although you can write your own components if required, Vaadin is really designed so that you can build applications by combining the ready-made components.

In Vaadin the browser client is just a dumb view of the server components and any user interaction is sent to the server for processing much like traditional Java web frameworks. This can be slow depending on the speed of the connection between the client and the server.

The main disadvantage of Vaadin is the dependency on the server. GWT or Ext GWT's JavaScript can run in a browser without needing to communicate with a server. This is not possible in Vaadin.

Ext GWT or GXT?

To avoid confusion with GWT-Ext and to make it easier to write, Ext GWT is commonly abbreviated to **GXT**. We will use GXT synonymously with Ext GWT throughout the rest of this book.

Working with GXT: A different type of web development

If you are a web developer coming to GXT or GWT for the first time, it is very important to realize that working with this toolset is not like traditional web development. In traditional web development, most of the work is done on the server and the part the browser plays is little more than a view-making request and receiving responses.

When using GWT, especially GXT, at times it is easier if you suspend your web development thinking and think more like a desktop-rich client developer. Java Swing developers, for example, may find themselves at home.

How GXT fits into GWT

GXT is simply a library that plugs into any GWT project. If we have an existing GWT project setup, all we need to do to use it is:

- Download the GXT SDK from the Sencha website
- Add the library to the project and reference it in the GWT configuration
- Copy a set of resource files to the project

If you haven't got a GWT project setup, don't worry. We will now work through getting GXT running from the beginning.

Downloading what you need

Before we can start working with GXT, we first need to download the toolkit and set up our development environment. Here is the list of what you need to download for running the examples in this book.

Recommended	Notes	Download from
Sun JDK 6	The Java development kit	`http://java.sun.com/javase/downloads/widget/jdk6.jsp`
Eclipse IDE for Java EE Developers 3.6	The Eclipse IDE for Java developers, which also includes some useful web development tools	`http://www.eclipse.org/downloads/`
Ext GWT 2.2.0 SDK for GWT 2.0	The GXT SDK itself	`http://www.sencha.com/products/gwt/download.php`

Google supplies a useful plugin that integrates GWT into Eclipse; it makes sense for us to use Eclipse in this book. However, there is no reason that you cannot use an alternative development environment, if you prefer.

Eclipse setup

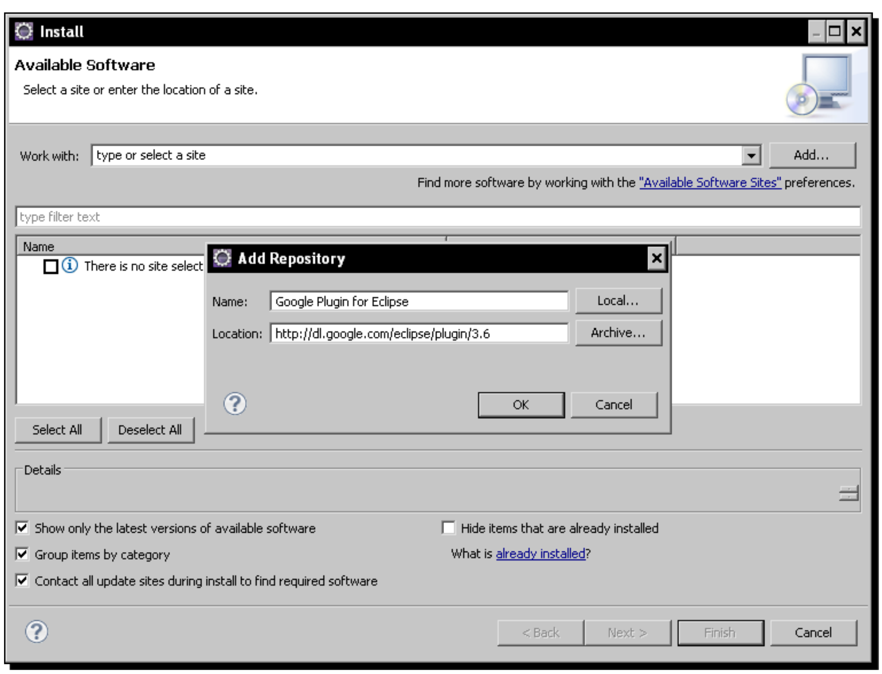
There are different versions of Eclipse, and although Eclipse for Java EE developers is not strictly required, it contains some useful tools for editing web-specific files such as CSS. These tools will be useful for GXT development, so it is strongly recommended. We will not cover the details of installing Eclipse here, as this is covered more than adequately on the Eclipse website. For that reason, we make the assumption that you already have a fresh installation of Eclipse ready to go.

GWT setup

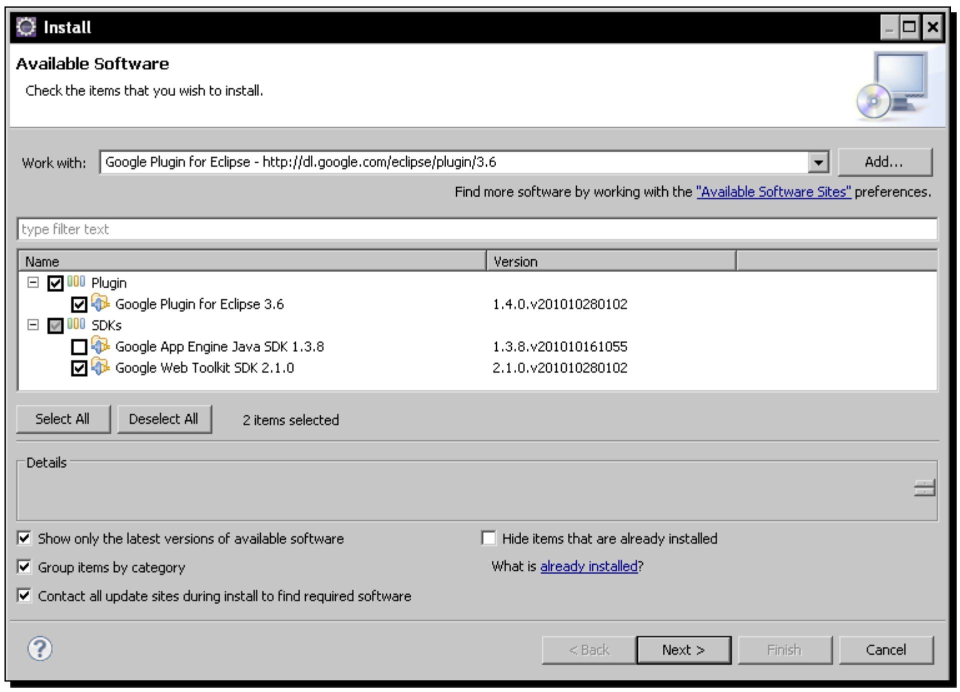
You may have noticed that GWT is not included in the list of downloads. This is because since version 2.0.0, GWT has been available within an Eclipse plugin, which we will now set up.

Time for action – setting up GWT

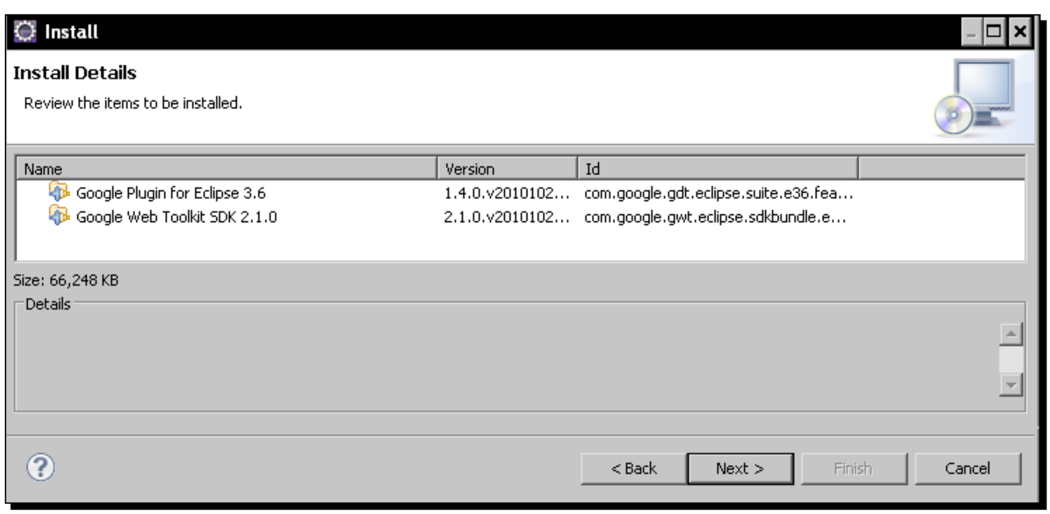
1. In Eclipse, select **Help | Install New Software**. The installation dialog will appear.
2. Click the **Add** button to add a new site.
3. Enter the name and location in the respective fields, as shown in the following screenshot, and click on the **OK** button.



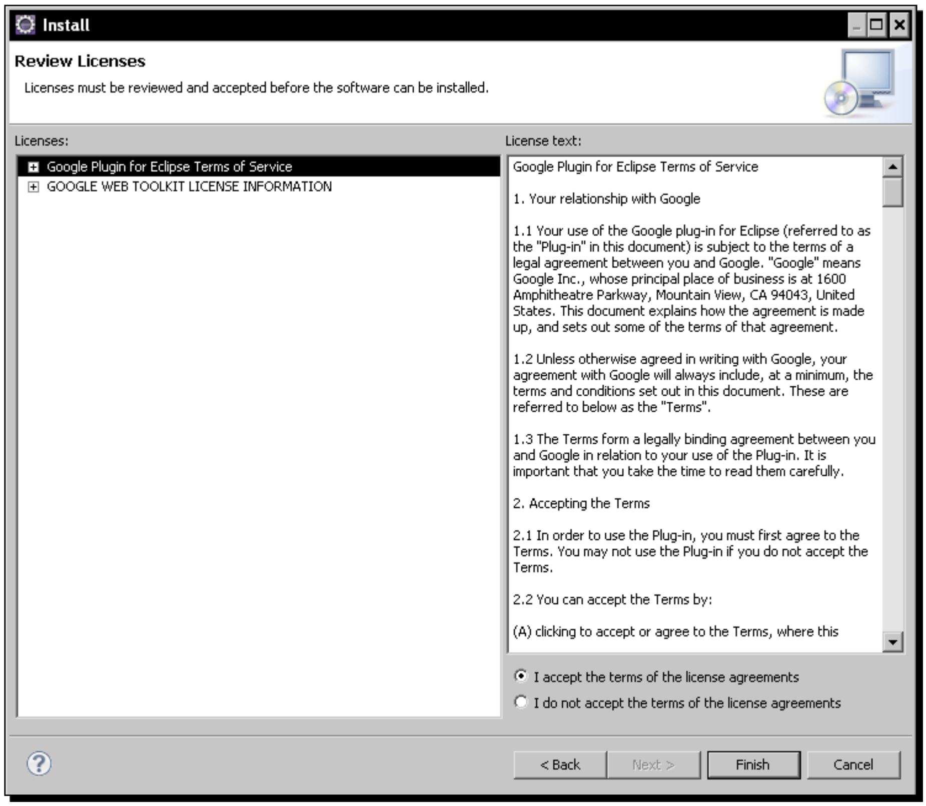
4. Select **Google Plugin for Eclipse** from the plugin section and **Google Web Toolkit SDK** from the SDKs section. Click on **Next**.



5. The following dialog will appear. Click on **Next** to proceed.

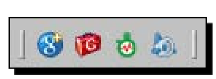


6. Click the radio button to accept the license. Click on **Finish**.



7. Eclipse will now download the Google Web Toolkit and configure the plugin. Restart when prompted.

8. On restarting, if GWT and the Google Eclipse Plugin are installed successfully, you will notice the following three new icons in your toolbar.



What just happened?

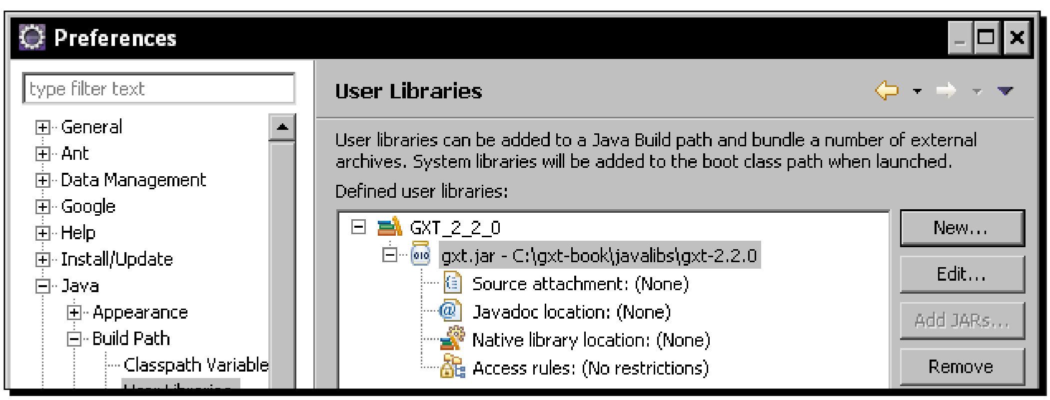
You have now set up GWT in your Eclipse IDE. You are now ready to create GWT applications. However, before we can create GXT applications, there is a bit more work to do.

GXT setup

Having downloaded the GXT SDK and extracted the zip file to a convenient location, we now need to configure Eclipse.

Time for action – setting up GXT

1. In Eclipse, select **Window | Preferences**.
2. From the tree, select **Java | Build Path | User Libraries**.
3. Create a new user library by selecting the new button and enter the name `GXT_2_2_0`.
4. Select the library you have just created and click on the **Add JARs** button.
5. Select the `gxt.jar` file from the location where you extracted the ZIP file.



What just happened?

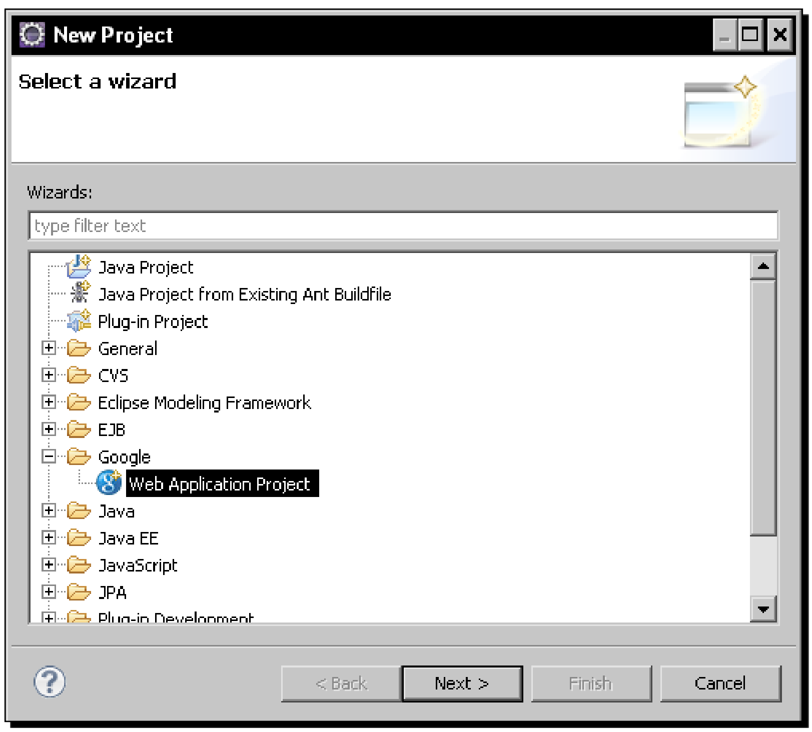
We have now set up GXT in Eclipse. At this point, we have everything in place and we are ready to test our development environment by creating our first GXT application.

GWT project creation

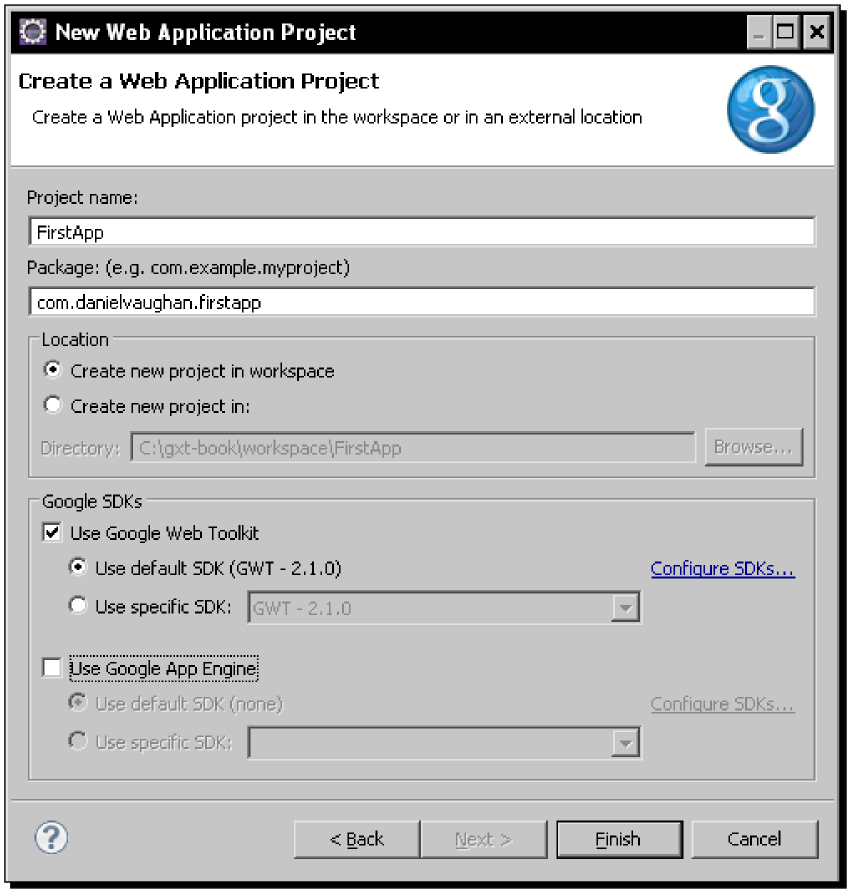
The development environment is ready to go. So let's create a GWT project to base our first GXT application on.

Time for action – creating a GWT project

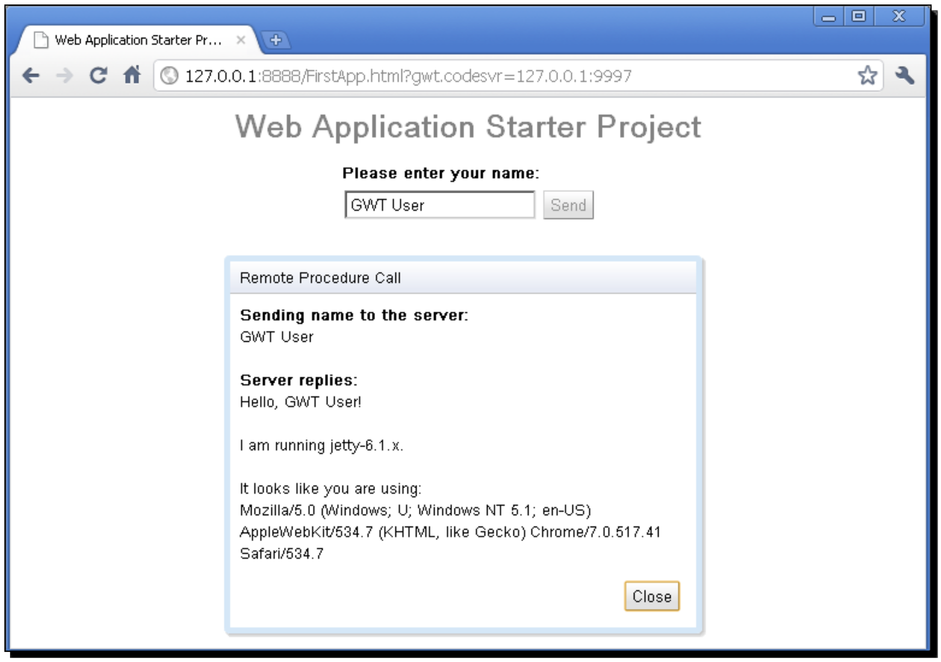
1. First, create a GWT project by going to **File | New | Project**.
2. From the dialog, select **Google** and then **Web Application Project** from the `Google` folder. Click on the **Next** button.



3. Enter the project name and package, as shown in the following screenshot and then click on **Finish**.



4. You will now have created a default GWT application. On running it as a web application, you will see the following in your browser:



What just happened?

We have created a new project comprising the default GWT application. At this stage, it is a pure GWT app.

GXT project configuration

We now need to make changes to the GWT application to enable it to make use of GXT.

This consists of the following steps:

- Include the GXT library
- Add an entry for GXT to the GWT module file
- Modify the HTML host file
- Copy resources

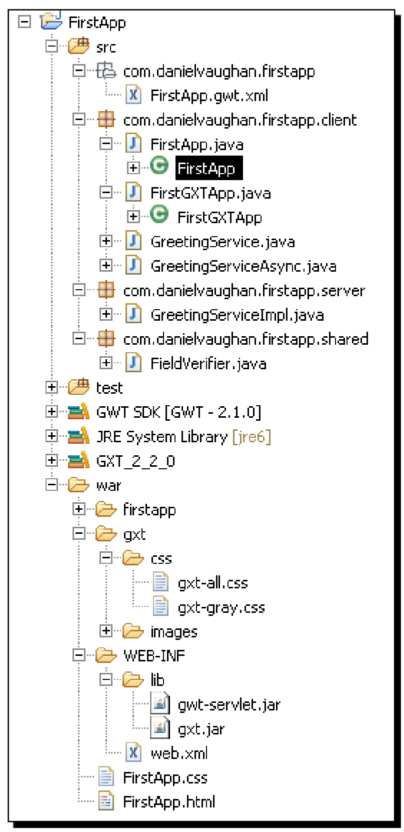
Time for action – preparing the project to use GXT

1. Earlier we set up a GXT user library. We now need to include it to the build path of our newly created GWT project and the `lib` folder of the `war` folder.

   Build path: Right-click on the **FirstApp** project and select **Properties**. Select **Java Build Path** and then select the **Libraries** tab. Click on the **Add Library** button, select **User Library** and click on the **Next** button. Now select the **GXT_2_2_0** user library. Click on the **Finish** button and then on **OK**.

   War: Copy the `gxt.jar` file to the `war\WEB-INF\lib` folder of your project.

   Your project structure should now look like this:



2. The GWT module file contains the entry point for a GWT application together with references to any additional libraries it uses. The module file always ends in `gwt.xml` and is in the root package of the source folder. In this case, it is named `FirstApp.gwt.xml`. In order to use GXT, there needs to be an entry added to this file.

The default GWT module file also contains a reference to the default GWT style sheet. This can be removed.

The line that we need to add should be put in the "Other module inherits" section as follows:

```
<inherits name='com.extjs.gxt.ui.GXT' />
```

The complete file should now look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
  <module rename-to='firstapp'>
    <!-- Inherit the core Web Toolkit stuff.
    -->
    <inherits name='com.google.gwt.user.User' />

    <!-- Other module inherits
    -->
    <inherits name='com.extjs.gxt.ui.GXT' />

    <!-- Specify the app entry point class.
    -->
    <entry-point
      class='com.danielvaughan.firstapp.client.FirstApp'
    />

    <!-- Specify the paths for translatable code
    -->
    <source path='client' />

  </module>
```

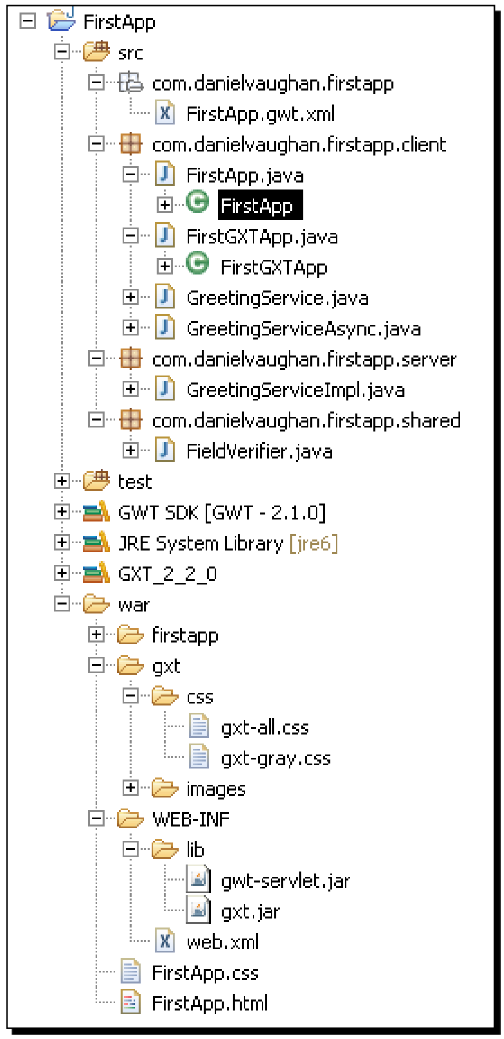
3. We now need to modify the host HTML file. In this project, it is named `FirstApp.html` and is located in the `war` folder. Edit this file, including the GXT stylesheets, by adding the following line into the head section beneath the existing stylesheet link:

```
<link type="text/css" rel="stylesheet" href="gxt/css/gxt-all.css">
```

4. Finally, we need to copy the GXT stylesheet and image resources into the project's `war` folder.

   Create a folder named `gxt` in the `war` folder, go to the location where you originally unzipped your downloaded GXT package, and open the resources folder. Now copy both the `css` and `images` folders into the newly created `gxt` folder.

Your `war` folder should now look like this:



What just happened?

We have configured our project so that it now has all the dependencies it needs for making use of GXT features.

Differences of GXT controls

Our application now includes the GXT library, but as yet it is not making any use of the library. In the example code of this chapter, we have left in the original `FirstApp` class together with a `FirstGxtApp` class. The `FirstGxtApp` class modifies the default GWT application to use GXT controls instead of the GWT equivalents. By comparing these, you can see how, although similar, GXT controls do have some differences in how they can be used. We will now summarize the main differences.

Time for action – adapting the GWT app to use GXT controls

1. When we created the GWT application, a class named `FirstApp` will be created. We created a copy of that class named `FirstGxtApp`.

2. In the imports section of the `FirstGxtApp` class, we removed the following GWT specific imports:

```
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DialogBox;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;
```

3. We then added imports to the equivalent GXT classes as follows:

```
import com.extjs.gxt.ui.client.event.ButtonEvent;
import com.extjs.gxt.ui.client.event.SelectionListener;
import com.extjs.gxt.ui.client.event.KeyListener;
import com.extjs.gxt.ui.client.event.ComponentEvent;
import com.extjs.gxt.ui.client.widget.Dialog;
import com.extjs.gxt.ui.client.widget.Label;
import com.extjs.gxt.ui.client.widget.VerticalPanel;
import com.extjs.gxt.ui.client.widget.button.Button;
import com.extjs.gxt.ui.client.widget.form.TextField;
```

You may notice that some of the GXT classes share a similar name to their GWT equivalents. The following table shows the GXT classes we used and the GWT equivalents:

GXT	GWT
com.extjs.gxt.ui.client.widget.Dialog	com.google.gwt.user.client.ui.DialogBox
com.extjs.gxt.ui.client.widget.Label	com.google.gwt.user.client.ui.Label
com.extjs.gxt.ui.client.widget.VerticalPanel	com.google.gwt.user.client.ui.VerticalPanel
com.extjs.gxt.ui.client.widget.button.Button	com.google.gwt.user.client.ui.Button
com.extjs.gxt.ui.client.widget.form.TextField	com.google.gwt.user.client.ui.TextBox
com.extjs.gxt.ui.client.event.ButtonEvent	com.google.gwt.event.dom.client.ClickEvent
com.extjs.gxt.ui.client.event.SelectionListener	com.google.gwt.event.dom.client.ClickHandler
com.extjs.gxt.ui.client.event.KeyListener	com.google.gwt.event.dom.client.KeyUpEvent
com.extjs.gxt.ui.client.event.ComponentEvent	com.google.gwt.event.dom.client.KeyUpHandler

4. We then needed to redefine the controls. In the GWT example, all the code sits inside the `onModuleLoad` method and makes use of inner classes. However, due to the way listeners are implemented in GXT, we lose some of the flexibility that enables this. Instead, we had to define the controls as private members as follows:

```
private final Button sendButton = new Button("Send");
private final TextField<String> nameField = new
              TextField<String>();
private final Dialog dialogBox = new Dialog();
private final Label textToServerLabel = new Label();
private final HTML serverResponseLabel = new HTML();
```

5. There are differences in syntax between the GXT and GWT methods. Although the GXT controls are similar to GWT controls, there are a number of differences. Firstly, there are many small differences on the methods of the controls between GWT and GXT. Here are the ones we see in this example:

GXT	GWT
`TextField.setValue()`	`TextBox.setText()`
`TextField.focus()`	`TextBox.setFocus(true)`
`DialogBox.setHeading()`	`DialogBox.setText()`
`DialogBox.setAnimCollapse(true)`	`DialogBox.setAnimationEnabled(true)`
`VerticalPanel .setHorizontalAlign(HorizontalAlignment.RIGHT);`	`VerticalPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT)`

6. Another difference that is important is that while GWT now uses event handlers for events such as clicking on a button, GXT uses event listeners similar to the earlier version of GWT. However, in this case, the actual code is very similar.

   Here is how you implement the close button click event in GWT using a click handler:

```
closeButton.addClickHandler(new ClickHandler()
{
  public void onClick(ClickEvent event)
    {
      dialogBox.hide();
      sendButton.setEnabled(true);
      sendButton.setFocus(true);
    }
});
```

   Here is the same thing in GXT using a selection listener:

```
closeButton.addSelectionListener(new
            SelectionListener<ButtonEvent>()
{
  public void componentSelected(ButtonEvent ce)
    {
      dialogBox.hide();
      sendButton.setEnabled(true);
      sendButton.focus();
    }
});
```

7. We now have two classes: the original GWT `FirstApp` class and our new `FirstGXTApp` class. To use the `FirstGXTApp`, we need to change the application's `gwt.xml` module file to use the `FirstGXTApp` instead of `FirstApp`.

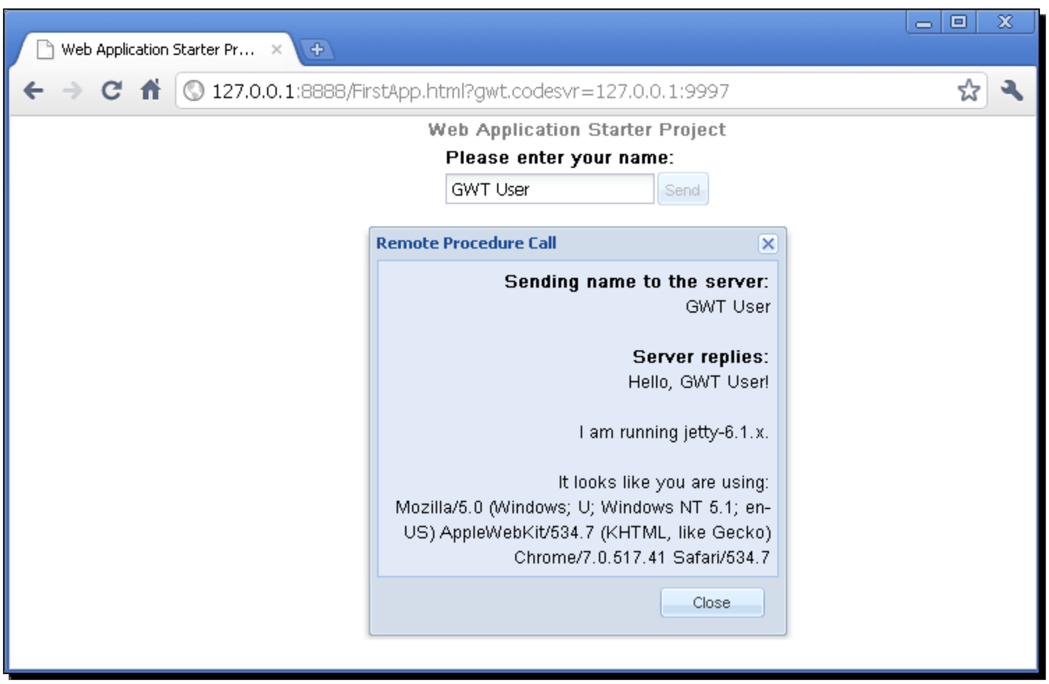
Open `FirstApp.gxt.xml` and change the entry point element from:

```
<entry-point class='com.danielvaughan.firstapp.client.FirstApp' />
```

to:

```
<entry-point class='com.danielvaughan.firstapp.client.FirstGXTApp'
/>
```

Now when running the web application again, you will see a new version with GXT controls.



What just happened?

Hopefully, you now can see that using GXT is not vastly different from using GWT. It is also important to realize that there are some subtle differences. Over the coming chapters, we will show that there are many great features in GXT that go far beyond the basics provided by GWT.

Pop quiz – introducing GXT

1. What JavaScript library is GXT closely related to?
2. Which GXT alternative wraps the Smart Client JavaScript library?
3. Which GXT alternative does most of the work on the server?
4. Which GXT alternative has a name and appearance that is easily confused with Ext GWT?
5. What is the name of the company that develops GXT?
6. What is the name of the GXT Java library file?
7. What is the license of GXT?
8. In what file must you inherit the GXT module?
9. Where must you include a reference to the GXT CSS?
10. Where must you copy the `gxt.jar` library file?

Summary

In this chapter, we have introduced GXT and set up the development environment. We then went on to modify the standard GWT sample application to use the GXT component. We used this to highlight the similarities and differences between GXT and GWT. In the next chapter, we will start delving into the GXT components in more depth.

2

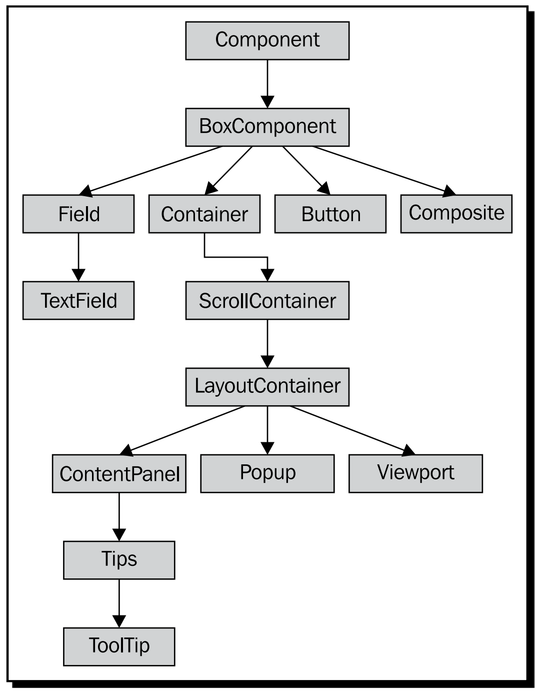
The Building Blocks

Now that we have set up our development environment, in this chapter, we are ready to take a proper look at GXT. We start by looking at the explorer demo application. We then introduce the world of GXT components, beginning with some key concepts, and quickly move on to practically working with an example application.

In this chapter, we shall learn about the following GXT features:

- `Component`
- `Container`
- `BoxComponent`
- `ScrollContainer`
- `LayoutContainer`
- `FlowLayout`
- `ContentPanel`
- `Viewport`
- `BorderLayout`
- `Loading messages`
- `Custom Components`
- `Buttons`
- `Tooltip`
- `Popup`
- `SelectionListener`
- `TextField`
- `KeyListener`

The following diagram shows how the components covered in this chapter fit together and it may be useful to refer back to it as the chapter goes on:



The Ext GWT Explorer Demo

The GXT package includes a sample called The Ext GWT Explorer. This demo showcases all the different components available in GXT. It also provides sample code that shows you how to use them. This is an invaluable tool for understanding what is available and for giving you an idea of the code required to make use of each component.

The explorer application is also hosted on the Sencha website and can be found here: `http://www.sencha.com/examples/explorer.html`. If you have not done so already, it is a really good idea to go and have a good look at the Explorer application before starting this chapter, as it shows you pretty much all the components available in GXT.

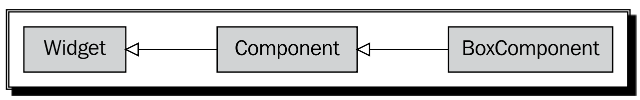
Essential knowledge

As you should have seen from looking at the Explorer applications, there are a wide range of components available to use in GXT. Some components are more complex than others, but they are all built on the same foundations. By understanding a few basic concepts, you can soon get to grips with them.

GXT building block 1: Component

In GWT, **Widget** (`com.google.gwt.user.client.ui.Widget`) is the base class of all visible elements we can see in a browser such as buttons, textboxes, and tree views, for example.

In GXT, the base class for all visual elements is **Component**, `com.extjs.gxt.ui.client.widget.Component`. As GXT is built on top of GWT, it should not come as a surprise that all of GXT components are based on GWT **Widget**. More formally, the GXT **Component** class subclasses **Widget** and introduces a number of new features:



All GXT's components participate in GXT's life cycle of creation, attach and detach automatically and use lazy rendering. Components inherit basic hide and show, enable and disable functionality.

BoxComponent

All GXT's visual elements inherit from Component, either directly or indirectly using BoxComponent. Components that subclass **BoxComponent** inherit sizing and methods additionally.

Components that subclass Component directly are those that don't exist outside of a containing component. For example, TreeItem subclasses Component, as it only exists inside a TreePanel. Any component that can be positioned or resized, such as a Button, TreePanel, or Grid is a subclass of BoxComponent.

Lazy Rendering

GWT works by manipulating elements of the DOM, the Document Object Model representation of the HTML page in the browser. GWT widgets are pieces of HTML that are added to and removed from the DOM.

For example, a GWT `button` widget will have HTML that looks like this:

```
<button type="button" class="gwt-Button" style="position: absolute;
left: 80px; top: 45px; ">Click Me!</button>
```

In GWT when the widget is initialised, the HTML is created at the same time. When a widget is added to a GWT panel, the HTML has already been created.

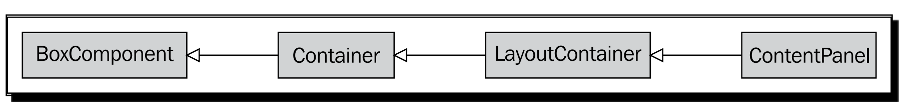
GXT components are different as they use lazy rendering. When a GXT Component is initialised, the HTML is not created straightaway. The HTML is only created when the render method is called on the component. That way the HTML is not created until it is needed to be added to the DOM. This approach is understandably more efficient, as it avoids unused HTML sitting in memory.

Although the HTML for a component is not created until the component is rendered, properties of components can be configured before they are rendered. For example, a TextField can be set up with a value before it is added to a Container and the HTML is generated.

If a GXT Component is added to a GWT panel, the render method will be called straightaway. If the same Component is added to a GXT equivalent, it is the Container that calls the render method of the Component.

GXT building block 2: Container

Containers are a type of BoxComponent that can contain other components. They subclass the `com.extjs.gxt.ui.client.widget.Container<T>` class, and have the ability to attach, detach, and manage their child components. Container itself does not deal with the laying out and rendering of components. This is left to subclasses.



LayoutContainer

The **LayoutContainer** inherits from **Container** indirectly by subclassing the ScrollContainer class. ScrollContainer adds support for the scrolling of content to Container. LayoutContainer itself adds the ability to lay out the child components using a Layout.

Let's see how this works with the idea of lazy rendering. First of all, we will create a `LayoutContainer`:

```
LayoutContainer layoutContainer = new LayoutContainer();
```

At this point, no HTML has been created because the LayoutContainer has not been added to either another GXT Container or a GWT Panel. Now we add a `Button`:

```
LayoutContainer layoutContainer = new LayoutContainer();
Button button = new Button("Click me");
```

Again, we have a LayoutContainer and a Button, but still no HTML has been created as nothing is rendered. We add the `Button` to the `LayoutContainer`:

```
LayoutContainer layoutContainer = new LayoutContainer();
Button button = new Button("Click me");
layoutContainer.add(button);
```

Even though a Button has been added to a Container, still no HTML will be created. This is because the LayoutContainer itself has not been rendered. However, if we add the `LayoutContainer` to a GWT Panel, in this case the `RootPanel`, things start to happen:

```
LayoutContainer layoutContainer = new LayoutContainer();
Button button = new Button("Click me");
layoutContainer.add(button);
RootPanel.get().add(layoutContainer);
```

Adding the `LayoutContainer` to the `RootPanel` will cause the render method of the LayoutContainer to be called. Containers use a system of cascading layout, so when the LayoutContainer is rendered, it will call the render method of each of its children, in this case the single Button. HTML will only now be generated for both the LayoutContainer and the Button and be added to the DOM.

If we now wanted to add a second button to the `LayoutContainer`, we could do it like this:

```
LayoutContainer layoutContainer = new LayoutContainer();
Button button = new Button("Click me");
layoutContainer.add(button);
RootPanel.get().add(layoutContainer);
Button anotherButton = new Button("Click me too");
layoutContainer.add(anotherButton);
```

You may think that this would make a second button appear in the LayoutContainer, but you would be wrong. The HTML for the second button will not have been created, as the LayoutContainer has already been rendered. In this case, we need to call the `layout` method of the LayoutContainer. This will call the render method for both Button components. The HTML for the second `Button` will then be added to the DOM.

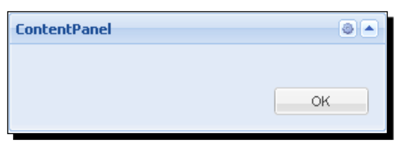
```
LayoutContainer layoutContainer = new LayoutContainer();
Button button = new Button("Click me");
layoutContainer.add(button);
RootPanel.get().add(layoutContainer);
Button anotherButton = new Button("Click me too");
layoutContainer.add(anotherButton);
layoutContainer.layout();
```

FlowLayout

FlowLayout is the default layout for a LayoutContainer. FlowLayout adds components to the container, but does not do anything regarding the sizing and positioning of the child components. The first component is rendered in the top left corner of the container, and each subsequent component is added to the right of the previous component. Later in the book, we will look at alternative layouts more closely.

ContentPanel

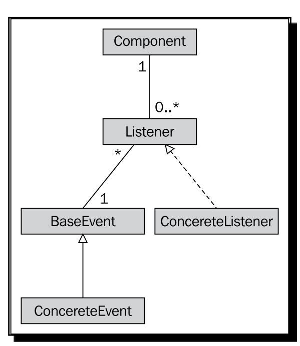
ContentPanel subclasses LayoutContainer and is a very useful building block for user interfaces in general and the interface that we will be developing later in this chapter. It features separate header, footer, and body sections, and can display top and bottom toolbars. ContentPanel also has the built-in ability to collapse and expand and have a number of predefined tool buttons that can be shown in the header to be used for custom functions. Here is what a **ContentPanel** can look like with the collapse and custom "gear" tool in the header:



GXT building block 3: Events

Events are the concepts used for informing the program that something has happened. This can be the user interacting in some way with the application such as clicking on a button or the state of a component changing. Each action causes an event to be fired and gives any component that is listening for the event the opportunity to respond.

More formally, this is known as the observer pattern. In GXT, listeners can be added to components so that when an event is fired, any listeners are notified and can handle the event.



The base class for event is `com.extjs.gxt.ui.client.event.BaseEvent` and GXT provides a wide range of events. We will cover a few of these later in this chapter and many more in later chapters.

Sinking and swallowing events

As part of GWT's design, widgets respond to some, but not all browser events, and this also applies to GXT. The reason for this is to keep memory usage down and to avoid memory leaks. If a widget needs to respond to a browser event, it needs to register that event by calling the `sinkEvents` method. For example, by default a Component may respond to an onClick event, but not to an onDoubleClick. You can extend the component to respond to a double-click by sinking the onDoubleClick event.

In a similar way, you can also swallow events to stop events being fired. For example, if you were to swallow the onClick event of a button, it would no longer fire an event when clicked on.

Introducing the example application

The example application that we will use in the book from this point onwards is an RSS reader. This application will give us a chance to exploit nearly all of the functions of GXT, including many of the more advanced ones. But first of all we need to put the basics in place.

The requirement

Our customer has a requirement for an easy-to-use RSS news feed reader that can handle multiple RSS feeds specified by the user. The application should be available on the Web, but must look and feel as much as possible the same as a conventional desktop application.

We have no control over the browser our potential users may have installed and no control over their screen resolutions, so our application must be as flexible as possible.

The solution

As GWT produces optimised cross-browser JavaScript, it is in a good position to meet these requirements. By adding GXT, we can make the application behave much more like a desktop application than GWT on its own, without losing any of GWT's flexibility.

Blank project

We now need to create a new GXT project for our example application. This is done by following the steps we went through in *Chapter 1* to create our first GWT project, and add GXT support, except that this time instead of **FirstApp**, we name the project **RSSReader**. This time, however, we don't want to make use of the default GWT code, so we need to trim down the default application.

Time for action – creating a blank project

1. Delete the following files:
    - `GreetingService.java`
    - `GreetingServiceAsync.java`
    - `GreetingServiceImpl.java`
    - `FieldVerifier.java`
2. Remove the content of the RSSReader class leaving only the definition of `onModuleLoad` method.
3. Remove the greet servlet definition from the project's `web.xml` file in `war\WEB-INF` so that the file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Default page to serve -->
  <welcome-file-list>
    <welcome-file>RSSReader.html</welcome-file>
  </welcome-file-list>

</web-app>
```

4. Edit the `RSSReader.html` file in the `war` directory so that it only contains the minimum code we need for this project as shown below. Note that having a valid `DOCTYPE` is important in order for GXT to render correctly:

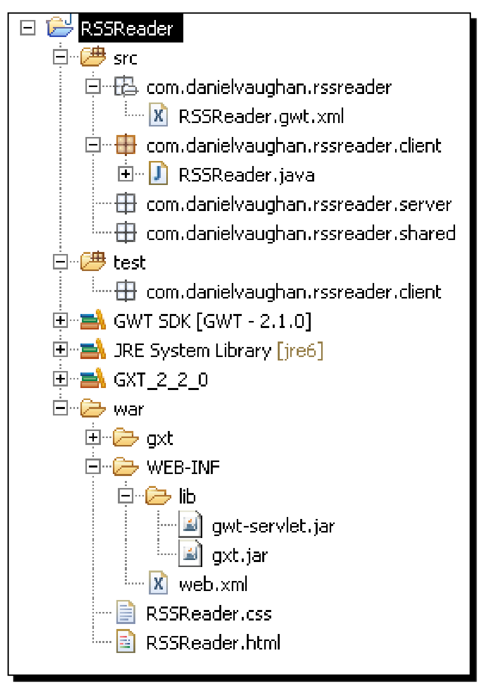
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

```
  <head>
    <meta http-equiv="content-type" content="text/html;
      charset=UTF-8">
    <link type="text/css" rel="stylesheet" href="RSSReader.css">
    <link type="text/css" rel="stylesheet" href="gxt/css/gxt-
      all.css">
    <title>RSSReader</title>
    <script type="text/javascript" language="javascript"
      src="rssreader/rssreader.nocache.js"></script>
  </head>
  <body>
  </body>
</html>
```

5. GWT will also create some default CSS code. As we don't need this, open `RSSReader.css` and delete the content, leaving it as a blank file.

6. If you compile and run the application now, all you should get is a blank page with the title **RSSReader** in your web browser.

7. The structure of your project should now look like this:



What just happened?

We removed all the GWT example code from a new project leaving us with an empty project to begin with.

Viewport

Viewport is a subclass of LayoutContainer that fills the browser window and then monitors the window for resizing. It then triggers any child components to be resized to deal with the new window size. It is a useful component when building an application that the user expects to behave like a desktop application.

We will use a Viewport as the base panel for our application. As such, it will be added directly to GWT's root panel. The viewport lays itself out automatically when it is added to the root panel so that it is not necessary to call the `layout()` method.

Time for action – adding a Viewport

1. In the `RSSReader.java` source file, add the following to the `onModuleLoad` method to create a new `Viewport` and add it to the application's `RootPanel` so that it looks like this:

```
public void onModuleLoad() {
  Viewport viewport = new Viewport();
  RootPanel.get().add(viewport);
}
```

2. If we started the application in the browser, it would still look blank. So to prove that the Viewport is there, open the `RSSReader.css` file in the `war` directory and add a css definition for the `.x-viewport` class:

```
.x-viewport
{
  background-color: #070;
}
```

3. The `.x-viewport` class is the default style for GXT Viewport components, and by adding this definition, we are making its background dark green. Now when we start the application, the browser window will initially be empty and white until the JavaScript code executes and the Viewport is rendered. When this happens, the browser window will turn dark green:

What just happened?

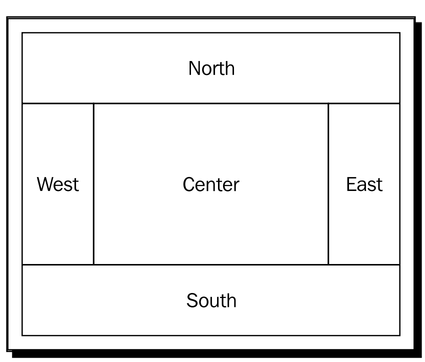
We created a Viewport, added it to GWT's root panel, and highlighted it in green to prove that the Viewport had loaded and took up the full screen.

Layout

Layout classes define how components added to a LayoutContainer are positioned and displayed. The base class for layouts is `com.extjs.gxt.ui.client.widget.Layout`. We will cover more layouts as this book goes on, but for the time being, we will be working with the BorderLayout.

BorderLayout

The BorderLayout provides a very convenient way to lay out the components of a fullscreen application. It allows us to split a layout component into a number of layout regions: a **center** region and then other regions around it in a compass fashion—**north**, **south**, **east**, and **west**. It supports the resizing of regions by the user by means of split bars and allows regions to be expanded and collapsed or hidden:



This type of layout is very common on websites, with the **north** being the header, the **south** the footer, the **center** the content, and the **west** and or the **east** being the navigation.

In our case, we are only going to make use of the **north**, **west** and **center** layout regions.

BorderLayoutData

Before adding a child component to a parent component that is laid out using a BorderLayout, we first need to define how that component will behave once it is added using a BorderLayoutData object.

When creating a BorderLayoutData object, we have to define which layout region it applies to, and optionally its initial size and a maximum and minimum size for the region.

Once created, there are also a number of other settings that can be defined such as if the region can be collapsed or split (resized) by the user.

When we have defined a BorderLayoutData object, we can use it to add a component to a Container that uses the BorderLayout.

We will now make use of BorderLayout in our example application.

Time for action – using BorderLayout

1. In the `onModuleLoad` method of the example application class, create a new `BorderLayout` and set the `Viewport` to use it:

```
public void onModuleLoad() {
  Viewport viewport = new Viewport();
  final BorderLayout borderLayout = new BorderLayout();
  viewport.setLayout(borderLayout);
  RootPanel.get().add(viewport);
}
```

2. Now create `BorderLayoutData` for the north layout region setting it to be `20`px high and neither collapsible nor resizable:

```
BorderLayoutData northData = new BorderLayoutData(LayoutRegion.
NORTH,20);
northData.setCollapsible(false);
northData.setSplit(false);
```

3. We can then create an HTML widget to use as the header and add it to the `viewport` in the north position using the `BorderLayoutData` we defined in the last step:

```
HTML headerHtml = new HTML();
headerHtml.setHTML("<h1>RSS Reader</h1>");
viewport.add(headerHtml, northData);
```

4. Now we define the `BorderLayoutData` for the `central` and `west` layout regions, the `west` region being collapsible and resizable. We define it as being `200`px wide initially, but also specify that it cannot be less than `150`px or more than `300`px wide:

```
BorderLayoutData centerData = new BorderLayoutData(LayoutRegion.
CENTER);
centerData.setCollapsible(false);

BorderLayoutData westData = new BorderLayoutData(LayoutRegion.
WEST, 200, 150, 300);
westData.setCollapsible(true);
westData.setSplit(true);
```

5. Finally, create two new content panels and add them to the view's `west` and `center` panels respectively:

```
ContentPanel mainPanel = new ContentPanel();
ContentPanel navPanel = new ContentPanel();
viewport.add(mainPanel, centerData);
viewport.add(navPanel, westData);
```

6. Run the application and check that the screen now looks like this:

What just happened?

We have now added Component to the Viewport using the BorderLayout. Now the application looks more like a desktop application, enabling users to collapse and expand the left panel.

Loading message

When building any GUI application, it is important to keep the user informed about what is going on. GWT and particularly GXT applications may take several seconds to load all the JavaScript and images on startup. Therefore, it is useful to display a loading message.

When our application first starts, the JavaScript has not yet loaded, so we have to place our loading message in the application's HTML page. GXT will then hide it when the main JavaScript has loaded and the UI has been rendered.

Time for action – adding a loading message

1. Open the application's HTML file, `war\RSSReader.html`, and add the following code to the body of the HTML:

```
<div id="loading">
  <div class="loading-indicator">
  <img src="gxt/images/default/shared/large-loading.gif"
    width="32" height="32"/>RSS Reader<br />
<span id="loading-msg">Loading...</span></div>
</div>
```

   This creates a new `div` with the ID `loading`. This name is important, as it makes hiding the loading message when the application has loaded straightforward. The `div` itself contains an animated gif from GXT's standard resources with a familiar AJAX loading animation together with the loading message itself.

2. We now need to add the styling for the loading indicator, so open up the `war\RSSReader.css` file, remove the previous styling, and add the following:

```
#loading {
  position: absolute;
  left: 45%;
  top: 40%;
  margin-left: -45px;
  padding: 2px;
  z-index: 20001;
  height: auto;
  border: 1px solid #ccc;
```

```
}

#loading a {
  color: #225588;
}

#loading .loading-indicator {
  background: white;
  color: #444;
  font: bold 13px tahoma, arial, helvetica;
  padding: 10px;
  margin: 0;
  height: auto;
}

#loading .loading-indicator img {
  margin-right:8px;
  float:left;
  vertical-align:top;
}

#loading-msg {
  font: normal 10px arial, tahoma, sans-serif;
}
```

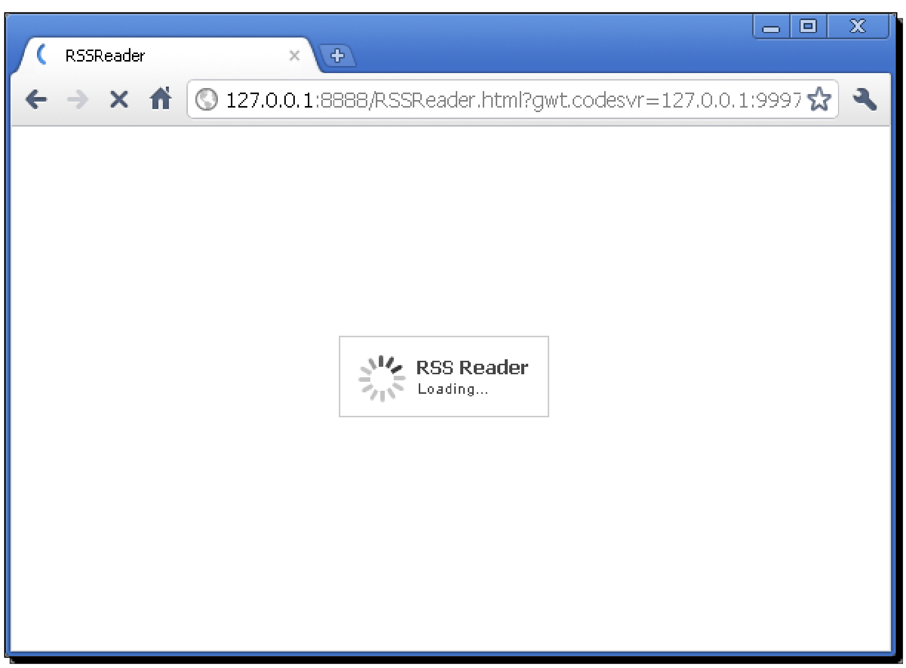
3. At this point, it is also a good idea to revisit the HTML and look at the area where the JavaScript is loaded. At the moment, the script tag that loads the GWT-generated `rssreader/rssreader.nocache.js` is in the head of the document, meaning that its loading is started before the body loads. We want to make sure our loading message is displayed before the JavaScript starts loading so that our user is not looking at an empty page for any noticeable time. So we need to move the `script` tag from the head of the document to the end of the body so that the HTML file looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-
  8">
<link type="text/css" rel="stylesheet" href="RSSReader.css">
<link type="text/css" rel="stylesheet" href="gxt/css/gxt-all.css">
<title>RSS Reader</title>
</head>
<body>
<div id="loading">
```

```
<div class="loading-indicator"><img
  src="gxt/images/default/shared/large-loading.gif" width="32"
  height="32" />RSS Reader<br />
<span id="loading-msg">Loading...</span></div>
</div>
<script type="text/javascript" language="javascript"
  src="rssreader/rssreader.nocache.js"></script>
</body>
</html>
```

4. If we were using this technique in a conventional GWT application, we would now need to add code to our `onModuleLoad` method to hide or remove the loading `div` when the application is loaded. However, as we are using a GXT Viewport panel, this is taken care of for us. If there is a `div` with ID `loading`, it will automatically be hidden. In fact, it will nicely fade away once the viewport is attached. If we had wanted to call our loading `div` something different, we would call the Viewport's `setLoadingPanelId(java.lang.String loadingPanelId)` method where `loadPanelId` is the ID of our loading div.

5. Start the application. It will now have a loading indicator that will disappear when the UI has loaded:



What just happened?

We added a loading message to our application in such a way that it is automatically hidden once the UI is ready.

Custom components

In GXT just as in GWT, it is possible to build on the existing components to make custom components. There are two reasons to make a custom component. The first is to modify the functionality of an existing component. The second is to encapsulate one or more existing components with additional functionality to make a new component. As in GWT, there is the concept of the Composite widget in GXT. This is a component that wraps another in order for you to be able to create a custom component. In GWT, a Composite behaves exactly the same way as the Widget it is wrapping. In GXT, you might encounter some problems if you wrap components in this way.

Take for example the ContentPanel we have added for navigation. It is being added to a layout region of a Viewport that is collapsible. However, if we were to use a Composite to make a custom component based on a ContentPanel, the collapse button would mysteriously disappear.

This is because when you wrap a component in a Composite, its public API methods are hidden. GXT is designed in such a way that it needs access to those public methods in order to query the capabilities of a component. Although a ContentPanel is collapsible, there is no way for GXT to work out that a Composite based on ContentPanel is collapsible. This is because the `isCollapsible` method is hidden and so the layout region would no longer show the collapse icon.

Therefore, to create a custom component in GXT, it is nearly always better to directly extend Component directly or indirectly using one of its subclasses such as BoxComponent, Container, or LayoutContainer. The decision of which component to extend depends on the features your custom component requires.

The onRender method

When extending any component, there is the option of overriding the `onRender` method. This goes back to the idea of lazy rendering. Any code that is in the constructor of a component will get executed as soon as the component is initialized. However, any code in the `onRender` method will only get executed when the component is rendered.

For this reason it is good practice when defining a component to consider if the code needs to execute before the component is rendered. If not, which is often the case, it is better to put the code in the `onRender` method, and the code will only run if and when the component is rendered. Steps like this can improve the efficiency of your GXT applications.

At the moment, in our example application, we have just used standard `ContentPanel` objects for our navigation and main panels. We could keep these as `ContentPanel` objects, but as the application develops, we are going to be adding more and more custom functionality to them. For this reason, we are now going to define them as custom components that extend ContentPanel.

Time for action – creating custom components

1. In your application, create a new class named `RssNavigationPanel` in a new components package under the application's client package. This class should subclass `ContentPanel`.

2. The code for `RssNavigationPanel` should look as follows:

```
public class RssNavigationPanel extends ContentPanel
{
  public RssNavigationPanel()
  {
    setHeading("Navigation");
  }
}
```

   At this point, the only customizing we are doing in this custom widget is giving the panel a heading of `Navigation`.

3. Now we need to do the same for the main panel, this time naming the class `RssMainPanel` and setting the heading to `Main`:

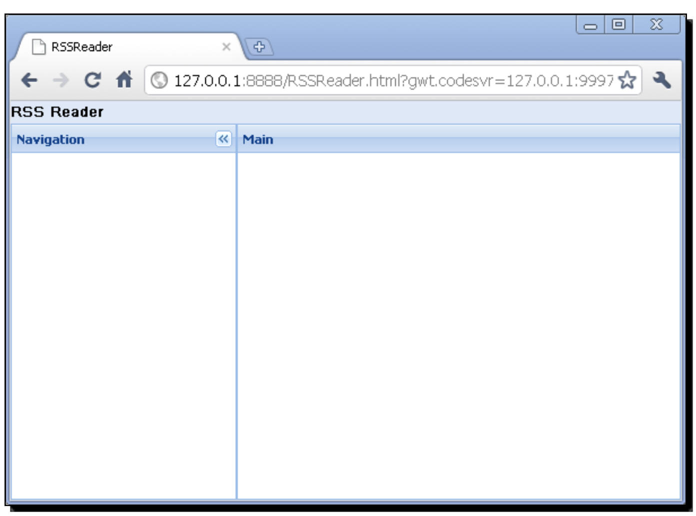
```
public class RssMainPanel extends ContentPanel
{
  public RssMainPanel()
  {
    setHeading("Main");
  }
}
```

4. We now need to replace the two ContentPanels in our main code with our new custom components. In the `onModuleLoad` method of the RSSReader class, modify it as follows:

```
public void onModuleLoad() {
  Viewport viewport = new Viewport();
  final BorderLayout borderLayout = new BorderLayout();
  viewport.setLayout(borderLayout);
```

```
    BorderLayoutData northData = new
      BorderLayoutData(LayoutRegion.NORTH,20);
    northData.setCollapsible(false);
    northData.setSplit(false);

    HTML headerHtml = new HTML();
    headerHtml.setHTML("<h1>RSS Reader</h1>");
    viewport.add(headerHtml, northData);

    BorderLayoutData centerData = new
      BorderLayoutData(LayoutRegion.CENTER);
    centerData.setCollapsible(false);

    BorderLayoutData westData = new
      BorderLayoutData(LayoutRegion.WEST, 200, 150, 300);
    westData.setCollapsible(true);
    westData.setSplit(true);

    RssMainPanel mainPanel = new RssMainPanel();
    RssNavigationPanel navPanel = new RssNavigationPanel();
    viewport.add(mainPanel, centerData);
    viewport.add(navPanel, westData);

    RootPanel.get().add(viewport);
  }
```

5. Now start the application. It should look pretty much the same as before, but now the navigation and main panels will have headings:



What just happened?

The application looks pretty much as it did before except that the navigation and main panels now have headings. However, what we have done is use custom components. By doing this now, we will make code a lot more manageable as the application develops.

First field components

Two of the most basic components that are used for accepting user input as opposed to laying out other components are Buttons and TextFields. These are similar to Buttons and TextBoxes respectively in GWT.

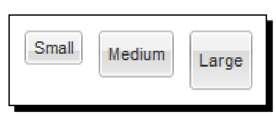
Like most GXT controls that have a counterpart in GWT, the GXT ones are a little richer.

Button

Let us start with buttons. In GXT, buttons have several different attributes, which can be merged to make a large number of combinations:

Size

Buttons come in three sizes—small, medium, or large. They have a property named button scale, which is set using the `setScale` method with the parameters `ButtonScale.SMALL`, `ButtonScale.MEDIUM`, and `ButtonScale.LARGE`, respectively. The text of the Button is set using the `setText()` method.



Icons

Buttons can also have icons as well as just text. This is achieved by using the button's `setIconStyle` method and referencing a CSS style that specifies a background image. So if we wanted to reference an icon named "bell", we would need an entry in our CSS stylesheet like this:

```
.bell {
  background: url(gxt/images/icons/bell.png) no-repeat center left
    !important;
}
```

Then we could call `setIconStyle("bell")` on the buttons and they would look like this:

Alternatively, you can have just icons without text by simply not using the `setText` method.

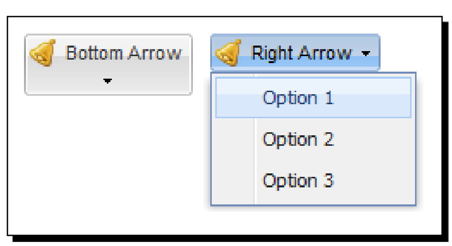
Icon position

You can control where the icon appears on the button relative to any text by using the `setIconAlign` method with the parameters `IconAlign.LEFT`, `IconAlign.RIGHT`, `IconAlign.BOTTOM`, and `IconAlign.TOP`, and get the following results:

Adding a menu

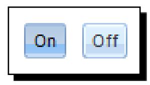
Normally a button performs one action when clicked on. However in GXT, you can add a menu to a button so that it displays a list of options instead. A small arrow is added to a button that has a menu. This can be added either to the bottom or the right of the button text.

The position of the menu arrow can be controlled by the `setArrowAlign` method using the parameters `ButtonArrowAlign.BOTTOM` or `ButtonArrowAlign.RIGHT`:



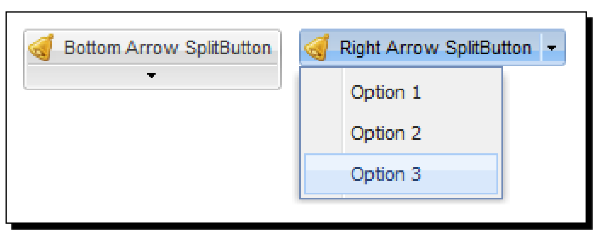
ToggleButton

ToggleButton subclasses button to add toggle (on/off) functionality rather than just executing an action. It maintains a pressed and un-pressed state. You can also group toggle buttons using the `toogleGroup` method so that only one of the group can be pressed at a time as shown below:



SplitButton

SplitButton also subclasses Button and allows you to both click on them and display a menu. SplitButton can be clicked on in a main area to perform an action or the menu arrow can be clicked on to display a menu of further options as with a normal button with a menu button. To use this functionality, you need to create a SplitButton component instead of a standard Button component:



Creating a Link feed button

We now want to add a **Link feed** button to our RssNavigationPanel. The purpose of the button is to display a form that lets the user enter an URL of an existing RSS feed they would like to link to. A user may change their mind and not enter an URL after all. We could provide a cancel button for this, but as we have `ToggleButton` at our disposal, it would make more sense and save limited space to make the **Link feed** button show and hide the URL field.

As our RssNavigationPanel is based on ContentPanel, we inherit a built-in container for any buttons we add to the ContentPanel, making adding a button very simple.

Time for action – adding a button

1. In the constructor of RssNavigationPanel, create a new `ToggleButton`:

   ```
   final ToggleButton btnLinkFeed = new ToggleButton("Link feed");
   ```

2. Add a style to the stylesheet (`war\RSSReader.css`), which includes a suitable icon:

```
.link-feed {
  background: url(gxt/images/icons/feed_link.png) no-repeat
    center left
    !important;
}
```

3. Back in RssNavigationPanel, set the icon for the button:

```
btnLinkFeed.setIconStyle("link-feed");
```

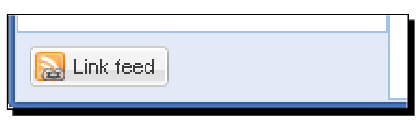
4. Set the horizontal alignment of the panel's default button container to be left-aligned:

```
setButtonAlign(HorizontalAlignment.LEFT);
```

5. Add the newly created ToggleButton to the panel's default button container:

```
addButton(btnLinkFeed);
```

6. Run the application and check that the **Link feed** button appears at the bottom left:



What just happened?

We added a ToggleButton with an icon to the RssNavigationPanel's built-in button container.

Tooltip

The labels on buttons are concise by nature. For users who are familiar with your application or other applications that are similar, the label and icon may be enough for them to understand what the button does. Other users, however, will appreciate a more detailed explanation and that is where tooltips come in.

Tooltips can be added to buttons or other components to give the user further information when they hover their mouse over them.

We are now going to add a tooltip to our **Link feed** button so that a message displays when the user hovers their mouse over the button:

Time for action – adding a tooltip

1. In the constructor of RssNavigationPanel, add the following code to define a new `ToolTipConfig` object. Notice how the tooltip can have a title and text:

```
ToolTipConfig linkFeedToolTipConfig = new ToolTipConfig();
  linkFeedToolTipConfig.setTitle("Link to existing RSS
    feed");
  linkFeedToolTipConfig.setText("Allows you to enter the URL
    of an existing RSS feed you would like to link to");
```

2. Then to associate the tooltip with the button, use the `setToolTip` method:

```
btnLinkFeed.setToolTip(linkFeedToolTipConfig);
```

3. Run the application and hover your mouse over the **Link feed** button to display the tooltip like this:

What just happened?

We created a new tooltip and associated it with the **Link feed** button so that when a user hovers over the button, the tooltip is displayed.

Popup

Popup subclasses LayoutContainer, adding the ability for the component to be displayed over other components. We would like to take advantage of our example application to pop up a small form for entering an URL when the user clicks on the **Link feed** button. We will point out some of the features of Popup as we go along.

Time for action – creating a popup

1. Create a new class in the `client.components` package named LinkFeedPopup that extends Popup.

2. Add a constructor as follows:

```
public LinkFeedPopup() {
  setSize(300, 55);
```

```
    setBorders(true);
    setShadow(true);
    setAutoHide(false);
  }
```

3. This will set the popup to be `300`px wide and `55`px high with borders and a shadow. If auto hide was not set to `false`, the popup would disappear if we clicked outside of that. As our popup's visibility is controlled by a toggle button, we need to disable this, or the button and the popup will become out of sync.

What just happened?

We defined a new component based on a popup, which we will use for displaying a TextField for the user to type in an URL.

SelectionListener

In order for our **Link feed** button to do anything, it needs a Listener. A Listener registers with a component and is informed when an event occurs. It can then execute code as a result to respond to the event. Listeners in GXT are similar to Listeners in earlier versions of GWT and Handlers in current versions. In this case, our listener needs to be registered for the **Link feed** button to listen for selection events being selected, so we use a `SelectionListener`. We will now add a `SelectionListener` to our **Link feed** button.

Time for action – adding a SelectionListener

1. In the RssNavigationPanel class, create a new instance of our `LinkFeedPopup` component:

   ```
   final LinkFeedPopup linkFeedPopup = new LinkFeedPopup();
   ```

2. We always want the popup to stay within the Viewport. That is, we always want all of it to be visible. To do this we use the `setContrainViewport` method:

   ```
   linkFeedPopup.setConstrainViewport(true);
   ```

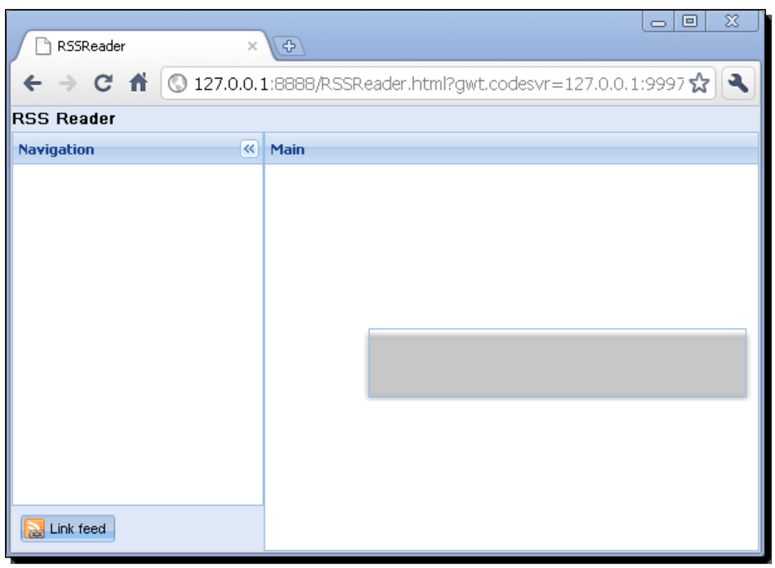
3. Now add a selection listener to the `btnLinkFeed` button:

   ```
   btnLinkFeed.addSelectionListener(new SelectionListener<
   ButtonEvent>() {
     @Override
     public void componentSelected(ButtonEvent ce) {
       if (btnLinkFeed.isPressed()) {
         linkFeedPopup.show();
   ```

```
            } else {
              linkFeedPopup.hide();
            }
          }
        });
```

As the listener is for a ToggleButton, we check the button's state using `isPressed`. If the button is pressed, we need to show our popup by calling the `show` method, otherwise we should hide it using the `hide` method.

4. Run the application and check that the **Link feed** button now displays the Link Feed Popup when toggled on and makes it disappear again when toggled off:



What just happened?

We have now added a SelectionListener to our **Link feed** ToggleButton to enable us to hide and show a popup.

Field

Field subclasses BoxComponent and provides the base class for all form fields. There is a GXT field corresponding to all the standard HTML form controls such as textboxes, checkboxes and list boxes. The Field base class provides default event handling, value handling, as well as some other functionality. Fields are an important feature in GXT and again offer significantly more functionality than their GWT equivalents. We will come back to fields in subsequent chapters, but for the time being we are just going to look at the TextField.

TextField

TextField subclasses Field and is the equivalent of GWT's TextBox widget. There are a few differences, one being that when defining a text field, you define the data type that the field will store using generics. For example, a `TextField` that stores strings will be defined as follows:

```
TextField<String> text = new TextField<String>();
```

GXT text fields also have a number of built-in functions that allow for setting validation criteria such as making the fields required, setting a minimum and maximum length, and validating against standard regular expressions.

In our example application, we need to use a TextField to allow the user to enter the URL of the news feed they want to add in our link feed popup.

Time for action – adding components to the Link feed popup

1. In the LinkFeedPopup class, create a text field for the user to enter the URL into:

```
private final TextField<String> tfUrl = new TextField<String>();
```

2. Override the `onRender` class of LinkFeedPopup:

```
@Override
protected void onRender(Element parent, int pos) {
  super.onRender(parent, pos);
}
```

3. In the method, add a `Text` component to tell the user to enter an URL in the text field:

```
final Text txtExplaination = new Text("Enter a feed url");
```

4. Create a `Button` to submit the value of the URL field and a `SelectionListener` to respond to the user clicking on the button:

```
final Button btnAdd = new Button("add");
  btnAdd.addSelectionListener(new
    SelectionListener<ButtonEvent>() {
      public void componentSelected(ButtonEvent ce) {
        addFeed(tfUrl.getValue());
      }
  });
```

5. You may have noticed that `SelectionListener` is calling a method called `addFeed` to deal with the value the user enters. We are not going to process the value yet, but for the time being we should just create a placeholder method so that it does something. In this case, displaying an alert box:

```
public void addFeed(String url) {
  Window.alert("We would now attempt to add " + url + " at
    this point");
  }
```

6. We now need to create a new `BorderLayout` like the one we used for the Viewport earlier in this chapter, but this time we are going to use it for laying out the popup, so we add it using `setLayout`:

```
final BorderLayout layout = new BorderLayout();
setLayout(layout);
```

7. With the layout set, we can use layout data to position the components. First the text:

```
final BorderLayoutData northData = new
BorderLayoutData(LayoutRegion.NORTH, 20);
northData.setMargins(new Margins(2));
add(txtExplaination, northData);
```

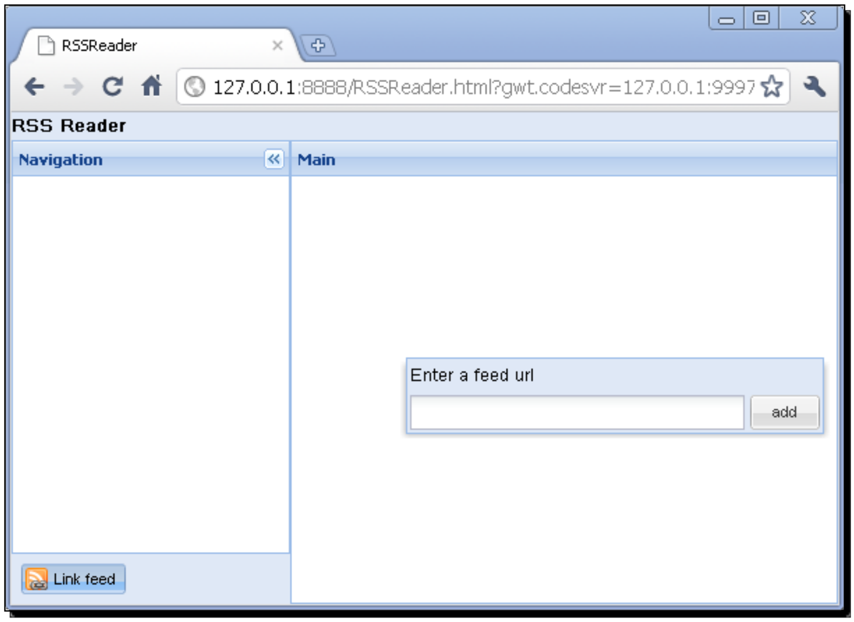
8. Then the textbox:

```
final BorderLayoutData centerData = new
BorderLayoutData(LayoutRegion.CENTER);
centerData.setMargins(new Margins(2));
add(tfUrl, centerData);
```

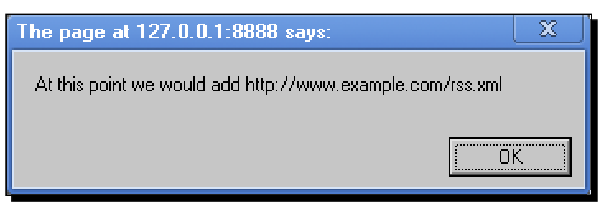
9. And finally, the button:

```
final BorderLayoutData eastData = new
BorderLayoutData(LayoutRegion.EAST, 50);
eastData.setMargins(new Margins(2));
add(btnAdd, eastData);
```

10. Now start the application, click on the **Link feed** button and check that the popup now contains fields as shown:



11. Now complete the URL field, click on the **add** button and check that a message like this is shown:



What just happened?

We added fields to our Link feed popup and created a SelectionListener to respond to the **add** button being pressed.

Pop quiz – matching the component with the description

In the chapter so far, we have covered a lot of components. Try to match the following components with the descriptions:

1. Tooltip
2. Popup
3. Viewport
4. ContentPanel
5. Button
6. Composite
7. TextField
8. BoxComponent
9. Component
10. LayoutContainer

a. has versions with icons and menus
b. fills the browser window and monitors for resizing
c. can appear over other components
d. subclasses component, adding sizing and positioning methods
e. accepts text input
f. base class for all GXT components
g. provides additional information when hovering over another element
h. has collapse and expand abilities and built-in toolbars
i. wraps another component hiding its public methods
j. component that can contain other components and control lays them out using a layout

Popup positioning and alignment

While our popup displays properly, it appears in the middle of the screen away from the button that made it appear. Although we are using a ToggleButton, it is not obvious that the button would also make it disappear again. It would be a lot more user friendly if the popup appeared directly above the **Link feed** button.

At the moment we are simply using the `show()` method to display the popup. What we want to do is display our popup in relation to the **Link feed** button. To do this, we pass the `show()` method the following information:

1. The underlying element of the button using the `getElement` method.
2. A string representing how the element should align with the target element (in this case the button's element). The string is made up of the anchor point of the element to align followed by a dash, and then the anchor point of the element we want to align the element to. If the string ends with a question mark, the element will attempt to align as defined, but it means that it will reposition the popup so that it remains in the viewport.
3. The different codes for alignment points are as follows:

Code	Meaning
tl	The top left corner (default)
t	The center of the top edge
tr	The top right corner
l	The center of the left edge
c	In the center of the element
r	The center of the right edge
bl	The bottom left corner
b	The center of the bottom edge
br	The bottom right corner

An alignment string containing "tl-bl?", the default, will align the top left of the element with the bottom left of the target element unless that would cause it to be outside the viewport. This means the element would appear directly below the target.

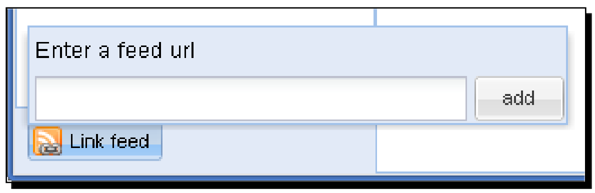
Time for action – positioning the popup

1. In the RssNavigationPanel, modify the action performed by the `SelectionListener` so that the bottom-left corner of the `linkFeedPopup` will align to the top-left corner of the **Link feed** Button:

```
btnLinkFeed.addSelectionListener(new SelectionListener<ButtonEve
nt>() {
  @Override
  public void componentSelected(ButtonEvent ce) {
    if (btnLinkFeed.isPressed()) {
      linkFeedPopup.show(btnLinkFeed.getElement(), "bl-
        tl?");
```

```
            } else {
              linkFeedPopup.hide();
            }
          }
        });
```

2. Start the application and check that the popup now appears directly above the **Link feed** Button:



What just happened?

We specified that the bottom-left of our popup should be aligned with the top-left of our Link feed button, unless it would mean that it will appear outside of the viewport.

Have a go hero – adding a KeyListener

At the moment the user has to type the URL of the feed they want to add into the TextField, and then press the **add** button. It would be quicker if pressing the *Enter* key could perform the same function.

TextField controls can take a `KeyListener` that responds to key presses. The key code for the *Enter* key can be obtained by using the GWT static method `KeyCodes.KEY_ENTER`. Try to add a `KeyListener` to the `tfUrl` field in the LinkFeedPopup class that responds to the *Enter* key being pressed in the same way that the `SelectionListener` responds to the **add** button being pressed.

Solution:

```
tfUrl.addKeyListener(new KeyListener() {
  public void componentKeyDown(ComponentEvent event) {
    if (event.getKeyCode() == KeyCodes.KEY_ENTER)
    {
      addFeed(tfUrl.getValue());
    }
  }
});
```

Summary

In this chapter, we have rapidly run through most of the basic interface building blocks of GXT. We have used them to start building a sample application. This is starting to look and feel more like a desktop application than a traditional web application.

In the next chapter, we will build upon this by introducing some more components.

3
Forms and Windows

In this chapter, we explore GXT's form features. We look at the form components that GXT provides and learn how to put them to use. We also introduce the GXT Registry and see how it can be used across the application.

Specifically in this chapter, we will learn about the following:

- The full range of fields available in GXT
- FormPanel
- FormLayout
- Window
- FitLayout
- FieldMessages
- Form submission
- Working with GWT RPC
- Using the registry

Change of requirements

So far, as our example application, we have been building an RSS reader. However, as it often happens, our customer has changed her mind and added to the requirements.

She now requires that the application should not only consume RSS feeds from the Internet, but also be able to create them.

This means that we need to create forms to enter data into our application. Fortunately, GXT has comprehensive form support.

The RSS 2.0 specification

Our RSS reader should consume RSS feeds that conform to the RSS (Really Simple Syndication) 2.0 specifications. An RSS feed is an XML document containing specific content. Now we also have to be able to support being able to create documents in this format in our example application.

RSS 2.0 is quite a simple specification. It can be found at `http://cyber.law.harvard.edu/rss/rss.html` and an example file can be found at `http://cyber.law.harvard.edu/rss/examples/rss2sample.xml`.

Put simply, an RSS file contains a channel element that first provides a name and description of a feed and then a number of item elements containing individual news items.

Some of the elements in the file are compulsory and some are optional, but for the sake of this example, we shall make use of the following channel elements:

- title
- link
- description

For each item, we will also create a similar set of elements:

- title
- link
- description

So we need to make two forms, one that will take channel information and another that will take item information. Let's look at what GXT provides to assist us with this.

FormPanel

`FormPanel` subclasses `ContentPanel` and provides features for managing form components. By default, it uses a layout called `FormLayout`. The only types of component that we can add to a `FormLayout` are `Field` components such as `TextField` and `LabelField`. If we try to add any other component, it will be ignored and not rendered.

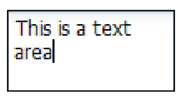
The main benefit `FormPanel` gives is the ability to act on all the fields contained within it. This includes features such as marking all fields as read-only, checking that all fields are valid, changing how labels are displayed, and ultimately submitting the form using HTTP post or GWT RPC.

Fields

In the last chapter, we introduced `TextField`. This is just one of the many fields available in GXT.

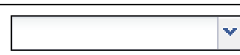
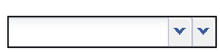
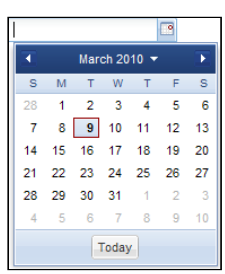
TextFields

The following components subclass `TextField` to provide more specialist features:

Component	Screenshot	Description
`NumberField`		A `TextField` that only allows numbers to be entered. It also provides additional methods for validating the numbers input, such as max and min values.
`TextArea`		A multiline text field similar to an HTML text area field.
`FileUploadField`		An HTML-style file upload field with a browse button to allow the user to locate a file.

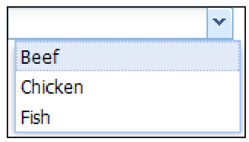
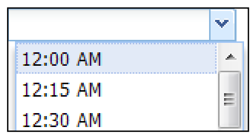
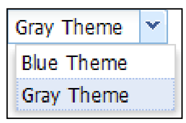
TriggerField components

The fourth field that subclasses `TextField` is called `TriggerField`. This looks like a `TextField`, but also adds a trigger button to the right of it. This too has several subclasses. These are the important ones:

Component	Screenshot	Description
`TriggerField`		Basic trigger field.
`TwinTriggerField`		A trigger field with two trigger buttons.
`DateField`		A trigger field that enables the user to enter a date by clicking on the trigger button and picking a date from a date picker.
`ComboBox`		A `combobox` that uses a `ModelData` object to provide the options. We cover `ModelData` in the next chapter.

ComboBox component

There are also a number of subclasses of `ComboBox`, which makes it more convenient to use:

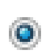
Component	Screenshot	Description
`SimpleComboBox`		`SimpleComboBox` lets us enter a hardcoded list of options for the `ComboBox` and handles the `ModelData` object automatically.
`TimeField`		`TimeField` populates the `ComboBox` with a list of times. The interval between the times can be varied. For example, every 15 minutes, every 30 minutes, and so on.
`ThemeSelector`		`ThemeSelector` populates the `ComboBox` with all registered GXT themes. Selecting one causes the application to reload with the new theme applied.

ListField component

`ListField` has similarities to `ComboBox`, as it uses `ModelData` to provide a list of options. However, unlike a `ComboBox`, a user can select multiple values rather than the single selection that a `ComboBox` allows.

CheckBox components

`CheckBox` fields are tickboxes that can be either off or on:

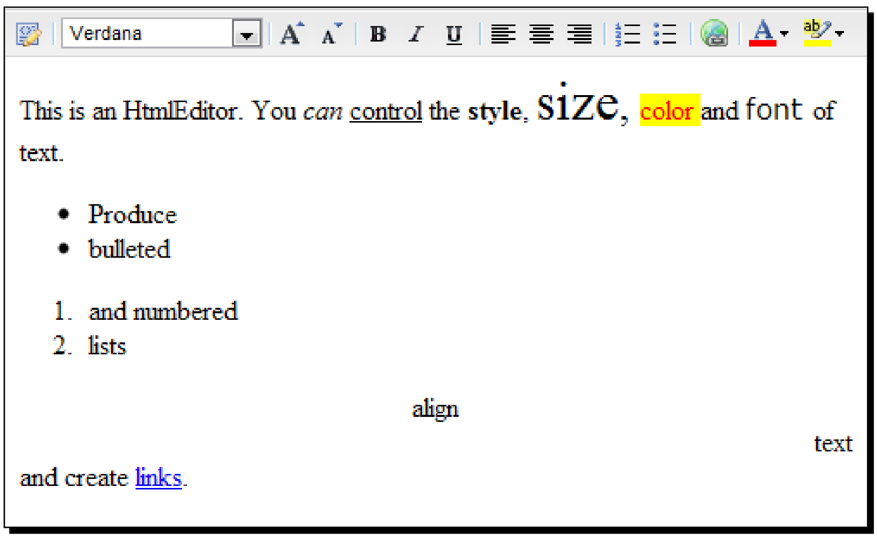
Component	Screenshot	Description
`CheckBox`		A single checkbox.
`Radio`		A subclass of `CheckBox`, but with a round radio button.

`CheckBox` and `Radio` fields can be used alone or in a group using `CheckBoxGroup` or `RadioGroup` fields respectively. Both group fields subclass `MultiField`, a field that displays multiple fields in a single row or column.

`CheckBoxGroup` allows for all the `CheckBoxes` within it to be aligned together either horizontally or vertically. A `RadioGroup` also makes sure that only one of the radio fields within it are selected at a time.

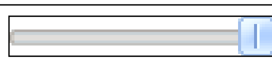
HtmlEditor component

The `HtmlEditor` field allows the user to enter rich text, which is stored as HTML. This provides a very user friendly way of entering text with a powerful range of formatting options. Users can also add their own HTML links, although this and other functionality can be restricted:



Other field components

There are some more fields that don't really fall into any other categories:

Component	Screenshot	Description
SliderField		`SliderField` is a wrapper for a slider component that allows it to be used in a form.
LabelField		`LabelField` simply displays static text.
AdapterField		An `AdaptorField` lets us wrap a custom component as a field for use in a form.
FieldSet		A `FieldSet` is a container for another field, which allows for a border, title, and expand/collapse functionality.
HiddenField		`HiddenField` is used for submitting a hidden value with the form. It is invisible.

Pop quiz – match the form components with their definitions

Match the following definitions with the form component it best matches:

1. Lets you wrap a custom component to use on a form.
2. Allows for the entry of rich text.
3. Allows the user to pick a date from a calendar.
4. Only allows numbers to be entered.
5. Uses ModelData objects to build a list from which only one item can be selected.
6. Allows multiline plain text to be entered.
7. Automatically changes the applications theme when an item is selected.
8. Allows for Radio components to be aligned together.
9. Used for submitting hidden values.
10. Displays static text.

a. `ThemeSelector`
b. `RadioGroup`
c. `HtmlEditor`
d. `LabelField`
e. `NumberField`
f. `TextArea`
g. `ComboBox`
h. `Adapter Field`
i. `DateField`
j. `HiddenField`

Expanding the example application

We are now going to use GXT form components to create a form for creating a new feed in our example application. First of all though, we need to create a new button that the user can click on to cause the form to be displayed.

Creating a Create feed button

We need to add a new button to the `RssNavigationPanel` that allows the user to create a new feed. This will be very similar to what we did in the last chapter to show the Add Feed pop up, but this time we will use a standard `Button` and not a `ToggleButton`.

Time for action – adding a Create feed button

1. In the `RSSReader.css` stylesheet, add a new element for a create button icon in the same way as we did for link-feed in the last chapter:

```
.create-feed {
  background: url(gxt/images/icons/feed_create.png) no-repeat
    center left
    !important;
}
```

2. In the constructor of the `RssNavigationPanel` class, define a new `Button` labeled **Create feed** in the same way as we did with the **Link feed** `Button`, remembering also to define a `ToolTipConfig` and an icon:

```
final Button btnCreateFeed = new Button("Create feed");
btnCreateFeed.setIconStyle("create-feed");

ToolTipConfig createNewToolTipConfig = new ToolTipConfig();
createNewToolTipConfig.setTitle("Create a new RSS feed");
createNewToolTipConfig.setText("Creates a new RSS feed");
btnCreateFeed.setToolTip(createNewToolTipConfig);

addButton(btnCreateFeed);
```

3. Start the application and check that there are now two buttons—**Create feed** and **Link Feed**:

What just happened?

We have added a **Create feed** button. We will now go on to create a Window that will display on clicking the button, but first we need to define an object to hold feed data.

Creating a Feed class

The first thing we will do is to build a Java object to represent feed data. This is a simple POJO (Plain Old Java Object).

As we plan to send this object over Google RPC, we need to make sure that it implements the `Serializable` interface and has a null argument constructor. As this class will be compiled in JavaScript and be used by the server, it is useful to put it in the `shared` package rather than `client` or `server`. This way, the class is shared between the client and server.

It is also important to tell GWT to include this new common package as JavaScript when compiling classes, and this is done in the `RSSReader.gwt.xml` module file.

Time for action – creating a feed data object

1. In your example application, create a new package named `shared.model` at the same level as your existing server and client packages.
2. Modify the `RSSReader.gwt.xml` file so that it includes the source code in the `shared` package by adding an entry for the `shared` package:

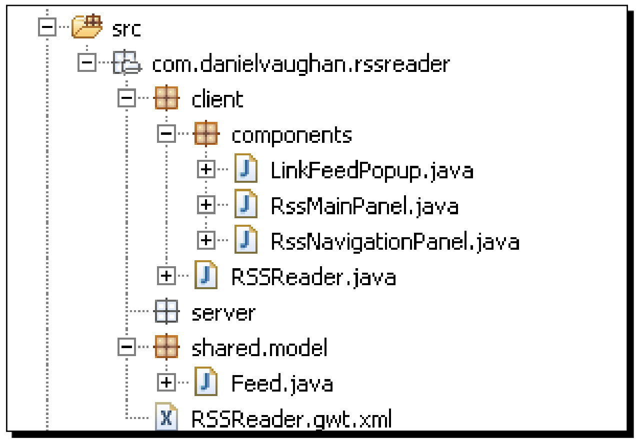
```
<!-- Specify the paths for translatable code -->
<source path='client'/>
<source path='shared'/>
```

3. Create a new class named `Feed` in the `shared.model` package and implement it, as shown here:

```
package com.danielvaughan.rssreader.shared.model;

import java.io.Serializable;

@SuppressWarnings("serial")
public class Feed implements Serializable {

  private String description;
  private String link;
  private String title;
  private String uuid;
```

```
  public Feed()
  {

  }

  public Feed(String uuid)
  {
    this.uuid = uuid;
  }

  public String getDescription() {
    return description;
  }

  public String getLink() {
    return link;
  }

  public String getTitle() {
    return title;
  }

  public String getUuid() {
    return uuid;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public void setLink(String link) {
    this.link = link;
  }

  public void setTitle(String title) {
    this.title = title;
  }
}
```

4. The structure of our project should now look like this:



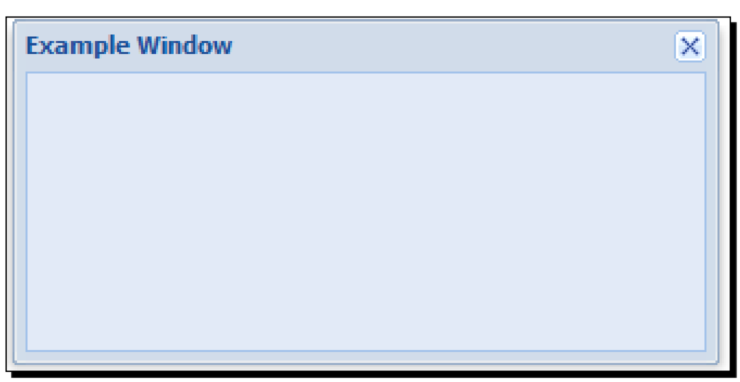
What just happened?

We created a feed data class to hold feed data in a shared package. We then included the package as a source package in the `RSSReader.gwt.xml` module file.

Window

`Window` is a specialized subclass of `ContentPanel` intended to be used as a window within an application. In some ways, it is like the `Popup` covered in the last chapter, in that it can be displayed in front of other components. However, it also can be dragged around the screen, closed by clicking on a close button, and optionally resized by the user.

As with a `Popup`, a `Window` does not need to be held in another container. It just needs to be created and the `show()` method called. An empty window looks like this:



FitLayout

When we add our form to a `Window`, we would like it to fill the window completely. To achieve this, there is a useful layout called `FitLayout`. This can be used with any container that contains a single item and will automatically expand the item so that it fills the container. We will use the `FitLayout` to let us add our `FormPanel` to a `Window` and get it to fill that `Window`.

Creating the FeedWindow component

We are now going to create a new `Window` that we will use as a container for the `FeedForm` that we will create next. It will be displayed in response to the user clicking on the **Create feed** button we created earlier.

Time for action – creating a Window

1. Create a new class called `FeedWindow`, which extends `Window` in a new package `client.windows`:

```
public class FeedWindow extends Window {}
```

2. Create a constructor for the class that takes a `Feed` object as an argument. In it, set the heading of the window to `"Feed"`:

```
public FeedWindow(final Feed feed) {
  setHeading("Feed");
}
```

3. Now set the width and height of the window:

```
public FeedWindow(final Feed feed) {
  setHeading("Feed");
  setWidth(350);
  setHeight(200);
}
```

4. In this case, we don't want the window to be resizable, so set resizable to `false`:

```
public FeedWindow(final Feed feed) {
  setHeading("Feed");
  setWidth(350);
  setHeight(200);
  setResizable(false);
}
```

5. To make sure that the `FormPanel` we are going to add fills to the `Window`, create a new `FitLayout`, and use it to set the layout of the `Window`:

```
public FeedWindow(final Feed feed) {
  setHeading("Feed");
  setWidth(350);
  setHeight(200);
  setResizable(false);
  setLayout(new FitLayout());
}
```

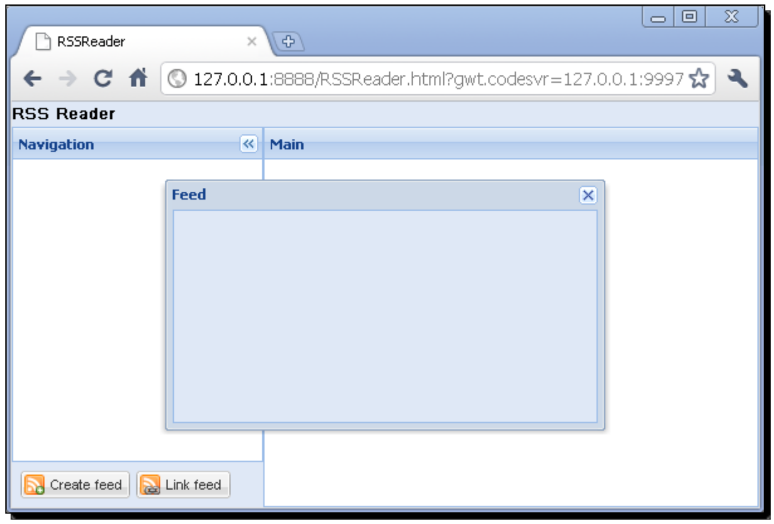
6. Now go back to the `RssNavigationPanel` and define a new method named `createNewFeedWindow`:

```
private void createNewFeedWindow()
  {
    final Window newFeedWindow = new FeedWindow(new Feed());
    newFeedWindow.show();
  }
```

7. In the constructor of `RssNavigationPanel`, add a `SelectionListener` to the **Create feed** `Button`. Implement the `SelectionListener` to call the `newFeedWindow` method, defined in the last step:

```
btnCreateFeed  .addSelectionListener(new
  SelectionListener<ButtonEvent>() {
  @Override
  public void componentSelected(ButtonEvent ce) {
    createNewFeedWindow();
  }
});
```

8. Finally, start the application and click on the **Create feed** button. Check that a new empty window is displayed like this:



What just happened?

We created a new `Window` called `FeedWindow` and created a `SelectionListener` for our **Create feed** button that causes the `FeedWindow` to be displayed. We now need to create the `FeedForm` to put in it.

Creating FeedForm

We now need to create a `FormPanel` to enable the user to enter information required to define a new feed. For now, we will just set up the compulsory fields—title, a text field, description, a multi-line text field, and link a text field that must be a URL.

Time for action – creating a feed form

1. Create a new class called `FeedForm`, which extends `FormPanel`. Place this in a package named `client.forms`.

2. As our form is displayed in a window, we will not make use of the header of the `FormPanel`, so add a constructor to `FeedForm` as follows:

```
public FeedForm()
{
  setHeaderVisible(false);
}
```

3. Now we need to define our fields. Define a `TextField` for the title and the link and a `TextArea` for the description:

```
private final TextField<String> tfTitle = new TextField<String>();
private final TextArea taDescription = new TextArea();
private final TextField<String> tfLink = new TextField<String>();
```

4. Override the `onRender` method, and in it, set the labels of the fields we just defined:

```
@Override
protected void onRender(Element parent, int pos) {
  super.onRender(parent, pos);

  tfTitle.setFieldLabel("Title");
  taDescription.setFieldLabel("Description");
  tfLink.setFieldLabel("Link");
}
```

5. Now add the three fields to the underlying `FormPanel`:

```
@Override
protected void onRender(Element parent, int pos) {
  super.onRender(parent, pos);

  tfTitle.setFieldLabel("Title");
  taDescription.setFieldLabel("Description");
  tfLink.setFieldLabel("Link");
  add(tfTitle);
  add(taDescription);
  add(tfLink);
}
```

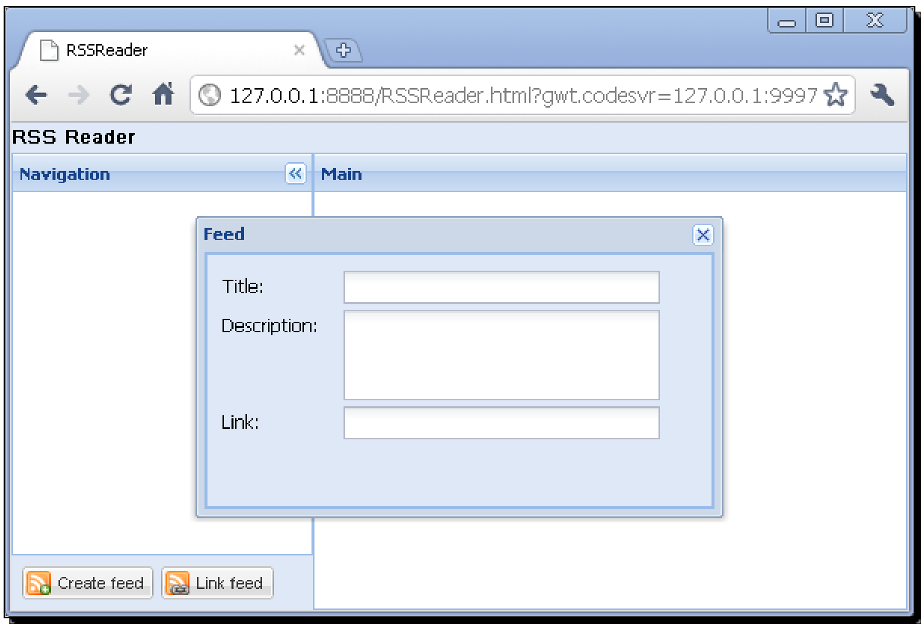
6. Go back to the `FeedWindow` class and define a field to hold a new instance of the `FeedForm`:

```
private final FeedForm feedForm = new FeedForm();
```

7. Override the `onRender` method to add the `FeedForm` to the underlying `Window`:

```
@Override
protected void onRender(Element parent, int pos) {
  super.onRender(parent, pos);
  add(feedForm);
}
```

8. Finally, run the application and check that the new `FeedForm` is now displayed in the `FeedWindow` when the **Create feed** button is pressed, as shown in the next screenshot:



What just happened?

We created a new FormPanel called `FeedForm` and added it to our `FeedWindow`. Now when the **Create feed** button is pressed, the `FeedForm` will be displayed in a `FeedWindow`.

Validating fields

GXT fields provide built-in support for field validation. We can check if any field contains valid data using the `isValid()` method. If fields are contained within a `FormPanel`, we can check that all child fields within it are valid by calling `isValid()` on the `FormPanel`.

By default, fields validate when the user exits them (on blur). However, we can get the field to validate as the user enters a value (after each key press) by calling `setAutoValidate(true)`.

The criteria that we can define for field validation varies between fields, as some may not be suitable for the data type. The tables shown next describe the types of validation available:

Text validation

Validation	Set Using	Description
Allow blank	`setAllowBlank`	If field with length of 0 is valid. Defaults to true.
Minimum field length	`setMaxLength`	The minimum length for a field to be valid. Defaults to 0.
Maximum field length	`setMinLength`	The maximum length for a field to be valid.
Regular expression	`setRegex`	A regular expression that the content of the field must match.

Numerical validation

Validation	Set Using	Description
Allow decimals	`setAllowDecimals`	If decimals are allowed. Defaults to true.
Allow negative	`setAllowNegative`	If negative values are allowed.
Minimum value	`setMinValue`	The minimum numerical value
Maximum value	`setMaxValue`	The maximum value

Custom validator

You can also set a custom validator using the `setValidator` method. A validator is a custom class that implements the Validator interface to provide a `validate` method, which takes a field and a string value. It should return null if validation passes, or return an error message if validation fails.

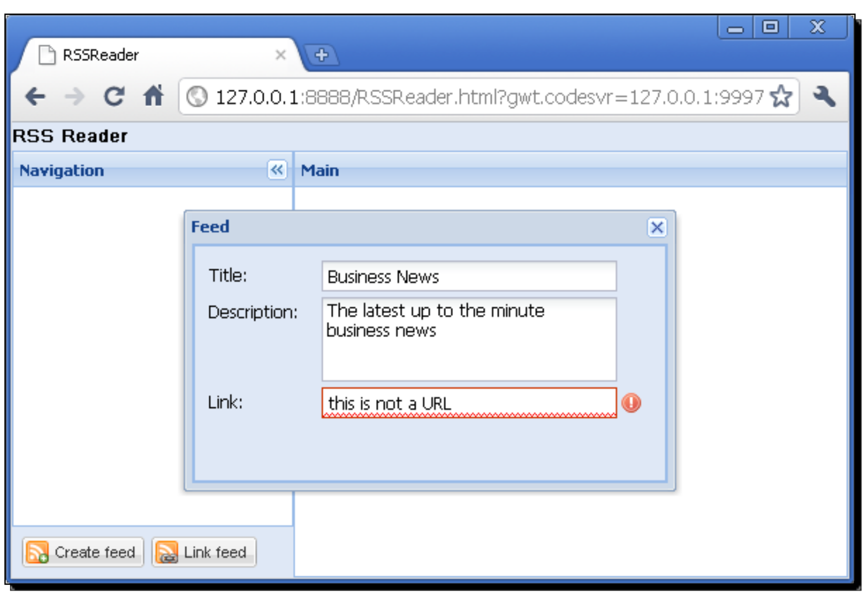
Now let's add field validation to our `FeedForm`. As the RSS specification requires a name, a description, and a link, we are going to disallow blank values. We are also going to add a regular expression to check that the link field is in the correct format.

Time for action – adding field validation

1. In the `onRender` method of the `FeedForm` class, add the following code to allow for validation of the fields. In this case, we are requiring a value in all fields and making sure that the link field contains a valid URL:

```
@Override
protected void onRender(Element parent, int pos) {
  super.onRender(parent, pos);

  tfTitle.setFieldLabel("Title");
  tfTitle.setAllowBlank(false);

  taDescription.setFieldLabel("Description");
  taDescription.setAllowBlank(false);

  tfLink.setFieldLabel("Link");
  tfLink.setAllowBlank(false);
  tfLink.setRegex("^http\\://[a-zA-Z0-9\\-\\.]+\\.[a-zA-
    Z]{2,3}(/\\S*)?$");

}
```

2. Now start your application and click on the **Create feed** button. Check that if you do not enter anything in a field or enter a non-URL in the link field, it is marked as invalid. The screenshot shows how the link field is marked as invalid:



What just happened?

We added validation to our form making all the fields required and making sure that the link field accepts only a valid URL.

Using FieldMessages

As well as specifying how fields should be validated, you can also specify the messages that are displayed when validations fail using `FieldMessages`.

`FieldMessages` are implemented as inner classes of the fields they apply to. For example, the `FieldMessages` implementation for `TextField` is `TextField<D>.TextFieldMessages`. Although you can create a new instance of the appropriate `FieldMessages` class, set the messages and then use the `setMessages` method of the field to attach the `FieldMessages` to the field. This is convoluted. The best way for setting `FieldMessages` is to use the `getMessages` method of the field and then set the appropriate message.

Time for action – adding FieldMessages to the fields

1. In the `onRender` method of the `FeedForm` class, retrieve the title field's `Fieldmessages` and set the text to display if the field is left blank:

```
tfTitle.setFieldLabel("Title");
tfTitle.setAllowBlank(false);
tfTitle.getMessages().setBlankText("Title is required");
```

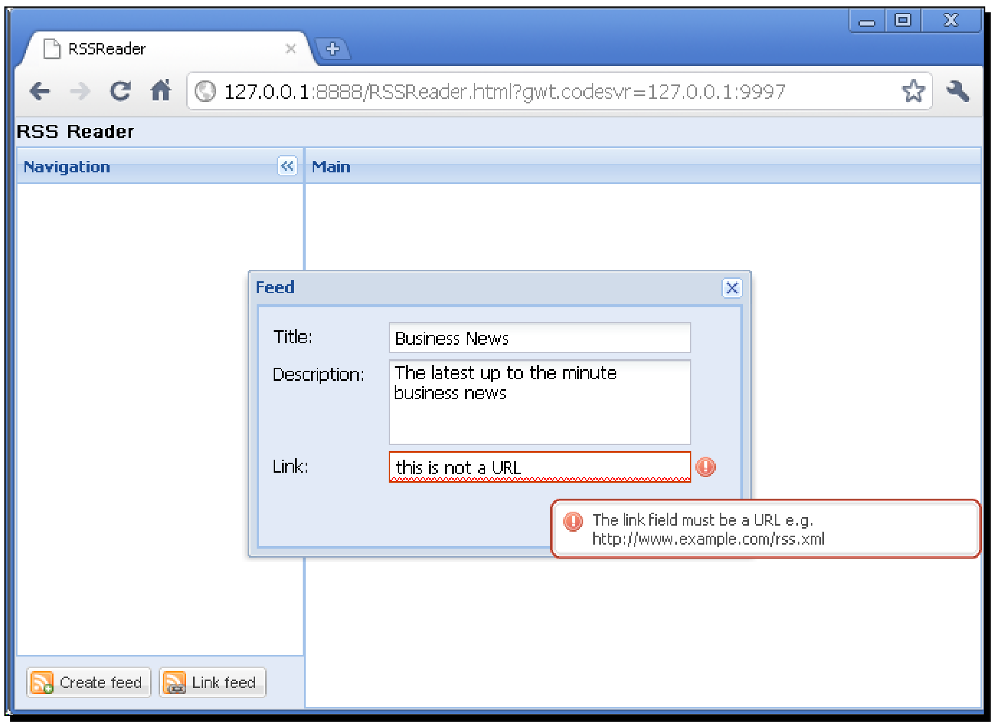
2. Do the same with the description field:

```
taDescription.setFieldLabel("Description");
taDescription.setAllowBlank(false);
taDescription.getMessages().setBlankText("Description is
  required");
```

3. Repeat again for the link field, but also add a message when the URL does not match the regular expression:

```
tfLink.setFieldLabel("Link");
tfLink.setAllowBlank(false);
tfLink.setRegex("^http\\://[a-zA-Z0-9\\-\\.]+\\.[a-zA-
    Z]{2,3}(/\\S*)?$");
tfLink.getMessages().setBlankText("Link is required");
tfLink.getMessages().setRegexText("The link field must be a URL
  e.g. http://www.example.com/rss.xml");
```

4. Now start the application again, and this time you will see that if you hover over the invalid icon, you will see the following message:



What just happened?

We added field messages to our fields so that the user is given feedback when a field fails validation.

Submitting a form using HTTP

There are two ways of submitting data collected on a form. The first is the traditional way, namely, by submitting the form data to the server using an HTTP POST method.

This is straightforward:

- Use the `setAction` method of the `FormPanel` to define the URL to submit the form to
- Use the `isValid` method of the `FormPanel` to check that all the fields are valid
- If the form is valid, use the `submit` method of the `FormPanel` to submit the form

The example code where the submission is triggered by a button would look like this:

```
setAction("http://www.example.com/submit.php");

final Button btnSave = new Button("Save");
btnSave.setIconStyle("save");
btnSave.addSelectionListener(new SelectionListener<ButtonEvent>()
{
  public void componentSelected(ButtonEvent ce) {
    if (isValid())
    {
      submit();
    }            }
});
addButton(btnSave);
```

Alternative to submitting a form using HTTP

With GWT and GXT, unlike traditional web applications, we have the option of storing and manipulating data as Java objects on the client. We can continue to work with these objects in the frontend or submit it to the backend using GWT RPC or other methods such as JSON. In the example application, we shall be using the GWT RPC and so we will need to build a GWT RPC service.

Creating a Feed service

In order to be able to retrieve a `Feed` object, it will need to have a unique ID. In Java, there is a built-in UUID generator, which is ideal for creating such an ID. However, this is part of the JDK that is not available in GWT, and so we cannot generate a UUID in the client.

We can, however, generate a UUID on the server and make it available through an RPC call. In fact, as our Feed objects are available to both the client and the server, we can generate `Feed` objects on the server with their UUID set and return these to the client for use.

This means that we need to create a GWT RPC service to handle `Feed` objects.

Time for action – creating service for feed objects

1. Create a client interface named `FeedService` that extends the GWT `RemoteService` in a new package called `client.services`. At the moment, the interface just needs to specify one method; `createNewFeed`:

```
@RemoteServiceRelativePath("feed-service")
public interface FeedService extends RemoteService {
  Feed createNewFeed();
}
```

2. Now we need to create the matching asynchronous interface. Name this `FeedServiceAsync` and again put it in the `client.services` package. Add the asynchronous version of the `createNewFeed` method:

```
public interface FeedServiceAsync {
  void createNewFeed(AsyncCallback<Feed> callback);
}
```

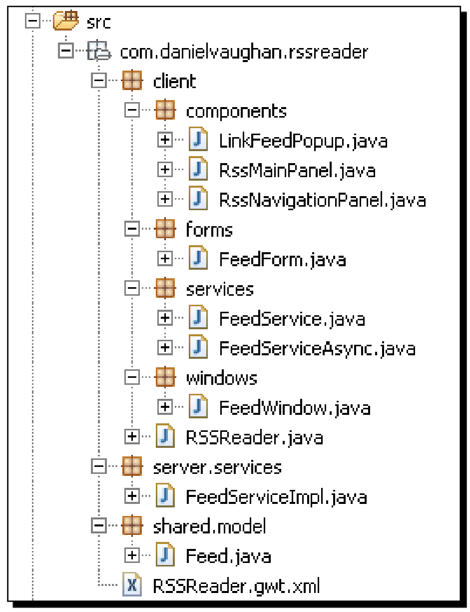
3. Next, create an implementation of the service on the server side. Create a new package called `server.services`, and in it, create a class called `FeedServiceImpl` that implements `FeedService` and extends `RemoteServiceServlet`. Implement the `createNewFeed` method so that it generates a new UUID, creates a new `Feed` object with that UUID, and returns it:

```
public class FeedServiceImpl extends RemoteServiceServlet
  implements FeedService {

  @Override
  public Feed createNewFeed() {
    UUID uuid = UUID.randomUUID();
    return new Feed(uuid.toString());
  }
}
```

4. Finally, we also need to specify the servlet for our service in the `war\WEB-INF\web.xml` by adding the following lines:

```
<servlet>
  <servlet-name>feedServlet</servlet-name>
  <servlet-class>
    com.danielvaughan.rssreader.server.services.FeedServiceImpl
</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>feedServlet</servlet-name>
  <url-pattern>/rssreader/feed-service</url-pattern>
</servlet-mapping>
```

5. The project structure should now look like this:



What just happened?

We created a GWT RPC service to deal with creating new `Feed` objects with a unique ID.

The Registry

GXT contains a class named the `Registry`. The registry is a `HashMap` of data that is available globally throughout the client of the application. The registry can be used for storing data and state in the client. However, there are better ways of doing this that we will be covering in later chapters.

Storing the service in the Registry

The `registry` is useful for storing items other than data. As items placed in the registry are available throughout the client code, it is also a good place to store services so that they can be created once, stored in the `Registry`, and retrieved when needed. We will create an instance of our asynchronous `FeedServiceAsync` and put it in the `Registry` using a constant named `FEED_SERVICE` as a key. We will then retrieve this service in another part of the code and make use of it.

Time for action – using the Feed object

1. First of all, we need to create a class to contain the constants we will use as keys for the objects that we will use to add to the registry. Name this class `RSSReaderConstants` in the `client` package and add a `FEED_SERVICE` constant:

```
public class RSSReaderConstants {
  public static final String FEED_SERVICE = "feedService";
}
```

2. Now at the beginning of the `onModuleLoad` method of the main `RSSReader` class, add a new instance of the `FeedService` to the registry:

```
Registry.register(RSSReaderConstants.FEED_SERVICE,
  GWT.create(FeedService.class));
```

3. In the `RssNavigationPanel`, modify the `createNewFeedWindow` method so that it retrieves the `FeedService` from the registry:

```
private void createNewFeedWindow()
{
  final FeedServiceAsync feedService =
    Registry.get(RSSReaderConstants.FEED_SERVICE);
}
```

4. Now call the `createNewFeed` method of the `FeedService` and then if the GWT RPC call is successful, create and display a new `FeedWindow` using the `Feed` object retrieved as a parameter:

```
private void createNewFeedWindow()
{
  final FeedServiceAsync feedService =
    Registry.get(RSSReaderConstants.FEED_SERVICE);
  feedService.createNewFeed(new AsyncCallback<Feed>() {
    @Override
    public void onFailure(Throwable caught) {
      Info.display("RSSReader", "Unable to create a new
        feed");
    }

    @Override
    public void onSuccess(Feed feed) {
      final Window newFeedWindow = new FeedWindow(feed);
      newFeedWindow.show();
    }
  });
}
```

What just happened?

We have now used our `FeedService` to create a new `Feed` object, which we passed to the `FeedWindow` so that the user can complete the details.

Saving a Feed

We now need to create a `save` method in our `FeedForm` to move the data from the controls into the `Feed` object, and a button for the `FeedWindow` to call the feed form's `save` method.

Time for action – saving an object to the registry

1. In the `FeedForm`, create a `save` method, which checks that the form is valid, and if so, sets the properties of a `Feed` object to the values inputted by the user:

```
public void save(final Feed feed) {
  feed.setTitle(tfTitle.getValue());
  feed.setDescription(taDescription.getValue());
  feed.setLink(tfLink.getValue());
}
```

2. In the constructor of `FeedWindow`, add a **Save** `Button` that, when clicked, calls the save method of the `FeedForm` with the `Feed` as a parameter:

```
public FeedWindow(final Feed feed) {
  setHeading("Feed");
  setWidth(350);
  setHeight(200);
  setResizable(false);
  setLayout(new FitLayout());

  final Button btnSave = new Button("Save");
  btnSave.setIconStyle("save");
  btnSave.addSelectionListener(new
    SelectionListener<ButtonEvent>()
  {
    public void componentSelected(ButtonEvent ce) {
      btnSave.setEnabled(false);
      if (feedForm.isValid()) {
        hide(btnSave);
      feedForm.save(feed);
      } else {
        btnSave.setEnabled(true);
      }
    }
  });
  addButton(btnSave);
}
```

3. Finally, add a CSS style to `war\RSSReader.css` for the **Save** button:

```
.save {
  background: url(gxt/images/icons/disk.png) no-repeat center
    left
    !important;
}
```

4. Start the application and check that the `FeedForm` window now has a **Save** button:

What just happened?

We added a `save` method to the `FeedForm`, which stores the values entered to a `Feed` object. We then added a **Save** button to the `FeedWindow`, which calls the `save` method of the `FeedForm` it contains.

Creating RSS XML

Later on, we are going to have to save our feeds on the server so that we can load them again. To do this, we are going to save them as an RSS XML document. We are now going to add a new `saveFeed` method to our `FeedService` that takes a feed and sends it to the server for processing.

We shall process the `Feed` object and turn it into XML using the JDOM 1.1 library. This can be downloaded from `http://www.jdom.org/downloads/index.html`. Download the version suitable for your platform and unzip the archive to a suitable location.

Time for action – saving a Feed

1. In the `FeedService` interface, add a new `saveFeed` method, which takes a `Feed` object as its only argument:

```
void saveFeed(Feed feed);
```

2. Add the corresponding method in the `FeedServiceAsync` interface:

```
void saveFeed(Feed feed, AsyncCallback<Void> callback);
```

3. Locate `jdom.jar`, which is in the build folder of the extracted JDOM archive. Copy the `jdom.jar` to the project's `war\WEB-INF\lib` folder and add `jdom.jar` to the project's class path.

4. In `FeedServiceImpl`, implement the `saveFeed` method so that it creates a new JDOM XML document and populates the elements with data from the `Feed` object:

```
public void saveFeed(Feed feed) {
  Element eleRoot = new Element("rss");
  eleRoot.setAttribute(new Attribute("version", "2.0"));

  //Create a document from the feed object
  Document document = new Document(eleRoot);

  Element eleChannel = new Element("channel");
  Element eleTitle = new Element("title");
  Element eleDescription = new Element("description");
  Element eleLink = new Element("link");

  eleTitle.setText(feed.getTitle());
  eleDescription.setText(feed.getDescription());
  eleLink.setText(feed.getLink());

  eleChannel.addContent(eleTitle);
  eleChannel.addContent(eleDescription);
  eleChannel.addContent(eleLink);

  eleRoot.addContent(eleChannel);
}
```

5. Now add code to take the document and serialize it to the console:

```
public void saveFeed(Feed feed) {
…

  try {
    XMLOutputter serializer = new XMLOutputter();
    Format prettyFormat = Format.getPrettyFormat();
    serializer.setFormat(prettyFormat);
    System.out.println("At this point we would serialize
      the feed " + feed.getTitle() + " to a file. For
      now we are just going to write it to the
      console.");
    serializer.output(document, System.out);
  } catch (IOException e) {
    System.out.println("Error saving feed");
  }
}
```

6. In the `save` method of the `FeedForm` class, retrieve the `FeedService` from the registry and call the `save` method with the `Feed` as a parameter:

```
public void save(final Feed feed) {
  feed.setTitle(tfTitle.getValue());
  feed.setDescription(taDescription.getValue());
  feed.setLink(tfLink.getValue());

  final FeedServiceAsync feedService = Registry
      .get(RSSReaderConstants.FEED_SERVICE);
  feedService.saveFeed(feed, new AsyncCallback<Void>() {
    @Override
    public void onFailure(Throwable caught) {
      Info.display("RSS Reader", "Failed to save feed: "
          + feed.getTitle());
    }

    @Override
    public void onSuccess(Void result) {
      Info.display("RSS Reader", "Feed " +
        feed.getTitle()
        + " saved sucessfully");
    }
  });
}
```

7. Finally, start the application, click on the **Create feed** button, complete the form, and click on the **Save** button. Check that the RSS XML document appears on your console like this:

```
<rss version="2.0">
  <channel>
    <title>Example Feed</title>
    <description>This is an example feed</description>
    <link>http://www.example.com/</link>
  </channel>
</rss>
```

What just happened?

We created a mechanism for creating an RSS XML document when we saved a `Feed` object and made that available as part of the GWT RPC `FeedService`.

Now that we have a service for dealing with feeds, we can also perform validation on the URL entered in the `LinkFeedPopup` we created in the last chapter and use the service to process the URL.

Time for action – adding to the LinkFeedPopup

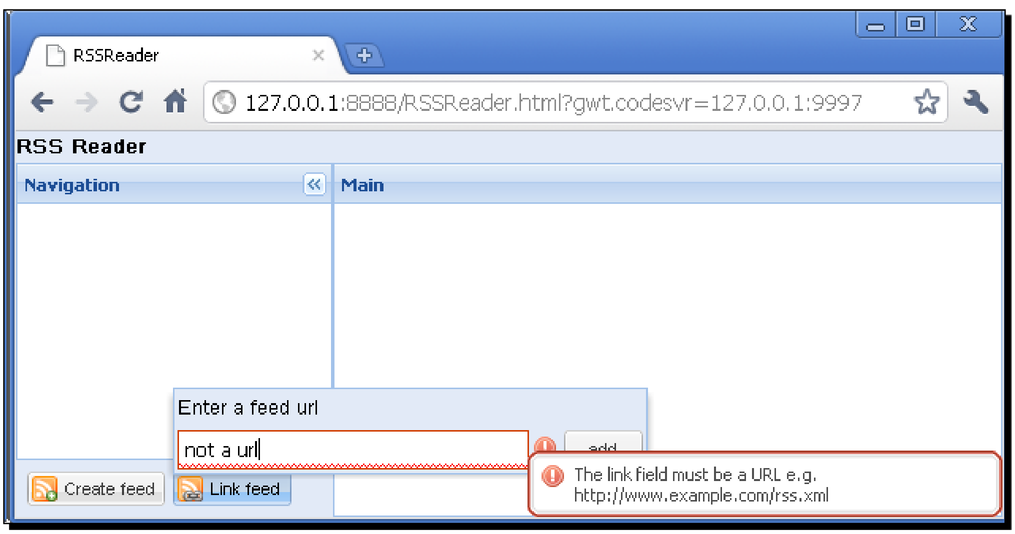
1. In the `onRender` method of the `LinkFeedPopup`, set the validation of the `tfUrl` to require an URL and to display to appropriate field messages if the validation fails. Also, use the `setAutoValidate` method to cause the validation to happen each time a character is entered:

```
tfUrl.setAllowBlank(false);
tfUrl.setRegex("^http\\://[a-zA-Z0-9\\-\\.]+\\.[a-zA-
  Z]{2,3}(/\\S*)?$");
tfUrl.setAllowBlank(false);
tfUrl.getMessages().setBlankText("Please enter the URL of an
  existing feed");
tfUrl.setAutoValidate(true);
tfUrl.getMessages()
  .setRegexText(
    "The link field must be a URL e.g.
      http://www.example.com/rss.xml");
```

2. When validation fails, an icon will be displayed to the right of the URL field. To accommodate this icon, we need to increase the right margin specified in the `eastData` to `20` pixels:

```
final BorderLayoutData eastData = new
BorderLayoutData(LayoutRegion.EAST, 50);
eastData.setMargins(new Margins(2,2,2,20));
add(btnAdd, eastData);
```

3. Start the application and try to enter a string that is not a URL and check that you get an error message like this:



What just happened?

We added validation to the `LinkFeedPopup`.

Have a go hero – create a new item form

In the next chapter, we shall be adding the ability to add items to feeds. To do this, we will need another form named `ItemForm`. For now, the form should take the similar input as the `FeedForm`. This time, however, all the fields are optional, but either the title or the description must be completed.

Have a go at creating the following:

- An `ItemForm` class
- An `ItemWindow` class
- An `Item` class
- `ItemService`, `ItemServiceAsync`, and `ItemServiceImpl` classes that allow you to create a new Item with its own UUID.

Summary

In this chapter, we have looked at forms and the components we can use to build them. We then moved on to display a form using windows and how to store data retrieved from them in the registry. Finally, we sent a completed `Feed` object to the server via GWT RPC and transformed it into an RSS XML file.

In the next chapter, we will look at how we can start working with data using GXT's built-in data handling features.

4
Data-backed Components

In this chapter, we introduce how GXT allows us to work with data. We look at the components available for retrieving, manipulating, and processing data, and then move on to work with the built-in data-backed display components.

We shall cover the following components:

Data

- ModelData
- BeanModel
- BeanModelTag
- BeanModelMarker
- BeanModelFactory
- Stores

Remote Data

- DataProxies
- DataReaders
- ListLoadResults
- ModelType
- Loaders
- LoadConfigs

Data-backed components

- ListField
- ComboBox
- Grid
    - ColumnModel
    - ColumnConfig
    - GridCellRenderer

Working with data

One of the advantages of AJAX applications, including those built with GXT, is the ability to manipulate data in the browser. GXT provides useful data-backed visual components that allow us to work with local data such as lists, combos, and grids. With them we can perform sorting, filtering, and editing operations on data quickly and efficiently.

There is also another set of components that work in the background allowing us to retrieve remote data, cache it on the client, and deliver it to the visual components. It is these two sets of components that we are going to focus on in this chapter.

First of all, we are going to look at how to produce the data that we need for display in the visual components.

ModelData interface

GXT provides us an interface named `ModelData`. Any data objects we wish to use with GXT data-backed components must implement this interface. The `ModelData` interface provides the ability for property names and values to be retrieved at runtime. As GWT does not support reflection, GXT does this using a form of introspection.

In our example application, at present we are using a JavaBean named `Feed` to store feed data. However, at the moment, it does not implement the `ModelData` interface, so we cannot use it with GXT's data-backed components.

We have three methods that will allow us to achieve this:

1. Modify the `Feed` JavaBean so that it extends `BaseModel`.
2. Modify the `Feed` JavaBean so that it implements `BeanModelTag`.
3. Create a `BeanModelMarker` interface to accompany the `Feed` JavaBean.
   This method allows us to avoid having to modify the `Feed` JavaBean.

Method 1: Extending BaseModel

`BaseModel` is the default implementation of the `ModelData` interface. Classes that extend `BaseModel` make use of a `HashMap` to store data rather than local fields. Data is added and retrieved using the `set` and `get` methods of the `BaseModel` respectively. The downside of this method is that we need to use strings as attribute names and as such it is easier for errors to creep in.

The `Feed` object implemented as a subclass of `BaseModel` would look like this:

```
public class Feed extends BaseModel {

  public Feed () {

  }

  public Feed (String uuid) {
    set("uuid", uuid);
  }

  public String getDescription() {
    return get("description");
  }

  public String getLink() {
    return get("link");
  }

  public String getTitle() {
    return get("title");
  }

  public String getUuid() {
    return get("uuid");
  }

  public void setDescription(String description) {
    set("description", description);
  }

  public void setLink(String link) {
    set("link", link);
  }

  public void setTitle(String title) {
    set("title", title);
  }
}
```

BeanModel class

If we already have a JavaBean we wish to use instead of creating a new `BaseModel`, GXT provides the `BeanModel`, a class which acts as a wrapper for JavaBeans. `BeanModel` objects cannot be created directly; instead they are generated by a `BeanModelFactory`.

BeanModelFactory class

`BeanModelFactory` is a useful class that allows us to take a JavaBean with a corresponding `BeanModelMarker` interface such as a `Feed` object, and get back a `BeanModel` representation.

The remaining two methods of providing a `ModelData` object involve wrapping a JavaBean as a `BeanModel`.

Method 2: Implementing BeanModelTag

`BeanModelTag` is an interface that allows us to tag existing Java objects that meet the JavaBean specification. This allows `BeanModel` instances to be generated from the JavaBean using a `BeanModelFactory`.

In order to make our existing `Feed` JavaBean usable as a GXT `BeanModel`, we simply need to implement the `BeanModelTag` interface like this:

```
public class Feed implements Serializable, BeanModelTag {

  private String description;
  private String link;
  private String title;
  private String uuid;

  public Feed() {

  }

  public Feed(String uuid) {
    this.uuid = uuid;
  }

  public String getDescription() {
    return description;
  }

  public String getLink() {
```

```
      return link;
    }

    public String getTitle() {
      return title;
    }

    public String getUuid() {
      return uuid;
    }

    public void setDescription(String description) {
      this.description = description;
    }

    public void setLink(String link) {
      this.link = link;
    }

    public void setTitle(String title) {
      this.title = title;
    }
  }
```

This still requires a change to the JavaBean, however. There are situations where making any change to a JavaBean would be unacceptable or at least undesirable. In this case, GXT provides the `BeanModelMarker`.

Method 3: Creating a BeanModelMarker

`BeanModelMarker` is an interface, which, as its name suggests, allows us to mark an existing JavaBean as a `BeanModel`. This is achieved by creating an interface that extends `BeanModelMarker`. It makes use of annotations to define the JavaBean to wrap.

In our example application, we already have a `Feed` JavaBean, and we will now create a `BeanModelMarker` for it so that we can use it with the GXT data-backed controls.

Notice the use of the `@BEAN` annotation to make a reference to the `Feed` class. We do not need to make any changes to the `Feed` JavaBean itself.

Time for action – creating a BeanModelMarker for Feed objects

1. Create a new class named `FeedBeanModel` that implements the `BeanModelMarker` interface in a new package named `client.model`:

   ```
   public class FeedBeanModel implements BeanModelMarker {}
   ```

2. Add an `@BEAN` annotation to tell GXT to use the `Feed` JavaBean class as follows:

   ```
   @BEAN(com.danielvaughan.rssreader.shared.model.Feed.class)
   public class FeedBeanModel implements BeanModelMarker {}
   ```

What just happened?

We created a `BeanModelMarker` for our existing `Feed` JavaBean. We can now use our `Feed` JavaBean with GXT's data-backed controls without having to modify the `Feed` JavaBean class in any way.

Stores

In GXT, a `Store` is an abstract class used to provide a client-side cache of `ModelData` objects of a specified class. Stores are where the data-backed GXT components keep data. There are two concrete `Store` classes. The first is `TreeStore`, which is used with `Tree` components, and we will look at these in the next chapter. The second is `ListStore`, which is used for storing the lists of data. These are typically used with `ListField`, `ComboBox`, and `Grid` components.

To create a `ListStore` to contain `Feed` objects, we would define it like this:

```
ListStore<BeanModel> feedStore = new ListStore<BeanModel>();
```

Note that the `ListStore` is set to contain `BeanModel` instances. This is because a `Store` can only contain objects that inherit from the `ModelData` class. If we wanted to add a `Feed` JavaBean object, we cannot do it directly. We need to use a `BeanModelFactory` to convert the `Feed` JavaBean object into a `BeanModel` representation.

We will now modify the example application so that when creating a new `Feed` JavaBean object, a `BeanModel` representation of the `Feed` object is added into a client-side `ListStore`.

Time for action – creating and populating a ListStore

1. In the `RSSReaderConstants` class, add a new constant named `FEED_STORE` for the feed store:

```
public static final String FEED_STORE = "feedStore";
```

2. In the `onModuleLoad` method of the RSSReader class, create a new `ListStore` and add it to the `Registry` using the `FEED_STORE` constant as the key:

```
public void onModuleLoad() {
  Registry.register(RSSReaderConstants.FEED_SERVICE,
    GWT.create(FeedService.class));
  Registry.register(RSSReaderConstants.FEED_STORE,  new
    ListStore<BeanModel>());
  …
}
```

3. In the save method of the `FeedForm`, modify the `onSuccess` method of the callback function to retrieve the feed store from the `Registry`:

```
public void save(final Feed feed) {

  @Override
  public void onSuccess(Void result) {
    Info.display("RSS Reader", "Feed " + feed.getTitle()
      + " saved successfully");
    final ListStore<BeanModel> feedStore = Registry
      .get(RSSReaderConstants.FEED_STORE);
  }
…
}
```

4. Retrieve a `BeanModelFactory` for the `Feed` class:

```
  public void onSuccess(Void result) {
    Info.display("RSS Reader", "Feed " + feed.getTitle() + " saved
      successfully");
    final ListStore<BeanModel> feedStore =
      Registry.get(RSSReaderConstants.FEED_STORE);
    BeanModelFactory beanModelFactory =
      BeanModelLookup.get().getFactory(feed.getClass());
}
```

5. Finally, use the `BeanModelFactory` to create a `BeanModel` representation of the `Feed` object and then add it to the feed store:

```
  public void onSuccess(Void result) {
    Info.display("RSS Reader", "Feed " + feed.getTitle() + " saved
      successfully");
    final ListStore<BeanModel> feedStore =
      Registry.get(RSSReaderConstants.FEED_STORE);
    BeanModelFactory beanModelFactory =
      BeanModelLookup.get().getFactory(feed.getClass());
    feedStore.add(beanModelFactory.createModel(feed));
}
```

What just happened?

`Feed` objects are now stored in a GXT `ListStore`. The advantage of this is that we can now simply link the data-backed components to the `Store`, and the values in the components will refresh automatically.

Data-backed ComboBox

Once we have a `ListStore` populated with data, we can use it to provide the options in a `ComboBox` by binding the `ComboBox` to the `Store`. We would take the feed store and create a `ComboBox` that uses the title field of each `Feed` to populate the values of the `ComboBox` like this:

```
ComboBox<Feed> combo = new ComboBox<Feed>();
combo.setDisplayField("title");
combo.setStore(feeds);
```

Here we use the `setDisplayField` of the `ComboBox` to set the title field as the field to use as the display value.

Once a data-backed component is linked with a `Store`, it then observes the `Store` for changes. If a change to the data in the `Store` occurs, such as an object being added to the `Store`, the content of the data-backed control will be updated automatically. The specific Store events that can be observed are listed as follows:

- Add
- Clear
- Data Changed
- Filter
- Remove
- Sort
- Update

Data-backed ListField

Associating a `ListField` with a `Store` is very similar to associating a `ComboBox` with a `Store`. In our example application, we will now add a `ListField` containing all current feeds to the left navigation panel.

Time for action – creating a ListField for feeds

1. Create a new package named `client.lists`, and in it create a new class named `FeedList` that extends `LayoutContainer`:

   ```
   public class FeedList extends LayoutContainer {}
   ```

2. In the constructor of the `FeedList` class, set the layout of the `LayoutContainer` to `FitLayout`:

   ```
   public FeedList() {
     setLayout(new FitLayout());
   }
   ```

3. Override the `onRender` method, and in it create a new `ListField`:

   ```
   @Override
   protected void onRender(Element parent, int index) {
     super.onRender(parent, index);
     final ListField<BeanModel> feedList = new
       ListField<BeanModel>();
   }
   ```

4. Again in the `onRender` method, retrieve the feed store from the `Registry`:

   ```
   final ListStore<BeanModel> feedStore =
     Registry.get(RSSReaderConstants.FEED_STORE);
   ```

5. Set the feed store as the `Store` for the feed list `ListField`:

   ```
   feedList.setStore(feedStore);
   ```

6. Tell the `ListField` to use the title field of the `Feed` object as the value to display in the `ListField`:

   ```
   feedList.setDisplayField("title");
   ```

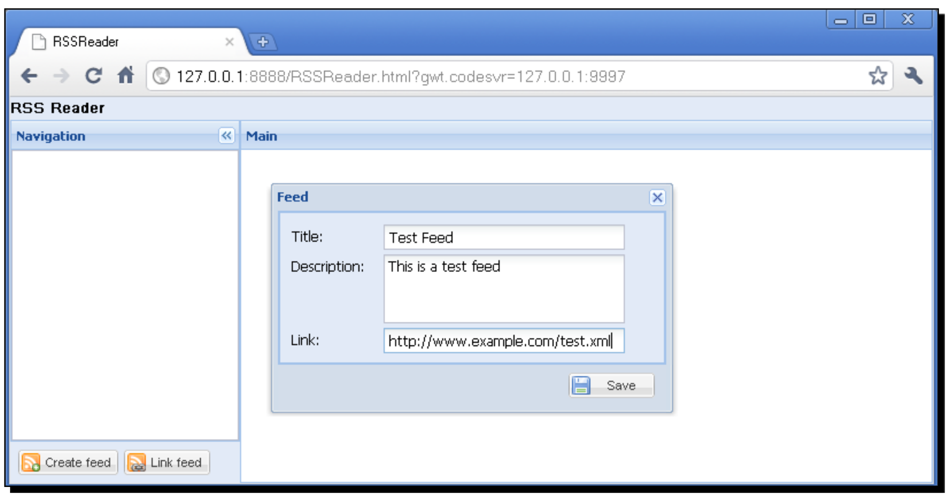
7. Add the `ListField` to the underlying container:

   ```
   add(feedList);
   ```

8. At the end of the constructor of the `RssNavigationPanel`, set the layout to a new instance of `FitLayout` and add a new instance of the `FeedList` component to the underlying `ContentPanel`:

```
setLayout(new FitLayout());
add(new FeedList());
```

9. Start the application, click on the **Create feed** button, and complete the form as shown in the screenshot below:



10. On clicking the **Save** button, the new feed's title, **Test Feed**, will appear in the list of feeds on the left:

What just happened?

We added a list of feeds to the RSS Reader application. When we created a new feed, it automatically appeared in the feed list on the left.

Server-side persistence

So far we do not have any persistence in our example application.

To make the example more realistic, we will now add persistence. As this is a GXT book rather than a GWT book, we will only implement basic server-side persistence so that we can concentrate on the client-side. However, we will put the actual persistence logic behind an interface so that we can replace it with another implementation later if required. For the initial implementation:

- When creating new feeds, we will simply save an XML file on the server-side.
- We will keep a list of the URLs of the feeds that we have created and any imported feeds in a simple text file.

The persistence code implementation is not GXT-specific, so it can be treated as a black box. The `Persistence` interface and a `FilePersistence` implementation of that interface can be found in the example code. It is this that we are going to make use of to store and retrieve RSS feeds.

Persisting an Existing Feed

In the second chapter, we created the **Link feed** button, the purpose of which was to let our RSS reader import an existing RSS feed from the Internet. We will now create an `addExistingFeed` method in the `FeedService` that with the help of the persistence layer stores the URL of the feed for later retrieval. We will then connect this method to the **add** button on the `LinkFeedPopup`.

Time for action – persisting a link to an existing feed

1. Add an `addExistingFeed` method to the `FeedService` interface that takes the URL of a feed as an argument:

```
void addExistingFeed(String feedUrl);
```

2. Add a corresponding asynchronous version of the `addExistingFeed` method to the `FeedServiceAsync` interface:

```
void addExistingFeed(String feedUrl, AsyncCallback<Void>
  callback);
```

3. Modify the `addFeed` method of the `LinkFeedPopup` class so that it retrieves the `FeedService` and calls the `addExistingFeed` method with the URL that the user has entered. If successful, the method should clear the URL `TextField` and hide the `Popup`:

```
public void addFeed(final String feedUrl) {
  final FeedServiceAsync feedService = Registry
    .get(RSSReaderConstants.FEED_SERVICE);
  feedService.addExistingFeed(feedUrl, new AsyncCallback<Void>()
{
    @Override
    public void onFailure(Throwable caught) {
      Info.display("RSS Reader", "Failed to add feed at: " +
        feedUrl);
  }

    @Override
    public void onSuccess(Void result) {
      tfUrl.clear();
      Info.display("RSS Reader", "Feed at " + feedUrl
        + " added successfully");
      hide();
    }
  });
}
```

4. In the `FeedServiceImpl` class, create a new Java Logging logger for the class:

```
private final static Logger LOGGER =
  Logger.getLogger(FeedServiceImpl.class
  .getName());
```

5. Again in the `FeedServiceImpl` class, create a new private method named `loadFeed`. The method takes an URL string and uses JDOM to retrieve the XML of an RSS from the Internet. It then parses the XML into a `Feed` object. This is implemented as follows:

```
  private Feed loadFeed(String feedUrl) {
    Feed feed = new Feed(feedUrl);
    try {
      SAXBuilder parser = new SAXBuilder();
      Document document = parser.build(new URL(feedUrl));
      Element eleRoot = document.getRootElement();
      Element eleChannel = eleRoot.getChild("channel");
      feed.setTitle(eleChannel.getChildText("title"));
      feed.setDescription(eleChannel.getChildText("description"));
      feed.setLink(eleChannel.getChildText("link"));
      return feed;
```

```
    } catch (IOException e) {
      LOGGER.log(Level.SEVERE, "IO Error loading feed", e);
      return feed;

    } catch (JDOMException e) {
      LOGGER.log(Level.SEVERE, "Error parsing feed", e);
      return feed;
    }
  }
```

6. Create a new instance of a `HashMap` to store `Feed` objects with their URL as a key:

```
private Map<String, Feed> feeds = new HashMap<String, Feed>();
```

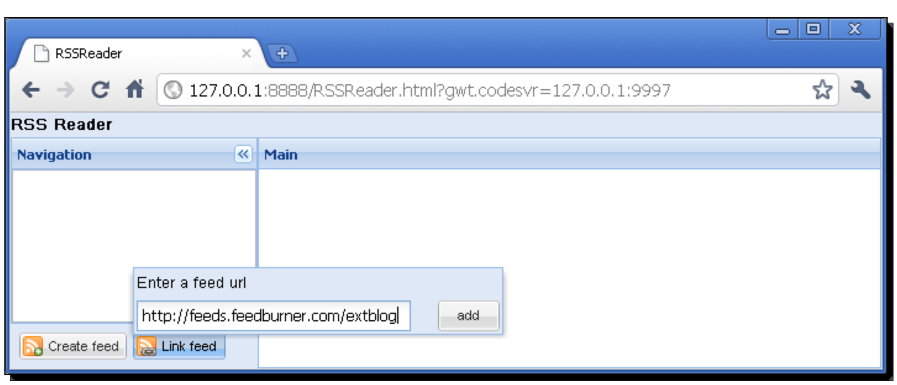
7. Create a new instance of the `FilePersistence` class:

```
private final Persistence persistence = new FilePersistence();
```

8. Implement the `addExistingFeed` method so that it uses the `loadFeed` method to retrieve the `Feed` object corresponding to the provided URL `String`. Check that the `Feed` has a title and add it to the `HashMap` of `Feed` objects and then use the `saveFeedList` method of the `FilePersistence` class to persist the updated list:

```
  @Override
  public void addExistingFeed(String feedUrl) {
    Feed loadResult = loadFeed(feedUrl);
    if (loadResult.getTitle() != null) {
      feeds.put(feedUrl, loadFeed(feedUrl));
      persistence.saveFeedList(feeds.keySet());
    }
  }
```

9. Start the application and add an existing feed by clicking on the **Link feed** button, entering an URL in the link feed popup, and clicking on the **add** button:



10. In the `war\data` folder, check that there is now a text file named `feed.txt` containing the URL that you entered in the user interface.

What just happened?

We used the server to retrieve the RSS XML from a specified URL and persist the URL on the server for later use in the application.

At the moment, if we create a new feed and click on the **Save** button, a feed object is added to the feed `ListStore` on the client, but this is just a cache and not a persistent data store.

In the last chapter, we created a GWT RPC service with a `saveFeed` method. This sent a `Feed` object to the backend, but so far all it does is convert the feed object to XML and print the result to the console. The XML is not saved, and any `Feed` objects created will be lost when the application is restarted. We will now add to our implementation of the `saveFeed` method so that it makes use of file persistence.

Time for action – persisting a feed as an XML document

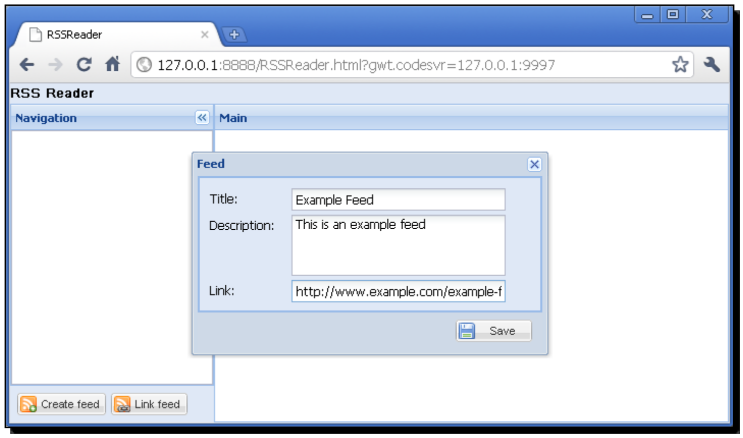
1. In the `FeedServiceImpl` class, remove the existing XML serialization code at the end of the `saveFeed` method and in its place add a call to the `saveFeedXml` method of the `Persistence` interface. The call should include the UUID of the `Feed` object and the JDOM document generated from it. This will write a file containing an XML representation of the `Feed`:

```
persistence.saveFeedXml(feed.getUuid(),document);
```

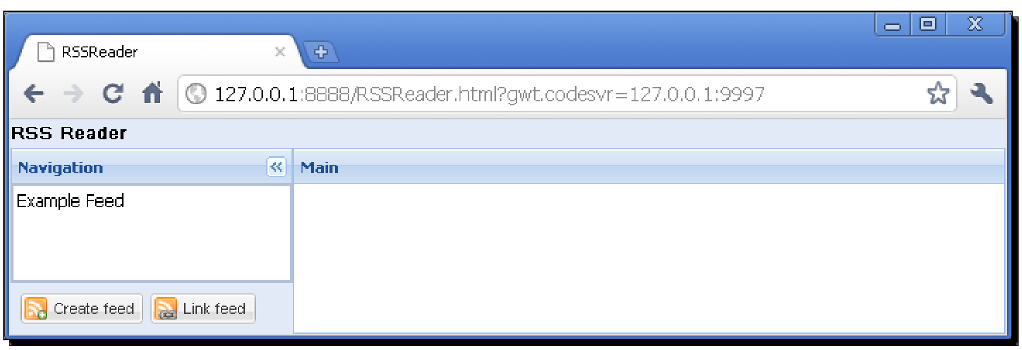
2. Also append a call to the `addExistingFeed` method. The parameter for this call should include the URL to the XML file created on the file system. This is retrieved from the `getUrl` method of the `Persistence` interface:

```
addExistingFeed(persistence.getUrl(feed.getUuid()));
```

3. Start the application and create a new feed using the **Create feed** button, complete the form as follows, and click on the **Save** button:



4. Check that **Example Feed** appears in the feeds `ListField`:



5. Finally, check that a new file has appeared in the `war\data` folder. It will have a filename in the format of `<uuid>.xml`: for example, `4a529f45-31af-4375-9da4-4b03280a4784.xml`. Check that the file contains the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>Example feed</title>
    <description>This is an example feed</description>
    <link>http://www.example.com/example-feed.xml</link>
  </channel>
</rss>
```

What just happened?

We created a mechanism to persist the XML representation of a `Feed` object as an XML file.

Server-side retrieval

Once feeds have been saved on the server, we need to be able to load them when we start the application again. For this we will add a `loadFeedList` method to the feed service. This will return `Feed` objects by loading RSS feeds from the URLs stored in the `feeds.txt` file. We will then add any `Feed` objects we retrieve into the client's feed store.

Time for action – loading feeds

1. Add a `loadFeedList` method to the `FeedService` interface:

```
List<Feed> loadFeedList();
```

2. Add an asynchronous version of the `loadFeedList` method to the `FeedServiceAsync` interface:

```
void loadFeedList(AsyncCallback<List<Feed>> callback);
```

3. In the `FeedServiceImpl` class, implement the `loadFeedList` method so that it populates the feeds `HashMap` using the `loadFeedList` method of the `Persistence` interface, and then returns a list of the `Feed` objects now contained within it:

```
@Override
public List<Feed> loadFeedList() {
  feeds.clear();
  Set<String> feedUrls = persistence.loadFeedList();
  for (String feedUrl : feedUrls) {
    feeds.put(feedUrl, loadFeed(feedUrl));
  }
  return new ArrayList<Feed>(feeds.values());
}
```

What just happened?

We added the ability to retrieve `Feed` objects from previously persisted XML files via a call to a GWT RPC service. We now have a service that persists `Feed` objects and allows us to retrieve them again.

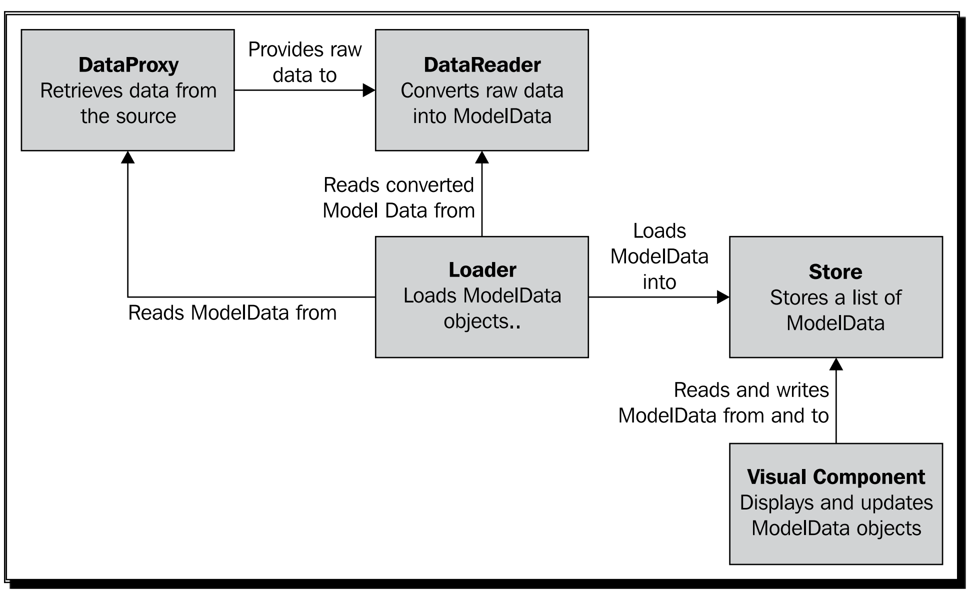
Using remote data

As well as populating stores with data on the client-side, we can also populate stores by retrieving remote data. GXT provides facilities for loading and working with remote data, be it XML or JSON retrieved via HTTP, or objects retrieved through GWT RPC. For each source, GXT provides mechanisms to retrieve and read data. If necessary, it can also convert the raw data into `ModelData` and then automatically add it into a `Store`.

There are several components involved in this process. They all perform a function in their own right but come together to retrieve and process data:

- **DataProxy**—retrieves the raw data from the source
- **DataReader**—takes the raw data and converts it into **ModelData**
- **Loader**—loads the processed data into the store automatically

The interaction between the various components is summarized in this diagram:



DataProxy interface

A `DataProxy` is used to retrieve raw data. All data proxies implement GXT's `DataProxy` interface. There are a number of `DataProxy` implementations, that retrieve different types of data in different ways:

DataProxy	Description
`HttpProxy`	Retrieves data using a GWT `RequestBuilder` instance to retrieve XML or JSON from the same server.
`MemoryProxy`	Simply passes on the data specified in its constructor.
`PagingModelMemoryProxy`	Like a `MemoryProxy`, but supports paging where all the data is in memory.
`RpcProxy`	Retrieves data using GWT RPC, but allows the conversion of JavaBeans for use with a loader.
`ScriptTagProxy`	Retrieves data from an URL, which may be in a domain other than the originating domain of the running page—that is, gets around cross-site scripting. Only works with JSON data.

Once we have retrieved data using a `DataProxy`, if it is not represented as `ModelData` objects already, we will need to convert it using a `DataReader` before it can be loaded into a `Store`.

DataReader interface

Data readers translate raw data into `ModelData` objects. All data readers implement GXT's `DataReader` interface. There are several different implementations of `DataReader` that deal with different raw data. A `DataReader` returns the results as one of the following:

- A set of `ModelData` objects.
- An object that implements the `ListLoadResult` interface. `ListLoadResult` contains one method, `getData()` that returns the data as `ModelData` objects.
- A `PagingLoadResult` object, which extends `ListLoadResult` to provide support for paging—that is, the ability to return a subset of the data.

> Paging is when the data is not displayed all at once, but instead presented in pages. For example, if you had 100 results and wanted to display them 10 per page in a grid, you may have a control at the bottom that displays 1-10 of 100, and a button to move to the next page. We will cover paging in later chapters.

There are a number of different implementations of `DataReader` summarized in this table:

DataReader	Data in	Converted using	Data out	Use when
`ModelReader`	`Model Data`	Not applicable as the `ModelData` is just packaged into a `ListLoadResult` object	`ListLoadResult`	Loading objects that already extend `BeanModel`
`BeanModelReader`	A list of JavaBeans	`BeanModel Factory`	`ListLoadResult`	Loading JavaBean objects that need to be converted to `BeanModel` objects
`JsonReader`	JSON data	`ModelType` definition	Set of `ModelData` instances	Loading JSON data as `ModelData`
`JsonLoadResult Reader`	JSON data	`ModelType` definition	`ListLoadResult`	Loading JSON data as `ModelData`
`JsonPagingLoadResult Reader`	JSON data	`ModelType` definition	`PagingLoadResult`	Loading JSON data as paged `ModelData`
`XmlReader`	XML data	`ModelType` definition	Set of `ModelData` instances	Loading XML data as `ModelData`
`XmlLoadResultReader`	XML data	`ModelType` definition	`ListLoadResult`	Loading XML data as `ModelData`
`XmlPagingLoadResult Reader`	XML data	`ModelType` definition	`PagingLoadResult`	Loading XML data as `ModelData` in pages

There is also a `TreeModel`-specific `DataReader` that we will cover in later chapters.

You may notice that an object called `ModelType` is used by many of the data readers to perform conversion of the data.

ModelType class

`ModelType` defines the structure of the raw data to enable the `DataReader` that deals with XML or JSON data to map the raw data from XML or JSON `ModelData` objects.

For example, if we had source data that was an XML document with this structure:

```
<books>
  <book>
    <title>The best book in the world</title>
  </book>
  <book>
    <title>The worst book in the world</title>
  </book>
</books>
```

the `ModelType` definition would look like this:

```
final ModelType modelType = new ModelType();
modelType.setRoot("books");
modelType.setRecordName("book");
modelType.addField("title");
```

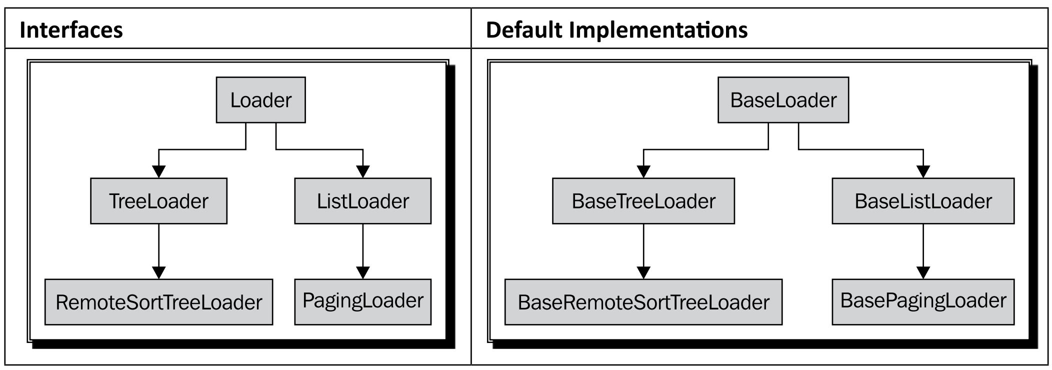
The root of the XML document is `books`, so we indicate this using the `setRoot` method. The record is `book`, so we use `setRecord` name to indicate this. Finally, a field of book is `title`, so we add this to the model using `addField`.

`ModelType` can also be used with JSON data. In fact, the `ModelType` definition we just used to represent the XML could be used with the following JSON without modification:

```
{
  "books": [
    {
      "book": {
        "title": "The best book in the world"
      },
      "book": {
        "title": "The worst book in the world"
      }
    }
  ]
}
```

Loader interface

Loaders are used for loading `ModelData` into a store, given a `DataProxy` and a `DataReader`. The base interface for all loaders is **Loader**, the abstract implementation of which is **BaseLoader**:



There are two types of loaders: one for lists that implement the **ListLoader** interface and one for trees that implement the **TreeLoader** interface. We will leave the tree loaders until later chapters and at the moment just look at the list loaders.

The default implementation of the **ListLoader** interface is **BaseListLoader**. There is also **BasePagingLoader**, which extends the functionality of **BaseListLoader** to add paging support and implements the **PagingLoader** interface.

Loaders can also sort data when loading it. This can be defined either by using the `setSortField` and `setSortDir` methods, or by specifying a `LoadConfig` object.

LoadConfig

`LoadConfigs` define how data is loaded. The `LoadConfig` interface has a number of implementations, including one only used for tree data. For now we will just look at the list implementations.

The first is `BaseListLoadConfig`, which allows you to specify how the data is sorted when loading.

`BaseListLoadConfig` also has two subclasses that refine loading further:

- `BaseGroupingLoadConfig`—that allows you to group data by a specified field using the `setGroupBy` method
- `BasePagingLoadConfig`—that provides paging support

How they fit together

Here is a summary of how the various backend components fit together:

- GXT uses classes that implement the `ModelData` interface to store information
- Stores provide a cache of `ModelData` on the client and provide this to data-backed components
- DataProxies retrieve the raw data from a remote source
- `ModelType` describes the structure of the raw data
- Certain DataReaders use the `ModelType` to define how to take raw data and produce `ModelData`
- Loaders load the data into the store using a `DataProxy` and a `DataReader`
- LoadConfigs optionally tell the loader how to sort, group, or page the `ModelData`

We will now return to our `FeedList` class and modify it to use an `RpcProxy`, `BeanModelReader`, and a `ListLoader` to populate the `ListStore` of the `ListField` using remote data.

Time for action – using remote data with a ListField

1. In the `onRender` method of the `FeedList` class, remove the line that retrieves the feed store from the `Registry`, and instead retrieve the feed service from the `Registry`:

```
@Override
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  final ListField<BeanModel> feedList = new
  ListField<BeanModel>();
  final FeedServiceAsync feedService = (FeedServiceAsync)
  Registry.get(RSSReaderConstants.FEED_SERVICE);
  feedList.setStore(feedStore);
  feedList.setDisplayField("title");
  add(feedList);
}
```

2. Create a new `RpcProxy` to retrieve a list of `Feed` objects using the `loadFeedList` method of the `FeedService` GWT RPC service:

```
  RpcProxy<List<Feed>> proxy = new RpcProxy<List<Feed>>() {
    @Override
    protected void load(Object loadConfig,
```

```
            AsyncCallback<List<Feed>> callback) {
              feedService.loadFeedList(callback);
          }
        };
```

3. Create a new instance of `BeanModelReader` to use to convert the `Feed` objects into `BeanModel` objects:

```
BeanModelReader reader = new BeanModelReader();
```

4. Now create a `ListLoader` that takes the `RpcProxy` and the `BeanModelReader` and uses them to load a list of `BeanModel` representations of the `Feed` objects:

```
ListLoader<ListLoadResult<BeanModel>> loader = new
  BaseListLoader<ListLoadResult<BeanModel>>(
    proxy, reader);
```

5. Define the feed store again, but this time define the `Store` so that it takes the `ListLoader` as a parameter in order to use the loader to populate the store:

```
ListStore<BeanModel> feedStore = new ListStore<BeanModel>(loader);
```

6. Finally, add a call to the `load` method of the `ListLoader` to trigger the loading of the `Store`.

```
loader.load();
```

7. Start the application, and now the feeds that were previously saved will be loaded into the feed list:

What just happened?

We modified the `FeedList ListField` so that it now retrieves the required data from the server using a call to a GWT RPC service.

Pop quiz – right tool for the job

Match the following requirements with the most suitable `DataProxy`, `DataReader`, and `Loader` to achieve the goal:

1. You have XML data on the same server as your application and you want to display it in one list.
2. You have a set of JavaBeans on your server and you want to be able to display them in a paged list.
3. You have a list of Model data and you want to display it in one list.
4. You have JSON data on the same server as your application and you want to display it in a paged list.
5. You have JSON data on a server with a different domain from your application and you want to display it in one list.

	DataProxy	DataReader	Loader
1			
2			
3			
4			
5			

Have a go hero – loading items

In a moment, we are going to load the items of a feed. To do this, we need to first implement the following:

1. A class named `Item` in the `shared.model` package. This class should extend `BaseModel` and needs to provide setters and getters for properties named: category, description, link, and title.
2. A method in the `FeedService` named `loadItems` that takes an URL `String` of a feed and returns a `List` of `Item` objects.
3. A corresponding asynchronous `loadItems` method in the `FeedServiceAsync` class.
4. An implementation of the `loadItems` method in the `FeedServiceImpl` class that makes use of JDOM.

Attempt to implement this functionality using the work we have done with the `Feed` object as a guide. Note that as the `Item` class will extend `BaseModel`, a corresponding `BeanModelMarker` is not required.

Solution:

See the `Item`, `FeedServiceAsync`, and `FeedServiceImpl` classes in the example code.

Grid

GXT contains `Grid` components with many different features. However, at the moment we are just going to look at a basic grid and how to get data into it. When constructing a `Grid` object, it requires both a `ListStore` and a `ColumnModel` to be specified.

ColumnConfig

A `ColumnConfig` object defines a column that a `Grid` will display. It specifies the data that the columns will use and how it should be rendered. These are then collected into a list and used in the constructor of a `ColumnModel` object, which acts as a container for the `ColumnConfigs` and can in turn be used in the constructor of a `Grid`.

Grid Example

We are now going to return to our example application and add a `Grid` that will display RSS items from an RSS feed.

Time for action – creating the ItemGrid

1. Create a new class named `ItemGrid` in a new package named `client.grids`.
2. The new class should extend the `LayoutContainer` class and override the `onRender` method:

```
public class ItemGrid extends LayoutContainer {

  @Override
  protected void onRender(Element parent, int index) {
    super.onRender(parent, index);
    ..
  }
}
```

3. Create a constructor for the class that sets the layout of the underlying `LayoutContainer` to be a `FitLayout`:

```
public ItemGrid() {
  setLayout(new FitLayout());
}
```

4. In the `onRender` method, define the `ColumnConfigs` for the `Grid` and add them to a list. One column should use the `title` field of the `Feed` object and the other the `description` field:

```
final List<ColumnConfig> columns = new ArrayList<ColumnConfig>();
columns.add(new ColumnConfig("title", "Title", 200));
columns.add(new ColumnConfig("description", "Description", 200));
```

5. Create a `ColumnModel` passing the list of `ColumnConfig` objects to the constructor:

```
final ColumnModel columnModel = new ColumnModel(columns);
```

6. We now need to define test data to load. Fortunately, there is an example RSS file available with the specification. Create a constant to store the String for the URL:

```
final String TEST_DATA_FILE =
  "http://cyber.law.harvard.edu/rss/examples/rss2sample.xml";
```

7. Retrieve the feed service from the `Registry`:

```
final FeedServiceAsync feedService = Registry
  .get(RSSReaderConstants.FEED_SERVICE);
```

8. Create an `RpcProxy` that uses the `loadItems` method of the `FeedService` to retrieve the `Item` objects for the feed at the URL defined in the `TEST_DATA_FILE` constant:

```
RpcProxy<List<Item>> proxy = new RpcProxy<List<Item>>() {
  @Override
  protected void load(Object loadConfig,
    AsyncCallback<List<Item>> callback) {
    feedService.loadItems(TEST_DATA_FILE, callback);
  }
};
```

9. Create a `BaseListLoader` that uses the `RpcProxy`. Note that as `Item` extends `BaseModel`, a `DataReader` is not needed:

```
ListLoader<ListLoadResult<Item>> loader = new
  BaseListLoader<ListLoadResult<Item>>(
    proxy);
```

10. Now create a `ListStore` for the `Item` objects:

```
ListStore<ModelData> itemStore = new ListStore<ModelData>(loader);
```

11. We can now create a grid using the `Store` and the `ColumnModel`. Also, set the auto expand column to description so that the `description` column expands to fill the available space:

```
Grid<ModelData> grid = new Grid<ModelData>(itemStore,
  columnModel);
grid.setBorders(true);
grid.setAutoExpandColumn("description");
```

12. Call the `load` method of the `ListLoader` to load the `Item` objects into the `Store` of the `Grid`:

```
loader.load();
```

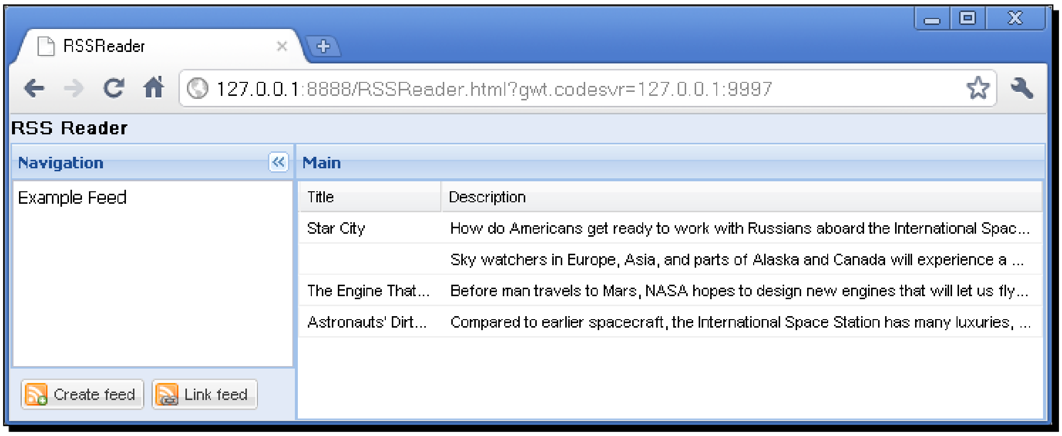
13. Add the `Grid` to the underlying `LayoutContainer`:

```
add(grid);
```

14. Finally, we can set the layout of `RssMainPanel` to `FitLayout` and add a new instance of `ItemGrid` in the constructor.

```
  public RssMainPanel()
  {
    setHeading("Main");
    setLayout(new FitLayout());
    add(new ItemGrid());
  }
```

15. Now start the application and it will have a grid populated with the sample RSS file's data:



What just happened?

We created a `Grid`, which makes use of a `Store` that is populated using `Item` objects retrieved from the server. We saw how to use a `DataProxy`, `DataReader`, and `Loader` to retrieve the `Item` objects, and load them into the `ListStore`.

GridCellRenderer

At the moment, in our sample application's item grid, we are displaying a single field in a column. However, if we want to combine fields or make them more than just plain text, we can use the `GridCellRenderer`. This enables us to specify the generation of HTML to render in a cell rather than the plain text value of a field.

Once defined, we can apply a `GridCellRenderer` to an entire column by using the `setRenderer` method of `ColumnConfig`.

We are now going to use a `GridCellRenderer` in our application's `ItemGrid`. Instead of the title and description appearing in different columns, we are going to use the `GridCellRenderer` to display the title above the description in the same column.

`GridCellRenderer` objects must include a render method, which returns an HTML string.

Time for action – using a GridCellRenderer

1. In the `onRender` method of the `ItemGrid` class, create a new `GridCellRenderer` named `itemsRenderer` with a `render` method:

```
GridCellRenderer<ModelData> itemsRenderer = new
  GridCellRenderer<ModelData>() {
  public String render(ModelData model, String property,
    ColumnData config, int rowIndex, int colIndex,
      ListStore<ModelData> store, Grid<ModelData> grid)
  {
  }
};
```

2. Implement the `render` method so that it retrieves the title and description fields from the model and combines them in an HTML string, which is returned from the method:

```
GridCellRenderer<ModelData> itemsRenderer = new
   GridCellRenderer<ModelData>() {
   public String render(ModelData model, String property,
     ColumnData config, int rowIndex, int colIndex,
     ListStore<ModelData> store, Grid<ModelData> grid)
   {
    String title = model.get("title");
    String description = model.get("description");
    return "<b>" + title + "</b><br/>" + description;
   }
  };
```

3. Create a new `ColumnConfig` with the ID of `items`, the header of `Items`, and set the renderer to be the `itemsRenderer` we created in the previous step:

```
ColumnConfig column = new ColumnConfig();
column.setId("items");
column.setRenderer(itemsRenderer);
column.setHeader("Items");
```

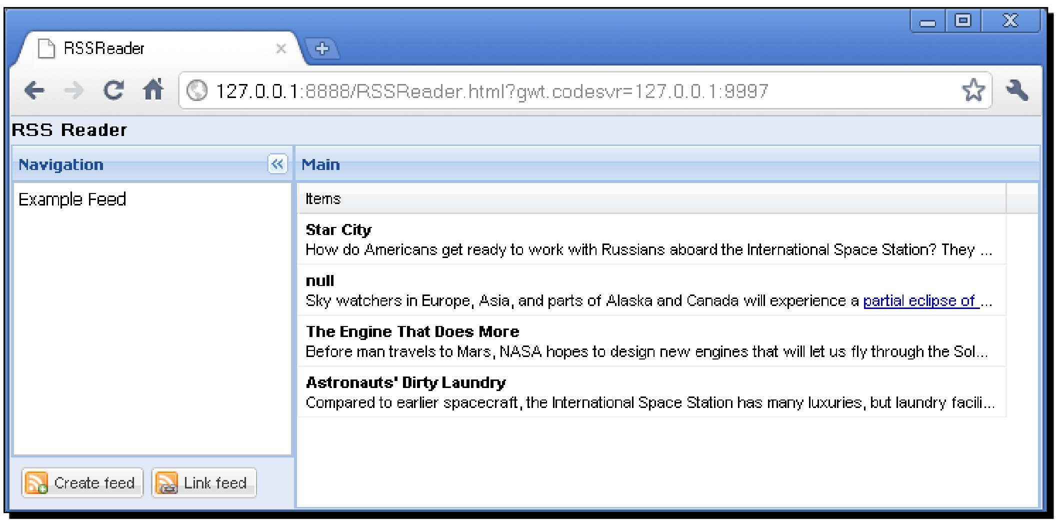
4. Add the `items` column to the list of columns in place of the previous title and description columns:

```
columns.add(column);
final ColumnModel columnModel = new ColumnModel(columns);
```

5. Change the grid to auto expand the `items` column instead of the now non-existent description column:

```
grid.setAutoExpandColumn("items");
```

6. Start the application and notice how the `GridCellRenderer` has rendered the fields in a single column:



What just happened?

We used a `GridCellRenderer` to create an HTML string to display two fields in the same column with formatting instead of just plain text.

Summary

In this chapter, we have introduced GXT's data-backed components. We have made use of a `Store` to cache data locally. We linked a `ListField` to a `Store` to show how the components' values could automatically be updated when data in a `Store` changed.

We then added to the server-side of our example application by providing a service that can persist and retrieve remote data.

We went on to use the service to retrieve remote `ModelData` and used that data to populate a `Grid`. We then formatted the `Grid` using a `GridCellRenderer`.

In the next chapter, we will look at some more advanced grids and also look at the useful tree-based components.

5

More Components

This chapter builds on the previous chapter by taking data-based controls further. We will look at Tree controls and show how they can improve on a ListField for organizing data and how the same tree concept can be applied to a Grid. We will then cover some of the more advanced functions available in grid. Finally, we will explore menus and toolbars.

In this chapter, we will specifically cover the following topics:

- Trees
- `BaseTreeModel`
  - `TreeStore`
  - `TreePanel`
  - `TreeGrid`
  - `TreeGridCellRenderer`
- Advanced grid features
  - Column grouping
    - `HeaderGroupConfig`
  - Aggregation rows
    - `AggregationRowConfig`
    - `SummaryType`

        - Paging
            - `PagingListResult`
            - `PagingLoadConfig`
                - `PagingModelMemoryProxy`
                - `PagingLoader`
                - `PagingToolBar`
- `ImageBundle`
- Toolbars and menus
    - `Menu`
    - `MenuItem`
    - `CheckMenuItem`
    - `MenuBar`
    - `MenuBarItem`
    - `MenuEvent`
    - `ToolBar`
    - `Status`

Trees

In the previous chapter, we worked with components that made use of lists of data. Now we are going to look at the components that work with trees of data instead.

Working with trees in GXT is similar to working with lists. The difference is that there are special tree versions of the `ModelData`—`Store`, `DataReader`, and `Loader` we used in the previous chapter.

BaseTreeModel class

`BaseTreeModel` extends the `BaseModel` we used in the previous chapter by implementing the `TreeModel` interface to add tree features. Essentially, this involves adding methods for managing parent and child relationships.

In order to be able to use `ModelData` in a `TreePanel` or `TreeGrid`, the objects must extend `BaseTreeModel` rather than just `BaseModel`.

In our example application, we are going to show the items from a feed in a categorized tree. To be able to do this, we need to create a `Category` class that extends `BaseTreeModel`.

Time for action – creating a BaseTreeModel

1. Create a new class in `shared.model` named `Category`. This will hold a category structure in a tree, so it needs to extend `BaseTreeModel`.

```
public class Category extends BaseTreeModel {}
```

2. Create a constructor that takes a title and assigns a sequential ID.

```
public class Category extends BaseTreeModel {

  private static int ID = 0;

  public Category(String title) {
    set("id", ID++);
    set("title", title);
  }
}
```

3. Add a zero-arguments constructor, as we will be passing these objects over GWT RPC.

```
  public Category() {
    set("id", ID++);
  }
```

4. Add getters for ID and the title properties.

```
public class Category extends BaseTreeModel {

  private static int ID = 0;

  public Category() {
    set("id", ID++);
  }

  public Category(String title) {
    set("id", ID++);
    set("title", title);
  }

  public Integer getId() {
    return (Integer) get("id");
  }

  public String getTitle() {
    return (String) get("title");
  }
}
```

What just happened?

We created a `BaseTreeModel` to store a category structure for organizing our feed items.

We now need to change our `FeedService` so that it delivers only the `Item` objects in a given category.

Time for action – providing categorized items

1. In the `FeedService` interface, define a `loadCategorisedItems` method. This should take a feed URL `String` and a `Category` as arguments and return a `List` of `Item` objects.

```
List<ModelData> loadCategorisedItems(String feedUrl, Category
category);
```

2. Create the corresponding asynchronous method in the `FeedServiceAsync` interface.

```
void loadCategorisedItems(String feedUrl, Category category,
      AsyncCallback<List<ModelData>> callback);
```

3. In the `FeedServiceImpl` class, implement the `loadCategorisedItems` method as follows. This method will return a `List` of `Item` objects in a category if a `Category` object is provided, otherwise it will return a `List` of `Category` objects.

```
@Override
public List<ModelData> loadCategorisedItems(String feedUrl,
    Category category) {
  List<Item> items = loadItems(feedUrl);
  Map<String, List<Item>> categorisedItems = new HashMap<String,
List<Item>>();
  for (Item item : items) {
    String itemCategoryStr = item.getCategory();
    if (itemCategoryStr==null) {
      itemCategoryStr = "Uncategorised";
    }
    List<Item> categoryItems = categorisedItems.
get(itemCategoryStr);
    if (categoryItems == null) {
      categoryItems = new ArrayList<Item>();
    }
    categoryItems.add(item);
    categorisedItems.put(itemCategoryStr, categoryItems);
  }
  if (category == null) {
```

```
        List<ModelData> categoryList = new ArrayList<ModelData>();
        for (String key: categorisedItems.keySet())
        {
          categoryList.add(new Category(key));
        }
        return categoryList;
      }
      else
      {
        return new ArrayList<ModelData>(categorisedItems.get(category.
    getTitle()));
      }
    }
```

What just happened?

We created a method in the feed service that returns either a list of `Category` objects or a `List` of `Item` objects contained within a specified `Category`.

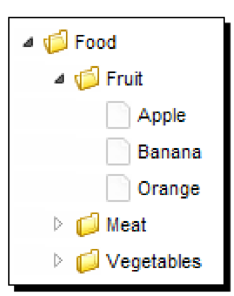
TreeStore class

`TreeStore` is another implementation of the `Store` class we covered in the previous chapter. The difference is instead of storing the data as a list, as with a `ListStore`, the `TreeStore` stores data in a hierarchy.

Although we can add `TreeModel` objects to a `TreeStore`, it does not use the parent and child relationships of the `TreeModel`, but instead manages the relationships internally. When we add a `TreeModel` to a tree store using the `add` method, there is a second Boolean parameter that allows you to specify whether the child object of the `TreeModel` should also be added.

TreePanel class

`TreePanel` is the actual visual tree component. Using it is not that different from a `ListField`, as the tree parts are mostly handled for you.



When we create a new `TreePanel`, a `TreeStore` must be provided as a parameter to the constructor. We then need to define the name of the store's property to use as the label for nodes using the `setDisplayProperty` method.

By default a folder icon is used for nodes that have children. If we would like the leaf nodes (the nodes without children) to have an icon too, we can set this using the `setLeafIcon` method, which takes a GWT `AbstractImagePrototype` as an argument.

ImageBundle class

`Tree` components make use of GWT's `ImageBundle` features to preload the icons that are used for nodes in the tree. We will want to use icons in the tree components for our example application, so we need to define an `ImageBundle`. Although `ImageBundle` is a part of GWT as opposed to GXT, it is deprecated in current versions of GWT where it has been replaced by `ClientBundle`.

Time for action – using an ImageBundle

1. Create a new package in the client named `resources` and in that package create a new interface named `Icons` that extends `ImageBundle`. You will get a depreciation warning for `ImageBundle` that you will probably want to suppress.

```
@SuppressWarnings("deprecation")
public interface Icons extends ImageBundle {
```

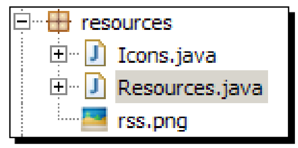
2. Each image that is added to an `ImageBundle` should have a method that returns an `AbstractImagePrototype` and no parameter. The actual image file to be loaded is defined in a `@Resource` annotation.

```
@SuppressWarnings("deprecation")
public interface Icons extends ImageBundle {

    @Resource("rss.png")
    AbstractImagePrototype rss();

}
```

3. Now create a class named `Resources` that will be used to instantiate the `Icons` interface and make it available as a static field.

```
public class Resources {

  public static final Icons ICONS = GWT.create(Icons.class);
}
```

4. Finally, place the actual image file to be loaded in the package along with the classes so that the package looks like this:

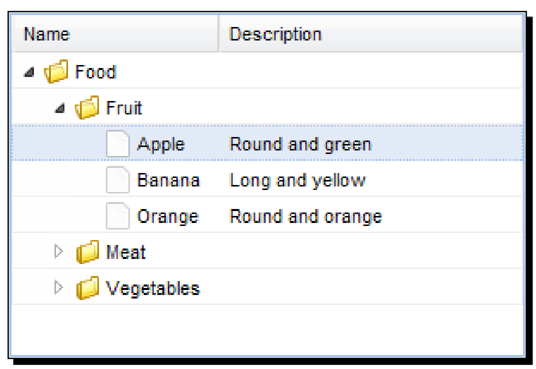


What just happened?

We created an `ImageBundle` that will allow us to use an icon within tree components.

TreeGrid class

`TreeGrid` is where trees and grids come together. Entries can have multiple columns like other grids, but a tree is also used to categorize the entries refer to the following screenshot.



As with the `TreePanel`, the `Store` used with a `TreeGrid` is a `TreeStore` and the model objects that are contained in the store need to extend `BaseTreeModel`.

In our example application, we are going to use this to group RSS items by their categories.

TreeGridCellRenderer class

The `TreeGridCellRenderer` is an implementation of the `GridCellRenderer` that we encountered in the previous chapter. Its function is to render a tree into a column.

We are now going to use a `TreeGridCellRenderer` of the first column of the grid, but it can be used in any column just like any other `GridCellRenderer`.

Time for action – replacing the Feed List with a Feed Tree

1. Create a new class named `ItemCategoryGrid` in the `client.grids` package. The class should extend `LayoutContainer` and in the constructor set the layout to be `FitLayout`.

```
public class ItemCategoryGrid extends LayoutContainer {

  public ItemCategoryGrid() {
    setLayout(new FitLayout());
  }
}
```

2. Override the `onRender` method and retrieve the `FeedService` from the `Registry`.

```
@Override
  protected void onRender(Element parent, int index) {
    super.onRender(parent, index);

    final FeedServiceAsync feedService = (FeedServiceAsync)
Registry
        .get(RSSReaderConstants.FEED_SERVICE);
}
```

3. Create an `RpcProxy` that uses the `loadCategorisedItems` method of the `FeedService`. The `loadConfig` should be cast to be a `Category` object for sending to the method.

```
final String TEST_DATA_FILE = http://feeds.feedburner.com/
extblog

RpcProxy<List<ModelData>> proxy = new RpcProxy<List<ModelData>>()
{
  @Override
  protected void load(Object loadConfig,
          AsyncCallback<List<ModelData>> callback) {
    feedService.loadCategorisedItems(TEST_DATA_FILE, (Category)
loadConfig, callback);
  }
};
```

4. Create a `BaseTreeLoader` and override the `hasChildren` method so that it returns `true` if the `ModelData` passed to it is an instance of the `Category` `BaseModel`.

```
final TreeLoader<ModelData> loader = new BaseTreeLoader<ModelData>
(proxy)
{
  @Override
  public boolean hasChildren(ModelData parent){
    if (parent instanceof Category)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
};
```

5. Create a `TreeStore` that uses the `TreeLoader`.

```
final TreeStore<ModelData> feedStore = new
TreeStore<ModelData>(loader);
```

6. Create a `ColumnConfig` for the title that uses the `TreeGridCellRenderer` to render the column as a tree.

```
ColumnConfig title = new ColumnConfig("title", "Title", 200);
    title.setRenderer(new TreeGridCellRenderer<ModelData>());
```

7. Define a description `ColumnConfig` and add it together with the title `ColumnConfig` to a new instance of `ColumnModel`.

```
ColumnConfig description = new ColumnConfig("description",
        "Description", 200);

    ColumnModel columnModel = new ColumnModel(Arrays.
      asList(title));
```

8. Define a `TreeGrid` that uses the feed store and `ColumnModel`, auto expand the description model, and set the icon to use as the leaf node to the RSS icon we defined earlier.

```
TreeGrid<ModelData> treeGrid = new TreeGrid<ModelData>(feedStore,
        columnModel);
treeGrid.setBorders(true);
treeGrid.setAutoExpandColumn("title");
treeGrid.getStyle().setLeafIcon(Resources.ICONS.rss());
```

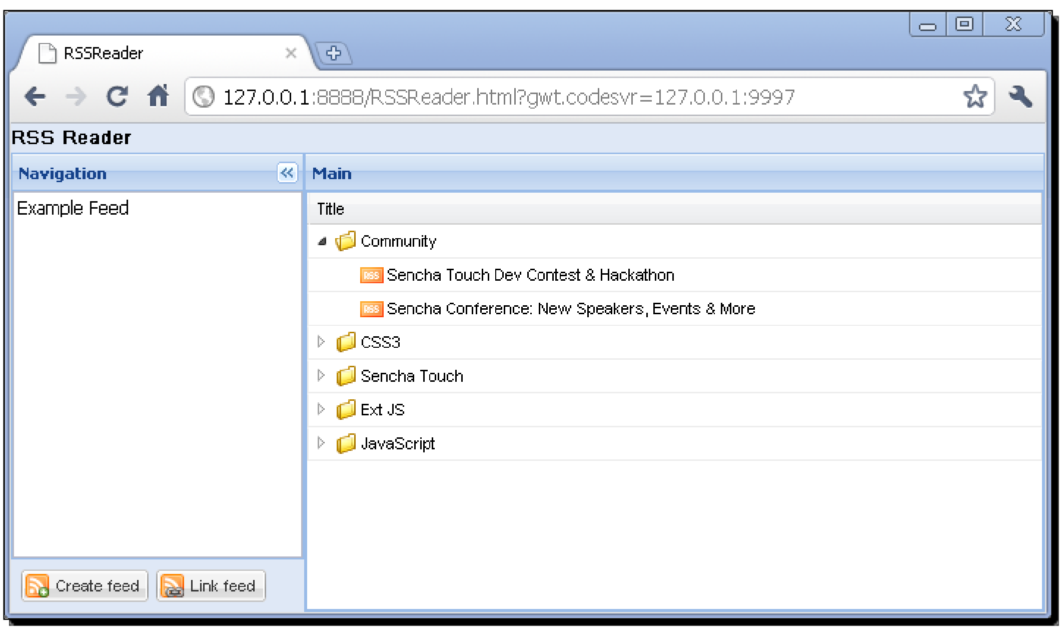
9. Call the `load` method of the `TreeLoader` and add `TreeGrid` to the underlying `LayoutContainer`.

```
loader.load();
add(treeGrid);
```

10. In the `RssMainPanel` class change the Grid that is added from `ItemGrid` to `ItemCategoryGrid`.

```
public RssMainPanel() {
        setHeading("Main");
        setLayout(new FitLayout());
        add(new ItemCategoryGrid());
}
```

11. Now start the application and you will see the `TreeGrid` with the items organized by category.



What just happened?

We replaced the `Grid` with a `TreeGrid`, which allows us to categorize feeds using a `TreeGridCellRenderer`.

Advanced grid features

In the previous chapter, we introduced grid controls. However, we only looked at the most basic of `Grid` features. Grids are very powerful and there are many options for expanding and customizing them. Let's look at some of the requirements that we may come across and how grid features can help.

HeaderGroupConfig class

Suppose we wanted to compare the population of Eastern European countries in the year 2000 with the population in 1950 in a grid, we could display them like this:

Country	1950 Population (000's)	2000 Population (000's)
Belarus	7745	10054
Bulgaria	7251	8006
Czech Republic	8925	10224
Hungry	9338	10215

Or we could group the columns like this:

Country	Population (000's)	
	1950	2000
Belarus	7745	10054
Bulgaria	7251	8006
Czech Republic	8925	10224
Hungry	9338	10215

We can do the same in a GXT `Grid`.

To create the columns, we need to perform the following steps:

1. Create `ColumnConfig` for each column.
2. Add each one to a list.
3. Use that list to create a `ColumnModel`.

```
final List<ColumnConfig> columns = new ArrayList<ColumnConfig>();
ColumnConfig column = new ColumnConfig("countryName",
"Country",100);
columns.add(column);
column = new ColumnConfig("population1950", "1950 Population
(000's)",130);
```

```
    columns.add(column);
    column = new ColumnConfig("population2000", "2000 Population
    (000's)",130);
    columns.add(column);
    final ColumnModel columnModel = new ColumnModel(columns);
```

4. This will produce columns like these:

Country	1950 Population (000's)	2000 Population (000's)
Belarus	7745	10054
Bulgaria	7251	8006

To group the columns, we need to simply add a new object called a `HeaderGroupConfig`.

```
HeaderGroupConfig headerGroupConfig = new HeaderGroupConfig(
        "Population (000's)", 1, 2);
```

The arguments in creating the `HeaderGroupConfig` are the title of the grouped column followed by the number of rows to merge and the number of columns to merge respectively.

Now we can use the `addHeaderGroup` method of the `ColumnModel` to use this `HeaderGroupConfig`, specifying the row and column to apply it to.

```
columnModel.addHeaderGroup(0, 1, headerGroupConfig););
```

The columns will now be grouped like this:

Country	Population (000's)	
	1950	2000
Belarus	7745	10054
Bulgaria	7251	8006
Czech Republic	8925	10224

AggregationRowConfig class

Another thing we may want to do is add summary rows to a grid. We can create aggregation rows using an `AggregationRowConfig` to create summary data.

Aggregation rows can summarize data in the following ways, defined by `SummaryType` constants.

SummaryType Constant	Description
`SummaryType.SUM`	Total of the values in the column
`SummaryType.AVG`	Average of the values in the column
`SummaryType.MIN`	Minimum value in a column
`SummaryType.MAX`	Maximum value in a column
`SummaryType.COUNT`	Number of values

Here is an example of an aggregation row that is being used to provide totals for the population columns:

Country	Population (000's)	
	1950	2000
Belarus	7745	10054
Bulgaria	7251	8006
Czech Republic	8925	10224
Hungary	9338	10215
Poland	24824	38433
Republic of Moldova	2341	4100
Romania	16311	22138
Russia	102702	146670
Slovakia	3463	5379
Ukraine	37298	48870
Total	220,198	304,089

To produce this, we would need to do the following:

- Create a new `AggregationRowConfig`:

  ```
  AggregationRowConfig<Statistic>> totals = new AggregationRowConfig
  <Statistic>();
  ```

- Create a label for the row using the `setHtml` method:

  ```
  totals.setHtml("countryName", "Total");
  ```

- For each column, if we want to display the total population,we need to set a `SummaryType`.

```
totals.setSummaryType("population1950", SummaryType.SUM);
totals.setSummaryType("population2000", SummaryType.SUM);
```

- For each column we also need to define a `NumberFormat` in order for the total to be displayed. Alternatively, we could also use an `AggregationRenderer`. We must use one of these or else the total will be blank:

```
totals.setSummaryFormat("population1950", NumberFormat.
getDecimalFormat());
totals.setSummaryFormat("population2000", NumberFormat.
getDecimalFormat());
```

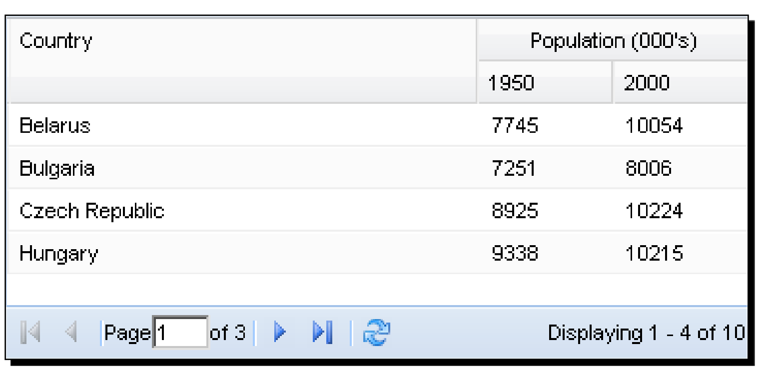
- Then add the `AggregationRowConfig` to the column model.

```
columnModel.addAggregationRow(totals);
```

Paging

Paging is another useful feature of GXT. This allows us to present grid data in multiple pages rather than a single long list. This allows for quicker load times and a more responsive application. GXT supports remote and local paging. **Remote paging** is when the client makes multiple requests to a custom backend to retrieve a subset of data items rather than all items. It is beyond the scope of this book. However, we can introduce the feature by looking at local paging, which is where we load all the data into a store but display in multiple pages in the grid.

For example, we could decide that we would like to display our data in pages of four items like this:



PagingLoadResult interface

To use paging, the data supplied must be in the form of a `PagingLoadResult`. This is an extension to the `ListLoadResult` interface that provides additional functions for getting and setting both the total number of available items *and* the offset from which the page of items is being returned.

PagingLoadConfig class

The `PagingLoadConfig` class is used to encapsulate the parameters required for retrieving a page of information, specifically the offset, to start returning data from and the limit, the number of items to return.

We will now create a new method in the feed service of the example application that returns a `PagingLoadResult` of `Item` objects.

Time for action – providing paged data

1. In the `FeedService` interface, define a second `loadItems` method. This one should take a `PagingLoadConfig` object as well as the feed URL `String` as arguments and return a `PagingLoadResult` of `Item` objects.

```
PagingLoadResult<Item> loadItems(String feedUrl, final
PagingLoadConfig config);
```

2. Create the corresponding asynchronous method in the `FeedServiceAsync` interface.

```
void loadItems(String feedUrl, PagingLoadConfig config,
      AsyncCallback<PagingLoadResult<Item>> callback);
```

3. In the `FeedServiceImpl` class, create a new private method named `getPagingLoadResult` that take a `List` of `Item` objects and a `PagingLoadConfig` as parameters. The purpose of this function is to take the full list of `Item` objects and return the page requested in the `PagingLoadConfig`.

```
private PagingLoadResult<Item> getPagingLoadResult(List<Item>
items, PagingLoadConfig config) {}
```

4. Create a new `List` of `Item` objects to use to return, retrieve the offset from the `PagingLoadConfig` and the limit from the size of the full list of items.

```
List<Item> pageItems = new ArrayList<Item>();
int offset = config.getOffset();
int limit = items.size();
```

5. Check that the end point for the page specified in the `PagingLoadConfig` is not greater than the number of `Item` objects available and if so set the limit to the lower number.

```
if (config.getLimit() > 0) {
  limit = Math.min(offset + config.getLimit(), limit);
}
```

6. Add the subset of the `Item` objects required to the list of `Item` objects in order to build the page.

```
for (int i = config.getOffset(); i < limit; i++) {
      pageItems.add(items.get(i));
}
```

7. Create a new `BasePagingLoadResult` of `Item` object to return, specifying the `List` of `Item` objects for the page and the offset and the total number of `Item` objects available.

```
return new BasePagingLoadResult<Item>(pageItems, offset,
        items.size());
```

8. Finally, implement the new `loadItems` method such that it loads all the `Item` objects and then retrieves and the correct `PagingLoadResult` form from the `getPagingLoadResult` method.

```
@Override
public PagingLoadResult<Item> loadItems(String feedUrl,
    PagingLoadConfig config) {
  List<Item> items = loadItems(feedUrl);
  return getPagingLoadResult(items, config);
}
```

What just happened?

We implement a method in the `FeedService` that retrieves a `PagingLoadResult` of `Item` objects for use with paging components.

When using paging, a paging implementation of both a `DataProxy` and a `Loader` must be used to move the correct subset of the data into the store.

PagingModelMemoryProxy class

`PagingModelMemoryProxy` is a special `DataProxy` that takes a set of data, specified in the constructor and is used to hold the data ready to be loaded with a `PagingLoader`.

PagingLoader class

`PagingLoader` takes the data provided by `PagingModelMemoryProxy` specified in the constructor and loads the correct subset into the store. It is created like this:

```
PagingLoader<PagingLoadResult<ModelData>> loader = new BasePagingLoade
r<PagingLoadResult<ModelData>>(proxy);
```

To kick things off, we need to load the initial data (in this case `4`) items starting from item `0`.

```
loader.load(0, 4);
```

A normal `ListStore` can then be used with the loader to store the cache of the data for the current page of the `Grid`.

PagingToolBar class

`PagingToolBar` is a predefined `ToolBar` that provides the controls for moving forward and backward through the pages of a `Grid`. It also shows the current range of items that are being displayed and also the total number of items.

As the `PagingToolBar` controls the data that a `PagingLoader` loads, it needs to be bound to the `PagingLoader` and added to the underlying panel.

```
toolBar.bind(loader);
add(toolBar);
```

To allow for paging, the `FeedService` needs to return a subset of the available `Item` objects based on a `PagingLoadConfig`.

We will now create a paging grid for the example application.

Time for action – creating a paging grid

1. Take a copy of the `ItemGrid` class and rename the copied class to `ItemPagingGrid`.

2. Currently, there is an `RpcProxy` that calls the non-paging `loadItems` method to retrieve a `List` of `Item` objects. Replace this with a call to the paging `loadItems` method that returns a `PagingListResult` of `Item` objects.

```
RpcProxy<PagingLoadResult<Item>> proxy = new RpcProxy<PagingLoadRe
sult<Item>>() {
  @Override
```

```
  protected void load(Object loadConfig,
      AsyncCallback<PagingLoadResult<Item>> callback) {
    feedService.loadItems(TEST_DATA_FILE, (PagingLoadConfig)
loadConfig, callback);
  }
};
```

3. Replace the `ListLoader` and the `ListLoadResult` with the paging equivalents.

```
PagingLoader<PagingLoadResult<Item>> loader = new BasePagingLoader
<PagingLoadResult<Item>>(
        proxy);
```

4. Define a constant for the page size of `10`.

```
private static final int PAGE_SIZE = 10;
```

5. Create a new `PagingToolBar` using the `PAGE_SIZE` constant as the page size and bind the toolbar to the loader.

```
final PagingToolBar toolBar = new PagingToolBar(PAGE_SIZE);
    toolBar.bind(loader);
```

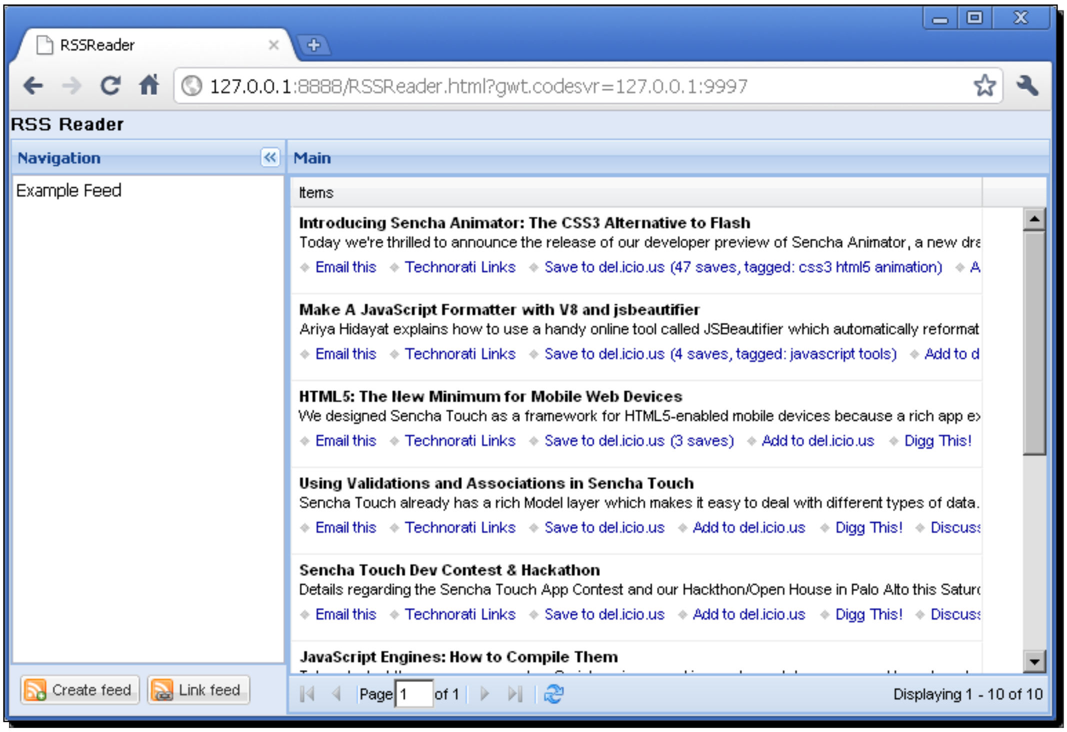
6. Instead of adding the `Grid` directly to the underlying `LayoutContainer`, create a new `ContentPanel` and add the `Grid` and the `PagingToolBar` to it. Then add this `ContentPanel` to the underlying `LayoutContainer`.

```
ContentPanel panel = new ContentPanel();
panel.setLayout(new FitLayout());
panel.add(grid);
panel.setHeaderVisible(false);
panel.setBottomComponent(toolBar);
add(panel);
```

7. In the `RssMainPanel` class, change to `ItemCategoryGrid` to the new `ItemPagingGrid`.

```
add(new ItemPagingGrid());
```

8. Start the application and the item grid will now be paged.



What just happened?

We created a version of our `ItemGrid` that supports paging. This allows us to deal with a larger list of `Item` objects in pages. This reduces the initial load time for feed items and reduces the memory requirements of the application in the web browser.

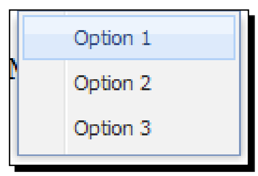
Menus and toolbars

GXT provides the types of toolbars and menus that users have come to expect in desktop applications.

Menu component

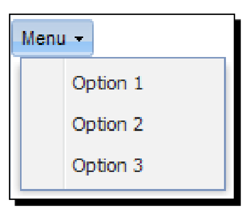
`Menu` is a very flexible component that can be displayed as a context menu in relation to other widgets, using the `show` method.

```
Menu contextMenu = new Menu();
contextMenu.add(new MenuItem("Option 1"));
contextMenu.add(new MenuItem("Option 2"));
contextMenu.add(new MenuItem("Option 3"));
Label label = new Label("Menu appears here");
contextMenu.show(label);
```



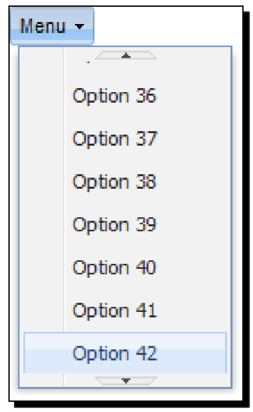
A `Menu` can be added to a `Button` to provide an additional option:

```
Menu contextMenu = new Menu();
contextMenu.add(new MenuItem("Option 1"));
contextMenu.add(new MenuItem("Option 2"));
contextMenu.add(new MenuItem("Option 3"));
Button button = new Button("Menu");
button.setMenu(contextMenu);
```



When a menu has many items, a maximum height can be specified using the `setMaxHeight` method and the menu becomes scrollable.

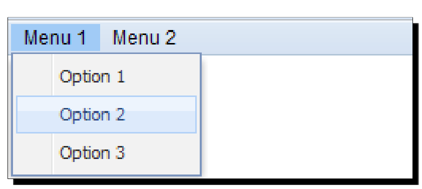
```
Menu contextMenu = new Menu();
for (int i = 1; i < 100; i++) {
  contextMenu.add(new MenuItem("Option " + i));
}
contextMenu.setMaxHeight(200);

Button button = new Button("Menu");
button.setMenu(contextMenu);
```



MenuBar component

`Menu` components can be collected together in a `MenuBar`, which again is very familiar to the users of desktop applications. Here we need to wrap a `Menu` in a `MenuBarItem` before adding it to a `MenuBar`.

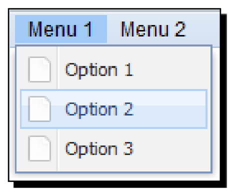
```
Menu menu1 = new Menu();
menu1.add(new MenuItem("Option 1"));
menu1.add(new MenuItem("Option 2"));
menu1.add(new MenuItem("Option 3"));

Menu menu2 = new Menu();
menu2.add(new MenuItem("Option 4"));
menu2.add(new MenuItem("Option 5"));
menu2.add(new MenuItem("Option 6"));

MenuBar menuBar = new MenuBar();
menuBar.add(new MenuBarItem("Menu 1", menu1));
menuBar.add(new MenuBarItem("Menu 2", menu2));

viewport.add(menuBar);
```



MenuItem component

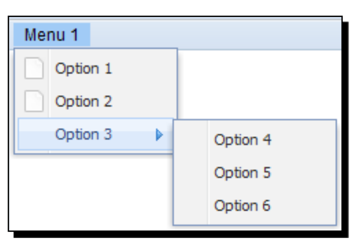
Menus act as a container for `MenuItem` objects that perform the actual functions of the `Menu`. They can have text, an icon, or both. The icon can either be set from a CSS style or from an `ImageBundle`.

```
Menu menu1 = new Menu();
menu1.add(new MenuItem("Option 1",Resources.ICONS.page()));
menu1.add(new MenuItem("Option 2",Resources.ICONS.page()));
menu1.add(new MenuItem("Option 3",Resources.ICONS.page()));
```



A `Menu` can be set as a submenu of a `MenuItem` using the `setSubMenu` method to produce nested menus.

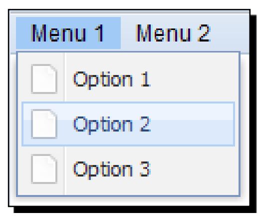
```
Menu menu1 = new Menu();
menu1.add(new MenuItem("Option 1",Resources.ICONS.page()));
menu1.add(new MenuItem("Option 2",Resources.ICONS.page()));

Menu menu2 = new Menu();
menu2.add(new MenuItem("Option 4"));
menu2.add(new MenuItem("Option 5"));
menu2.add(new MenuItem("Option 6"));

MenuItem miOption3 = new MenuItem("Option 3");
miOption3.setSubMenu(menu2);
menu1.add(miOption3);
```



CheckMenuItem component

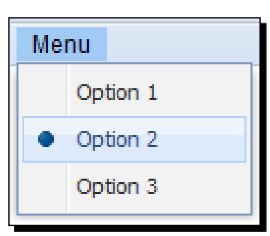
`CheckMenuItem` components extend `MenuItem` components to provide checkable menu items.

```
Menu menu = new Menu();
menu.add(new CheckMenuItem("Option 1"));
menu.add(new CheckMenuItem("Option 2"));
menu.add(new CheckMenuItem("Option 3"));

MenuBar menuBar = new MenuBar();
menuBar.add(new MenuBarItem("Menu", menu));
```



`CheckMenuItem` components also can be grouped together to provide a radio button style group, where only one of the items in the group can be selected at one time. This is achieved by defining a group for each `CheckMenuItem`.

```
Menu menu = new Menu();
CheckMenuItem checkMenuItem1 = new CheckMenuItem("Option 1");
CheckMenuItem checkMenuItem2 = new CheckMenuItem("Option 2");
CheckMenuItem checkMenuItem3 = new CheckMenuItem("Option 3");
checkMenuItem1.setGroup("options");
checkMenuItem2.setGroup("options");
checkMenuItem3.setGroup("options");
menu.add(checkMenuItem1);
menu.add(checkMenuItem2);
menu.add(checkMenuItem3);

MenuBar menuBar = new MenuBar();
menuBar.add(new MenuBarItem("Menu", menu));
```



MenuEvent class

`MenuEvent` is the event that is created when a `MenuItem` is selected. It is the equivalent of the `ButtonEvent`, which is triggered when a `Button` is pressed.

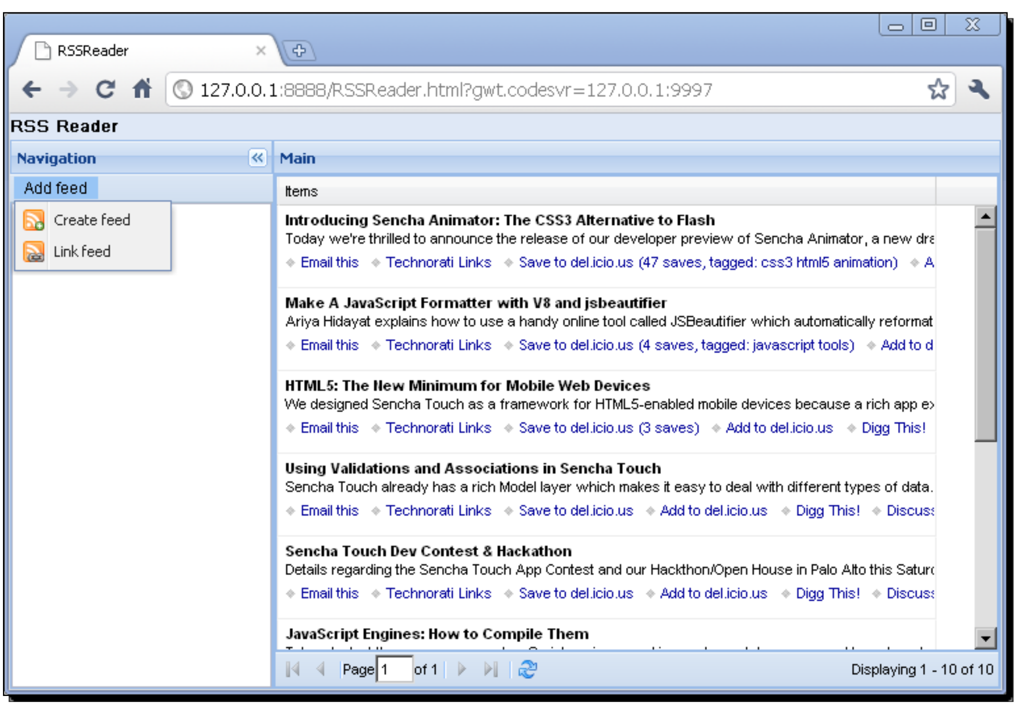
`MenuItem` objects can have `SelectionListener` objects assigned to respond to a `MenuEvent` again in the same way as `Button` components.

Here is what a `SelectionListener` looks like when it is added to a `MenuItem`:

```
MenuItem menuItem = new MenuItem("Option1");
menuItem.addSelectionListener(new SelectionListener<MenuEvent>(){
  @Override
  public void componentSelected(MenuEvent ce) {
    //Action goes here
  }
});
```

Have a go hero – add a menu

Currently, we have two buttons in the `RssNavigationPanel`—**Create feed** and **Link feed**. Replace these buttons with a `MenuBar` that could perform the same functions and add the `MenuBar` to the `RssNavigationPanel` so that it looks like the next screenshot:



Solution:

```
    public RssNavigationPanel() {
      setHeading("Navigation");
      setLayout(new FitLayout());

      Menu menu = new Menu();

      final MenuItem miCreateFeed = new MenuItem("Create feed");
      miCreateFeed.setIconStyle("create-feed");

      ToolTipConfig createNewToolTipConfig = new ToolTipConfig();
      createNewToolTipConfig.setTitle("Create a new RSS feed");
      createNewToolTipConfig
          .setText("Creates a new RSS feed");
      miCreateFeed.setToolTip(createNewToolTipConfig);
      miCreateFeed.addSelectionListener(new
  SelectionListener<MenuEvent>() {
        @Override
        public void componentSelected(MenuEvent me) {
          createNewFeedWindow();
        }
      });
      menu.add(miCreateFeed);

      final MenuItem miLinkFeed = new MenuItem("Link feed");
      miLinkFeed.setIconStyle("link-feed");
      menu.add(miLinkFeed);

      ToolTipConfig linkFeedToolTipConfig = new ToolTipConfig();
      linkFeedToolTipConfig.setTitle("Link to existing RSS feed");
      linkFeedToolTipConfig
          .setText("Allows you to enter the URL of an existing RSS feed
  you would like to link to");
      miLinkFeed.setToolTip(linkFeedToolTipConfig);

      final LinkFeedPopup addFeedPopup = new LinkFeedPopup();
      addFeedPopup.setConstrainViewport(true);
      miLinkFeed.addSelectionListener(new SelectionListener<MenuEvent>()
  {
        @Override
        public void componentSelected(MenuEvent me) {
            addFeedPopup.show(miLinkFeed.getElement(), "tl-bl?");
        }
      });
```

```
        MenuBar menuBar = new MenuBar();
        MenuBarItem menuBarItem = new MenuBarItem("Add feed", menu);
        menuBar.add(menuBarItem);

        setTopComponent(menuBar);

        add(new FeedList());
    }
```

ToolBar component

`ToolBar` is a component that goes beyond what you can do with simple buttons or menus. At present, in our example application, we are adding buttons to the `RssNavigationPanel` and `ContentPanel` using the `addButton` method and they are being placed in the default button location.

However, we can use a `ToolBar` to provide richer functions. With a `ToolBar` we are not just limited to buttons but can add other components such as a `ComboBox` or `Label`. In fact, most control components can be used in a `ToolBar`.

A `ContentPanel` provides a placeholder in which toolbars can be added at the top as well as the bottom of the panel.

To tidy up our **Create Feed** and **Import Feed** buttons, we are going to add a `ToolBar` with an **Add Feed** `Button` and create a submenu, which will perform the functions previously performed by the individual buttons.

Time for action – adding a toolbar

1. In the `RssNavigatorPanel`, create a new method named `initToolbar`

```
private void initToolbar() {
```

2. In the `initToolbar` method, create a new `ToolBar` component.

```
final ToolBar toolbar = new ToolBar();
```

3. Create an **Add feed** button, and assign an icon and a tooltip.

```
final Button btnAddFeed = new Button("Add feed");
btnAddFeed.setIconStyle("create-feed");

ToolTipConfig addFeedToolTipConfig = new ToolTipConfig();
addFeedToolTipConfig.setTitle("Add a new RSS feed");
addFeedToolTipConfig.setText("Adds a new RSS feed");
btnAddFeed.setToolTip(addFeedToolTipConfig);
```

4. Create a new `Menu` component.

```
Menu menu = new Menu();
```

5. Create a new menu item for **Create feed** and assign the `SelectionListener` to perform the same function as the **Create feed** button and add to the `Menu`.

```
final MenuItem miCreateFeed = new MenuItem("Create feed");
miCreateFeed.setIconStyle("create-feed");

ToolTipConfig createNewToolTipConfig = new ToolTipConfig();
createNewToolTipConfig.setTitle("Create a new RSS feed");
createNewToolTipConfig
        .setText("Creates a new RSS feed");
miCreateFeed.setToolTip(createNewToolTipConfig);
miCreateFeed.addSelectionListener(new
SelectionListener<MenuEvent>() {
  @Override
  public void componentSelected(MenuEvent me) {
    createNewFeedWindow();
  }
});
menu.add(miCreateFeed)add;
```

6. Do the same for **Link feed**:

```
final MenuItem miLinkFeed = new MenuItem("Link feed");
miLinkFeed.setIconStyle("link-feed");

ToolTipConfig linkFeedToolTipConfig = new ToolTipConfig();
linkFeedToolTipConfig.setTitle("Link to existing RSS feed");
linkFeedToolTipConfig
        .setText("Allows you to enter the URL of an existing RSS
feed you would like to link to");
miLinkFeed.setToolTip(linkFeedToolTipConfig);

final LinkFeedPopup addFeedPopup = new LinkFeedPopup();
addFeedPopup.setConstrainViewport(true);
miLinkFeed.addSelectionListener(new SelectionListener<MenuEvent>()
{
  @Override
  public void componentSelected(MenuEvent me) {
    addFeedPopup.show(miLinkFeed.getElement(), "tl-bl?");
    }
});
menu.add(miLinkFeed);
```

7. Now add the menu to the **Add feed** button using the `setMenu` method.

```
btnAddFeed.setMenu(menu);
```

8. Add the **Add feed** button to the toolbar.

```
toolbar.add(btnAddFeed);
```

9. Use `setTopComponent` to add the `ToolBar` to the underlying Container's top placeholder.

```
setTopComponent(toolbar);
```

10. Finally, modify the constructor of `RssNavigationPanel` to remove the existing buttons, and to add a button call the `initToolBar` method so that it looks like this:

```
public RssNavigationPanel() {
  setHeading("Navigation");
  setLayout(new FitLayout());
  initToolbar();
  add(new FeedList());
}
```

11. Now start the application and it will now have a `ToolBar` with an **Add feed** button and a `Menu` with **Create new feed** and **Link feed** options.

What just happened?

We created a `ToolBar` with a single button that in turn had a menu, which replaced the functions of our **Create feed** and **Import feed** buttons.

TabPanel class

The `TabPanel` class extends `Container` and acts as a container for displaying and managing `TabItem` objects. TabItem objects can be added and removed using the `add` and `remove` methods respectively. `TabItem` objects have an id what can be used with the `findItem` method to retrieve a `TabItem`. An existing `TabItem` can be selected and the selected `TabItem` retrieved by using the `setSelectedItem` and `getSelectedItem` methods respectively.

TabItem class

The `TabItem` class extends `LayoutContainer` and add the ability to be closed, disabled and to have an icon displayed in their heading when used in conjunction with a `TabPanel`. A closable TabItem with and icon set looks like this:

We shall be making use of tabs in *Chapter 7*.

Status component

`Status` is a component usually used with a `ToolBar` for creating a status bar similar to those seen in desktop applications.

The best way to demonstrate it is to add one to our example applications. It will not do much at the moment, but we will use it in later chapters.

Time for action – adding a Status component

1. In `RssMainPanel`, change the `Grid` added in the constructor back to `ItemGrid`.
2. Create a new `ToolBar` at the end of the current constructor.

```
ToolBar toolBar = new ToolBar();
```

3. Create a new `Status` component and set its width to 150 px.

```
Status status = new Status();
status.setWidth(150);
```

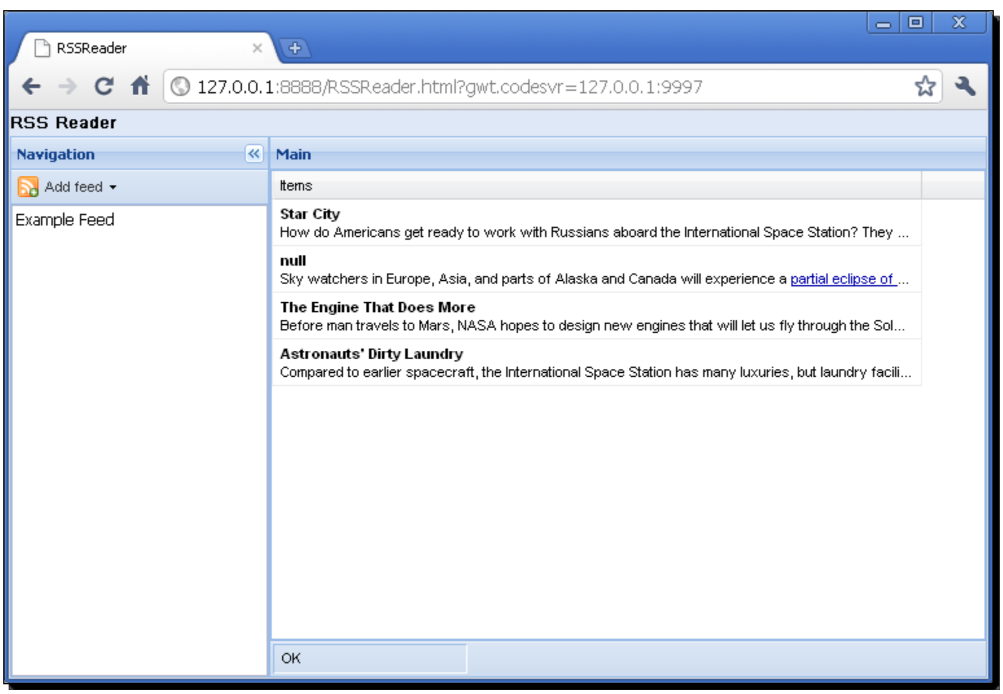
4. Use the `setBox` method of the `Status` component to display the status with an indented border and set the text of the `Status` to `OK`.

```
status.setBox(true);
status.setText("OK");
```

5. Add the status to the `ToolBar` and then set the `ToolBar` to be the bottom component of the underlying `ContentPanel`.

```
toolBar.add(status);
setBottomComponent(toolBar);
```

6. Start the application and the `ToolBar` with its status will be below the `Grid`.



What just happened?

We added a `ToolBar` to our application that included a `Status` component. At the moment, it just displays **OK**, but in the future we will make more use of it.

Pop quiz – matching the component with the definition

In this chapter, we have again covered a lot of components. Match the feature with the component that best matches in the following two lists:

1. `Menu`
2. `Status`
3. `ToolBar`
4. `MenuEvent`
5. `CheckMenuItem`
6. `PagingLoader`
7. `HeaderGroupConfig`
8. `TreeGridCellRenderer`
9. `BaseTreeModel`
10. `ImageBundle`

a. Extends `BaseModel` to add parent-child relationship management
b. Can appear on its own or in a `MenuBar`
c. A menu item with `CheckBox` or `RadioBox` functionality
d. Used to display a `Grid` column as a tree
e. Loads subsets of a dataset into a store.
f. Used to merge header columns or rows.
g. Interface that allows a set of images to be preloaded
h. Can contain `Button`, `ComboBox`, and other components
i. Is the `Menu` equivalent of `ButtonEvent`
j. Can display text in an indented box in a `ToolBar`

Summary

In this chapter, we have covered some of the more advanced data display and navigation components in GXT. We first looked at the `TreePanel` and saw how `TreeGrid` provided similar tree functions in a grid. We then went on to look at some of the more advanced grid features. Finally, we looked at toolbars and menus and how they can better organize user interaction.

In the next chapter, we will be looking at how to present data more creatively using GXT's template features.

6
Templates

In this chapter, we look at templates and how they can be used to easily format and display data in a highly customizable way. We also introduce the more powerful features of XTemplates.

Specifically, we will cover the following:

- `Template`
- `XTemplate`
- `RowExpander`
- `ListView`
- `ModelProcessor`
- `CheckBoxListView`

In previous chapters, we looked at automatically populating data-backed components using `ModelData` objects. This involved using a specific field from the `ModelData` object as a selectable value or as the value of a column.

What if we wanted to display more than one field? For example, what if we had a `ModelData` object with the first name and last name fields, but wanted to display the full name.

GXT has thought of this and provided two solutions. The first is a `ModelProcessor` that pre-processes `ModelData` to define additional fields. We will look at `ModelProcessor` later in this chapter. The other option is to use a `Template`.

First, however, we need to make some additions to the backend services to add more fields to the `Feed` and `Item` classes. These new fields will be used in this chapter.

Time for action – adding to the Feed and Item

1. In the `Feed` class, define two new fields, namely, a `String` to hold an image URL and a `List` of `Item` objects to hold the items for the feed:

```
private String imageUrl;
private List<Item> items = new ArrayList<Item>();
```

2. Add getters and setters for the newly created fields:

```
public String getImageUrl() {
  return imageUrl;
}

public List<Item> getItems() {
  return items;
}

public void setImageUrl(String imageUrl) {
  this.imageUrl = imageUrl;
}

public void setItems(List<Item> items) {
  this.items = items;
}
```

3. In the `Item` class, add getters and setters for new fields to hold publication data and the URL for a thumbnail:

```
public Date getPubDate() {
  return get("pubDate");
}

public String getThumbnailUrl()
{
  return get("thumbnailUrl");
}

public void setPubDate(Date pubDate) {
  set("pubDate", pubDate);
}

public void setThumbnailUrl(String thumbnailUrl) {
  set("thumbnailUrl", thumbnailUrl);
}
```

4. In the `FeedService` class, modify the definition of the `loadFeedList` method, so that there is a parameter to specify if the items should also be loaded:

```
List<Feed> loadFeedList(boolean loadItems);
```

5. Modify the `loadFeedList` method in the `FeedServiceAsync` method to match:

```
void loadFeedList(boolean loadItems, AsyncCallback<List<Feed>>
  callback);
```

6. Modify the `loadFeedList` method to include the `loadItems` parameter as defined in the interface and pass that parameter to the call to the `loadFeed` method:

```
@Override
public List<Feed> loadFeedList(boolean loadItems) {
  feeds.clear();
  Set<String> feedUrls = persistence.loadFeedList();
  for (String feedUrl : feedUrls) {
    feeds.put(feedUrl, loadFeed(feedUrl, loadItems));
  }
  return new ArrayList<Feed>(feeds.values());
}
```

7. In the `FeedServiceImpl` class, modify the `loadFeed` method to include the new `loadItems` parameter. If the `loadItems` is true, load the feed's items into the `items` field of the `Feed` object:

```
private Feed loadFeed(String feedUrl, boolean loadItems) {
  Feed feed = new Feed(feedUrl);
    …
    feed.setLink(eleChannel.getChildText("link"));
    if (loadItems) {
      feed.setItems(loadItems(feedUrl));
    }
  …
}}
```

8. Also retrieve any image available in the RSS feed XML, and if it exists, extract the URL of the image and use it to set the `imageUrl` field of the `Feed` object:

```
Element eleImage = eleChannel.getChild("image");
  feed.setImageUrl("");
  if (eleImage != null) {
    Element eleUrl = eleImage.getChild("url");
    if (eleUrl != null) {
      feed.setImageUrl(eleUrl.getText());
    }
  }
```

9. Similarly, in the `loadItems` method, extract any thumbnail from the item in the RSS feed XML. Also extract any publication date and use this data to set the `thumbnailUrl` and the `pubDate` fields of the `Item` object respectively:

```
Namespace ns =
  Namespace.getNamespace("media","http://search.yahoo.com/mrss/");
Element eleThumbnail = eleItem.getChild("thumbnail", ns);
if (eleThumbnail != null) {
  item.setThumbnailUrl(eleThumbnail.getAttributeValue("url"));
}
String pubDateStr = eleItem.getChildText("pubDate");
if (pubDateStr != null) {
try {
  DateFormat df = new SimpleDateFormat("EEE', 'dd' 'MMM' 'yyyy'
  'HH:mm:ss' 'Z");
  item.setPubDate(df.parse(pubDateStr));
} catch (ParseException e) {
  item.setPubDate(null);
}         }
```

10. Modify the `addExistingFeed` method so that the `loadFeed` method returns the feeds without the items loaded:

```
@Override
public void addExistingFeed(String feedUrl) {
  Feed loadResult = loadFeed(feedUrl, false);
  if (loadResult.getTitle() != null) {
    feeds.put(feedUrl, loadResult);
    persistence.saveFeedList(feeds.keySet());
  }
}
```

11. In the `FeedList` class, modify the call to the `loadFeedList` method of the `FeedService` to include a `false` parameter as we don't want to load the items in this case:

```
protected void load(Object loadConfig, AsyncCallback<List<Feed>>
  callback) {
  feedService.loadFeedList(false, callback);
}
```

What just happened?

We modified the `Feed` class, `Item` class, and `FeedService`. We can now retrieve a `Feed` object with the `Item` objects loaded. The `Feed` object now contains an URL to an image, if available. The `Item` object also contains an URL to a thumbnail and a publication date is available.

Template class

A `Template` is a class for generating HTML fragments that define how to render a `ModelData` or `Params` item as an HTML string. Templates are strings with placeholders for fields to be added.

To define a placeholder to insert a field into a string, we simply insert the field name surrounded by curly brackets. Creating a `Template` that uses fields named `firstName` and `lastName` would look like this:

```
Template template = new Template("My full name is {firstName}
  {lastName}.");
```

We can then define data with `firstName` and `lastName` fields like this:

```
Params data = new Params();
data.set("firstName", "Daniel");
data.set("lastName", "Vaughan");
```

We can apply the template to the data using the `applyTemplate` method of the `Template` object:

```
template.applyTemplate(data);
```

The `applyTemplate` method will then return a string that incorporates this data. In this case, that would be:

My full name is Daniel Vaughan.

Templates also can be pre-compiled, which reduces the overhead from using regular expressions. This is achieved by calling the `compile` method of the `Template` object.

In our example application, we have an `Item ModelData` object. It would be useful to create a new component named `ItemPanel` that takes an object and renders it to HTML. A `Template` is the ideal tool for achieving this. In this case, we are going to use the output of the `Template` to populate the value of a GWT HTML widget.

We are now going to create the `ItemPanel`.

Time for action – creating the ItemPanel

1. Create a new class in the `client.components` package named `ItemPanel` that extends `ContentPanel`:

   ```
   public class ItemPanel extends ContentPanel {
   ```

2. Create a new instance of the GWT `HTML` widget:

   ```
   private final HTML html = new HTML();
   ```

3. Override the `onRender` method, setting the title to `Item` and adding the `HTML` widget to the underlying `ContentPanel`:

```
@Override
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  setHeading("Item");
  add(html);
}
```

4. We want the `ContentPanel` to be filled with the `HTML` widget, so set the layout of the underlying `ContentPanel` to `FitLayout`. We also want the `HTML` widget to inherit a CSS style, so set the style of the `html` to `item`:

```
@Override
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  setHeading("Item");
  setLayout(new FitLayout());
  html.setStylePrimaryName("item");
  add(html);
}
```

5. We now need to construct our template string. Remember that a `Template` takes a standard `String`, so we can build it using a standard Java `StringBuilder`. Note that we are inserting fields into the template string by surrounding field names with curly brackets. For convenience, we put this in a method named `getTemplate`:

```
private String getTemplate() {
    StringBuilder sb = new StringBuilder();
    sb.append("<h1>{title}</h1>");
    sb.append("<p><i>{pubDate}</i></p>");
    sb.append("<hr/>");
    sb.append("<img src=\"{thumbnailUrl}\"/>");
    sb.append("<p>{description}</p>");
    return sb.toString();
  }
```

6. We can now create a public method named `displayItem`. This will take an `Item` object as a parameter. The underlying `JavaScript` object of the `Item` retrieved using the `Util.getJsObject` method will then be used as an argument to the `applyTemplate` method of the `Template` used to generate the relevant HTML string. This in turn will be used as the HTML for the `HTML` widget:

```
public void displayItem(Item item)
{
```

```
    setHeading(item.getTitle());
    Template template = new Template(getTemplate());
    html.setHTML(template.applyTemplate(Util.getJsObject(item, 1)));
  }
```

7. Now we need to add style definitions for rendering the `Item` objects. Create a new style sheet in the `war\css` folder named `item.css`. In `RSSReader.html`, add a reference to this new stylesheet in the `head` section of the HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-
  8">
<link type="text/css" rel="stylesheet" href="RSSReader.css">
<link type="text/css" rel="stylesheet" href="css/item.css">
<link type="text/css" rel="stylesheet" href="gxt/css/gxt-all.css">
<title>RSSReader</title>
</head>
```

8. In the `item.css` stylesheet, define styles for the `h1`, `img`, and `hr` elements. As the `HTML` widget is defined to inherit the `item` style, this will enable us to define styles just for rendering `Item` objects. Add the styles to the stylesheet as follows:

```
.item h1 {
  font-size: 1.5em;
}

.item img {
  border: 1px solid #000;
  float: left;
  margin-right: 10px;
}

.item hr {
  border-bottom: 1px solid #000;
}
```

9. We now need to create a test `Item` object to try out the `ItemPanel`. In the `client` package, create a new class named `TestObjects` and implement it as follows:

```
public class TestObjects {

  public static Item getTestItem()
  {
```

```
        Item testItem = new Item();
        testItem.setTitle("Computers get more powerful");
        testItem
          .setDescription("New computers are more powerful than the
            computers that were around a year ago. They are also much
            more powerful than the computers from five years ago. If
            you were to compare current computers with the computers
            of twenty years ago you would fine they are far more
            powerful.");
        testItem.setLink("http://www.example.com/item573.html");
        testItem.setPubDate(new Date());
        testItem.setCategory("Category");
        testItem
          .setThumbnailUrl("http://www.danielvaughan.com/gxt-
            book/examples/images/computers.jpg");
        return testItem;
      }

    }
```

10. In the main `RSSReader` class, comment out the following line and add the new lines to put the `ItemPanel` in place of the `RssMainPanel` and call the displayItem method of the `ItemPanel` with the `Item` retrieved from the `TestObjects` class:

```
//RssMainPanel mainPanel = new RssMainPanel();
ItemPanel mainPanel = new ItemPanel();
mainPanel.displayItem(TestObjects.getTestItem());
```

11. Start the application and you will see the `Item` object rendered as follows:

What just happened?

We created an `ItemPanel` in the example application. This uses a `Template` to render the data in a given `Item` object into HTML.

Using a Template with other components

As well as producing standalone HTML, a `Template` can be used with other components to define HTML with embedded fields. Specifically, a `Template` can be used with a `ListField`, a `ComboBox`, or `ToolTipConfig`.

When used with a `ListField` or `ComboBox`, the `Template` defines the appearance of each list item. For example, instead of a single field being displayed in a `ListField` as we have seen before, we can use a `Template` to combine both multiple fields and HTML.

As a `ListField` and `ComboBox` have multiple `ModelData` items to display, the `Template` needs to be applied to each one.

Templates have a special `<tpl>` tag, and this provides a `for` function to iterate through each item in a list and apply a template to it. We will cover `<tpl>` functions in more detail when we move onto `XTemplate`.

For the time being, we will modify the `FeedList` class in our example application to display both the `name` and part of the `description` fields of a `Feed` object, instead of just the `name` field. Although we are using a `ListField`, the same principle also applies to `ComboBox` components.

Time for action – using a Template with a ListField

1. First, we need to create a `getTemplate` method that returns the `Template` content as a string in our `FeedList` class:

```
private String getTemplate()
{

}
```

2. In the template string, we need to use the `<tpl>` to process each of the data objects in the store:

```
private String getTemplate() {
  StringBuilder sb = new StringBuilder();
  sb.append("<tpl for=\".\">");
  sb.append("</tpl>");
  return sb.toString();
}
```

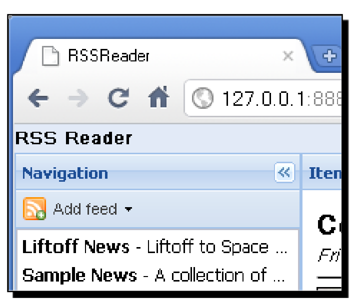
3. We can now define the actual template to display. Both `ListField` and `ComboBox` items require a `div` with the CSS class `x-combo-list-item` to function. In this case, we are defining the entry in the list to be made up of the `title` of the feed in bold, followed by the value of the `description` field:

```
private String getTemplate() {
  StringBuilder sb = new StringBuilder();
  sb.append("<tpl for=\".\">");
  sb.append("<div class='x-combo-list-item'><b>{title}</b> -
    {description}</div>");
  sb.append("</tpl>");
  return sb.toString();
}
```

4. Now we add a call to the `setTemplate` method of the `ListField` in place of the call to `setDisplayField` in the `onRender` method of the `FeedList` class:

```
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  ...
  feedList.setStore(feedStore);
  feedList.setTemplate(getTemplate());
  loader.load();
  ...
}
```

5. On starting the application, you will notice that the formatting of the `ListField` now has the title in bold and a part of the description for each feed:



What just happened?

We modified the `FeedList` class to use a `Template`, so that it now shows the title of the `Feed` in bold and part of the description, instead of the title alone.

XTemplate class

A `Template` is useful, but the `XTemplate` class is even more useful. An `XTemplate` is like a `Template` on steroids. An `XTemplate` performs the same functions as a `Template`, but adds a number of other useful capabilities.

As well as creating HTML templates that can contain field values like a normal `Template`, an `XTemplate` allows for basic programmatic functions to be defined using more custom template `<tpl>` tags.

For the following examples, let's first define a `ModelData` class named `Person` as follows:

```
public class Person extends BaseModel {

  public Person(String firstName,String lastName) {
    set("firstName", firstName);
    set("lastName", lastName);
  }
}
```

The for function

First of all, let's look at the `for` function we used in the previous example in more detail.

First, we will define a list of two people named `friends`:

```
List<Person> friends = Arrays.asList(new Person("Fred", "Bloggs"),
  new Person("John", "Smith"));
```

The `tpl` tag and the `for` operator can be used to move through the array of friends and apply a template block to each one. The "." specifies that the template should process each element of the provided list of `Person` objects:

```
<tpl for=".">
  <p>{firstName} {lastName}</p>
</tpl>
```

When applied, the template can produce a list of names of the two people in the `friends` list:

Fred Bloggs

John Smith

Let's create a new `Person` object and define a `friends` field that contains the `friends` list we previously defined:

```
Person person = new Person("Daniel", "Vaughan");
person.set("friends", friends);
```

Now we can process the person and then process the friends using a template like this to apply a template to the person object and a template block to each person object in the `friends` field:

```
<p>{firstName} {lastName}'s friends:</p>
<ul>
<tpl for="friends">
  <li>{firstName} {lastName}</li>
</tpl>
</ul>
```

To produce the following:

Daniel Vaughan's friends:

- **Fred Bloggs**
- **John Smith**

When using templates in the Java code, it makes sense to define the HTML in its own method, as the templates get more complex like this:

```
private String getTemplate() {
  StringBuilder sb = new StringBuilder();
  sb.append("<p>{firstName} {lastName}'s friends</p>");
  sb.append("<ul>");
  sb.append("<tpl for=\"friends\">");
  sb.append("<li>{firstName} {lastName}</li>");
  sb.append("</tpl>");
  sb.append("</ul>");
  return sb.toString();
}
```

The template can then be created and applied like this:

```
XTemplate xTemplate = XTemplate.create(getTemplate());
String html = template.applyTemplate(Util.getJsObject(person, 2));
```

Note that the `Util.getJsObject` returned the underlying JavaScript object of the `person` `ModelData` object. The second parameter is the number of levels of child objects to incorporate.

The if function

The `tpl` tag also has an `if` function for conditional processing.

Let's add an `age` field to the `Person` class:

```
public class Person extends BaseModel {

  public Person(String firstName,String lastName, int age) {
    set("firstName", firstName);
    set("lastName", lastName);
    set("age", age);
  }
}
```

Now let's define a `person` with `friends` again and include `ages` this time:

```
List<Person> friends = Arrays.asList(new Person("Fred", "Bloggs",
  20), new Person("John", "Smith", 40));
Person person = new Person("Daniel", "Vaughan", 30);
person.set("friends", friends);
```

We can use a `tpl` tag `if` function to restrict the list of `friends` to those over `30`:

```
<p>{firstName} {lastName}'s friends over 30:</p>
<ul>
<tpl for="friends">
  <tpl if="age &gt; 30">
    <li>{firstName} {lastName}</li>
  </tpl>
</tpl>
</ul>
```

Note that we must encode the greater than `>` operator as `>` in order for it to work.

The following operators are available:

Comparison	Operator	Note
Equals	==	If testing a string
Greater than	>	Encode > as >
Less than	<	Encode < as <
Not Equals	!=	

When applied as before, the result will be a list containing only the friend over 30; John Smith:

Daniel Vaughan's friends over 30:

- **John Smith**

There is no `else` function available in `tpl` tags. If that functionality is needed, we can use the inverse of the `if` statement.

We can use fields in `if` comparison statements. For example, instead of saying friends over 30 to produce the above, we can say friends older than the person's age. We use the `parent` variable to refer to a `ModelData` object's parent, which will produce the same result as the previous example:

```
<p>{firstName} {lastName}'s friends over {age}:</p>
<ul>
<tpl for="friends">
  <tpl if="age &gt; parent.age">
    <li>{firstName} {lastName}</li>
  </tpl>
</tpl>
</ul>
```

> Warning: When creating your model elements, avoid using hyphens in the field names. This is because when used with a `tpl` function such as `if`, the hyphen will be interpreted as a minus sign between two fields and evaluation will fail. Therefore, use camel case when your fields consist of two names, for example, `firstName` instead of `first-name`.

Special built-in template variables

There are also a number of build template variables that can be used:

Template variable	Description
`{#}`	Special field which will auto number each item.
`parent`	The parent of the value in scope
`values`	The values in the current scope
`{[ … ]}`	Anything enclosed in this way will be treated as executable code
`xindex`	The current index of an array being looked at in a `for` statement (1-based)
`xcount`	The length of the array that is being looped in a `for` statement
`fm`	An alias for the format function

Basic math function support

It is also possible to perform basic math functions on fields in templates.

For example, if we wanted to add 1 to a field named `age`, we would simply add it to the template as `{age+1}`.

If we wanted to use a template to display friends older than the person and show how many years older they were, we could do this:

```
<p>{firstName} {lastName}'s friends over {age}:</p>
<ul>
<tpl for="friends">
  <tpl if="age &gt; parent.age">
    <li>{firstName} {lastName} ({age}-{parent.age} years older)</li>
  </tpl>
</tpl>
```

Inline code execution

We can go even further by creating member functions within templates for more complex functions, but that is out of the scope of this beginner's guide.

Using an XTemplate

An `XTemplate` can be used to process the values displayed in several components. In addition to being able to be used in `ComboBox` and `ListField` like a `Template`, an `XTemplate` can be used with the following components, some of which will be explained below:

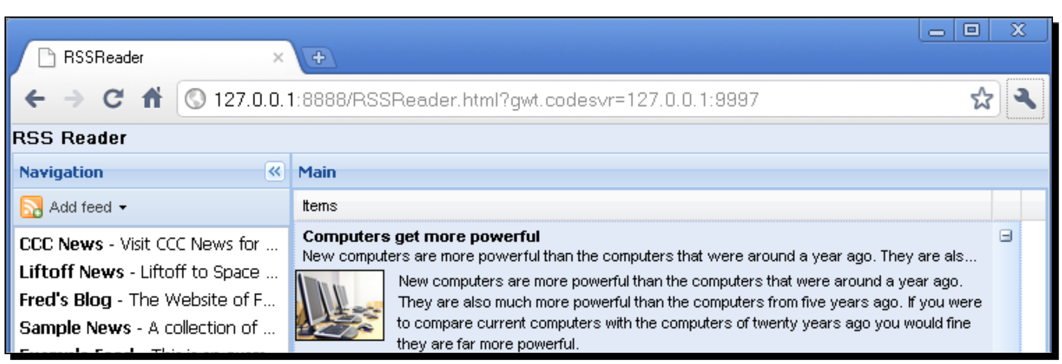
- `RowExpander`
- `ListView`
- `CheckBoxListView`
- `ColorPalette`

The RowExpander class

An `XTemplate` can be used to style a column of a Grid. The `RowExpander` class extends the `ColumnConfig` class we covered in *Chapter 5*. As a result, it is defined in a similar way to a `ColumnConfig` and can be added to a `ColumnModel` in the same way.

When a `RowExpander` is added to a `Grid`, it appears as a column containing a small **+** button like the one shown on the far right in the screenshot:

When the button is clicked on, the row expands to show more information, as defined by an `XTemplate` like this:



In order for a `RowExpander` to take effect, however, we must remember to add it to the `Grid`, specifically using the `addPlugin` method.

When used, the content defined in the `XTemplate` associated with the `RowExpander` is applied to all the rows of the column automatically. We do not have to use the `tpl for` tag as with a `ListField`.

We will now create a new version of our example application's `ItemGrid` that makes use of a `RowExpander` to display data.

Time for action – using a RowExpander

1. In the `onRender` method of the `ItemGrid`, create a new `XTemplate` after the last column definition. The actual template is an image with the `src` being the value of the `thumbnailUrl` field of the `Item` object followed by the value of the `description` field. Since the actual template string is only one line, it makes sense to enter it directly as a parameter to the `create` method of the `XTemplate`:

```
XTemplate xTemplate = XTemplate
  .create("<img class=\"left\" src=\"{thumbnailUrl}\"
    height=\"49px\"/><p>{description}</p>");
```

2. Next, create a new `RowExpander` instance and set the `Template` to be the `XTemplate` that we have just defined:

```
RowExpander rowExpander = new RowExpander();
rowExpander.setTemplate(xTemplate);
```

3. Now add the `RowExpander` to the grid's columns in the same way that you would add a normal `ColumnConfig`:

```
columns.add(rowExpander);
```

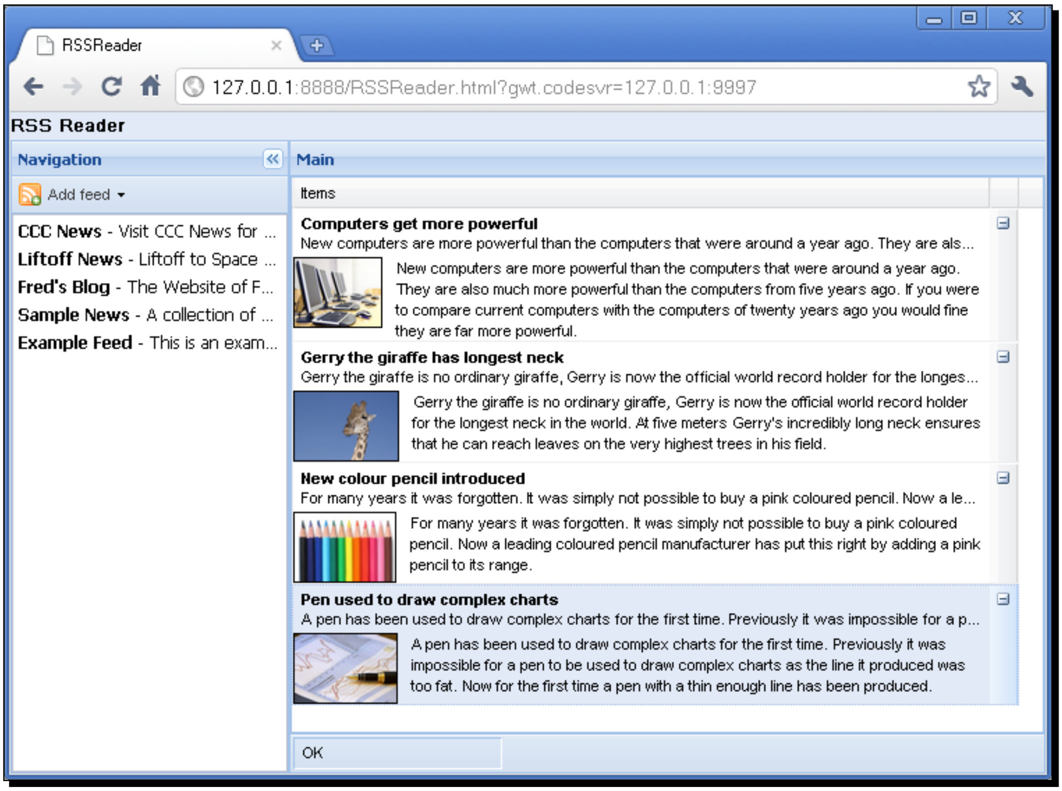
4. Also add the `RowExpander` to the `Grid` as a plugin to allow it to work:

```
grid.addPlugin(rowExpander);
```

5. In the `RSSReader` class, remove the `ItemPanel` code we added earlier and uncomment the commented code to reinstate `RssMainPanel` as `mainPanel`:

```
RssMainPanel mainPanel = new RssMainPanel();
```

6. When you now start the application, you will notice that all the item rows in the `ItemGrid` can be expanded to show further details:



What just happened?

We added a `RowExpander` to the `ItemGrid` to allow rows to be expanded to give more details.

The ListView class

`ListView` allows for the custom display of a list of data using an `XTemplate` object. This is a very flexible component as it lets us control exactly how the data is displayed, whether as icons, a grid, a list, or whatever else we can construct with a combination of `XTemplates` and CSS.

To demonstrate how a `ListView` can work, we are going to create a `ListView` that renders a list of `Feed` objects as a list of boxes.

Time for action – creating a Feed overview ListView

1. Create a new class in the `client.lists` package named `FeedOverviewView` that extends `LayoutContainer`:

```
public class FeedOverviewView extends LayoutContainer {
```

2. Define a `ListView` field:

```
private ListView<BeanModel> listView = new ListView<BeanModel>();
```

3. Override the `onRender` method of the `LayoutContainer` and add a `DataProxy`, `DataReader`, and `Loader` to populate a `feedStore` in the same way as we did in the `FeedList` class:

```
  @Override
  protected void onRender(Element parent, int index) {
    super.onRender(parent, index);

    final FeedServiceAsync feedService = (FeedServiceAsync)
      Registry
        .get(RSSReaderConstants.FEED_SERVICE);

    RpcProxy<List<Feed>> proxy = new RpcProxy<List<Feed>>() {
      @Override
      protected void load(Object loadConfig,
        AsyncCallback<List<Feed>> callback) {
          feedService.loadFeedList(false, callback);
      }
    };
    BeanModelReader reader = new BeanModelReader();

    ListLoader<ListLoadResult<BeanModel>> loader = new
      BaseListLoader<ListLoadResult<BeanModel>>(
        proxy, reader);

    ListStore<BeanModel> feedStore = new
      ListStore<BeanModel>(loader);
    loader.load();
}
```

4. Now define a `getTemplate` method that returns the string to use to generate an `XTemplate`. In this case, we are applying it to all feed objects in the list and only adding an image if the `imageUrl` is not blank, using a `tpl if` function:

```
private String getTemplate() {
  StringBuilder sb = new StringBuilder();
  sb.append("<tpl for=\".\">");
  sb.append("<div class=\"feed-box\">");
  sb.append("<h1>{title}</h1>");
  sb.append("<tpl if=\"imageUrl!=''\">");
  sb.append("<img class=\"feed-thumbnail\" src=\"{imageUrl}\"
    title=\"{title}\">");
  sb.append("</tpl>");
  sb.append("<p>{description}</p>");
  sb.append("</div>");
  sb.append("</tpl>");
  return sb.toString();
}
```

5. Returning to the `onRender` method, set the store of the `ListView` and then set the template using the string obtained from the `getTemplate` method:

```
listView.setStore(feedStore);
listView.setTemplate(getTemplate());
```

6. Then add the `ListView` to the underlying `LayoutContainer`:

```
add(listView);
```

7. We now need to add a few styles to the `war\items.css` stylesheet to control how the template is rendered. The style `div.feed-box` defines a box that acts as a container for a feed and `img.feed-thumbnail` defines the size of the image to display, if any:

```
div.feed-box {
  float: left;
  margin: 5px;
  padding: 5px;
  border: 1px solid black;
  width: 200px;
  height: 120px;
  text-align: center;
}

img.feed-thumbnail {
```

```
    width: 100px;
    height: 100px;
  }
```

8. In the `RssMainPanel` class, add a new instance of `FeedOverviewView` in the place of the existing `ItemGrid`:

```
add(new FeedOverviewView());
```

9. The result is a `ListView` of feeds that looks like this:

What just happened?

We used a `ListView` to create a custom rendering of our feed list using XTemplates and CSS. However, it is not perfect, because one of the descriptions is too long, and we will address that next.

The ModelProcessor class

The `ModelProcessor` class provides a way to pre-process model data using templates before it is passed to a data component. It does not change the actual values of fields in the model data object. Rather, it allows us to create new fields containing the result of formatting the data, which can be processed in the same way as normal fields.

For example, one of the descriptions of a feed in the `ListView` we just built for our example application is too long and spills out of the box. What we can do is abbreviate the description using GXT's `Format.ellipse` method. This abbreviates a string to a defined number of letters and then adds three dots at the end to show that this has happened.

As a `ModelProcessor` is built into the `ListView`, pre-processing model data involves overriding the `prepareData` function of the `ListView`, and that is what we will do now. We are going to create shorter versions of both the `title` and the `description` fields.

Time for action – pre-processing model data

1. In the `onRender` method of the `FeedOverviewView`, override the `prepareData` method of the `ListView`:

```
listView = new ListView<BeanModel>() {
  @Override
  protected BeanModel prepareData(BeanModel feed) {
    return feed;
  }
};
```

2. This will return just the feed without modifications. We now need to add a new field to the feed object named `shortTitle` containing the content of the `title` field, abbreviated to `50` characters:

```
listView = new ListView<BeanModel>() {
  @Override
  protected BeanModel prepareData(BeanModel feed) {
    feed.set("shortTitle", Format.ellipse((String)
      feed.get("title"), 50));
    return feed;
  }
};
```

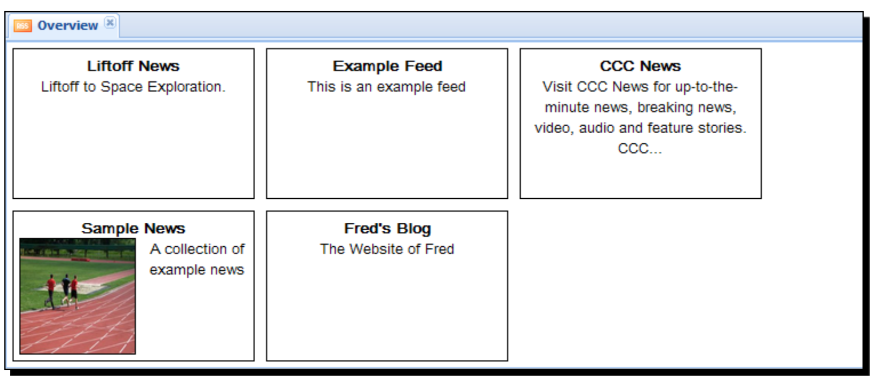
3. We then need to create a `shortDescription` field from the `description` field, but this time abbreviated to `100` characters:

```
listView = new ListView<BeanModel>() {
  @Override
  protected BeanModel prepareData(BeanModel feed) {
    feed.set("shortTitle", Format.ellipse((String)
      feed.get("title"), 50));
    feed.set("shortDescription", Format.ellipse((String)
      feed.get("description"), 100));
    return feed;
  }
};
```

4. Now we can modify the template defined in the `getTemplate` method to use the `shortTitle` and `shortDescription` fields instead of `title` and `description`:

```
private String getTemplate() {
  StringBuilder sb = new StringBuilder();
  sb.append("<tpl for=\".\">");
  sb.append("<div class=\"feed-box\">");
  sb.append("<h1>{title}</h1>");
  sb.append("<tpl if=\"imageUrl!=''\">");
  sb.append("<img class=\"feed-thumbnail\" src=\"{imageUrl}\"
    title=\"{shortTitle}\">");
  sb.append("</tpl>");
  sb.append("<p>{shortDescription}</p>");
  sb.append("</div>");
  sb.append("</tpl>");
  return sb.toString();
}
```

5. The offending long `description` field has now been abbreviated so that it neatly fits into the box:



What just happened?

We used the `ModelProcessor` built into `ListView` to create two new abbreviated fields for use in the `ListView`.

Item selectors

You may notice that the items in the `FeedOverviewView` are not selectable. This is because when we use a custom template with a component, like `ListView`, we must set an item selector. This is a block that can be selected. In this case, we will want to use the `feed-box` div. The `FeedOverviewView` will then make the block selectable and respond to selection events.

Time for action – making ListView items selectable

1. First, in the `onRender` method of the `FeedOverviewView`, we need to define the item selector of the `ListView` to be the `feed-box` div:

```
listView.setItemSelector("div.feed-box");
```

2. We can then add a `Listener` for the `SelectionChange` event to the `ListView` that will display the name of the feed selected in an `Info` box:

```
listView.getSelectionModel().addListener(Events.SelectionChange,
  new Listener<SelectionChangedEvent<BeanModel>>() {
  public void handleEvent(SelectionChangedEvent<BeanModel> be) {
    BeanModel feed = (BeanModel) be.getSelection().get(0);
    Info.display("Feed selected", (String)feed.get("title"));
  }
});
```

3. Now start the application and select a feed from the `FeedOverviewView`. The name of the feed will be displayed in the `Info` box:

What just happened?

We used the `setItemSelector` method of the `ListView` to define a selectable block in the `FeedOverviewView` and added a selection listener.

Have a go hero – showing item titles in the feed overview

Now we have a `ListView` that previews all the feeds. Modify the `XTemplate` of the `FeedOverviewView` so that it lists the title of the first two items in each feed as bullets like this:

Remember that the `loadFeedList` method of the `FeedService` now has the ability to load `Feed` objects that contain the child `Item` objects.

Solution:

Modified load method of the RpcProxy in the FeedOverviewView class:

```
RpcProxy<List<Feed>> proxy = new RpcProxy<List<Feed>>() {
  @Override
  protected void load(Object loadConfig,
      AsyncCallback<List<Feed>> callback) {
    feedService.loadFeedList(true, callback);
  }
};
```

Modified getTemplate method:

```
  private String getTemplate() {
    StringBuilder sb = new StringBuilder();
    sb.append("<tpl for=\".\">");
    sb.append("<div class=\"feed-box\">");
    sb.append("<h1>{title}</h1>");
    sb.append("<tpl if=\"imageUrl!=''\">");
    sb.append("<img class=\"feed-thumbnail\" src=\"{imageUrl}\"
      title=\"{shortTitle}\">");
    sb.append("</tpl>");
```

```
        sb.append("<p>{shortDescription}</p>");
        sb.append("<ul>");
        sb.append("<tpl for=\"items\">");
        sb.append("<tpl if=\"xindex &lt; 3\">");
        sb.append("<li>{title}</li>");
        sb.append("</tpl>");
        sb.append("</tpl>");
        sb.append("</ul>");
        sb.append("</div>");
        sb.append("</tpl>");
        return sb.toString();
    }
```

Additional CSS:

```
.feed-box li {
  text-align: left;
  list-style: circle inside;
}

.feed-box ul {
  clear: both;
}
```

CheckBoxListView

`CheckBoxListView` extends `ListView` by adding CheckBox functionality that allows the selection of multiple items by checking them. It works in exactly the same way as a `ListView`, but just puts a CheckBox alongside each item in the list to allow users to select items.

Pop quiz – what does what?

We have covered quite a few concepts in this chapter, but can you remember what does what? Match the component or function with the definition:

1. `Template`
2. `<tpl>`
3. `<tpl for=".">`
4. `<tpl if>`
5. `xindex`
6. `RowExpander`
7. `ListView`

a. Template function for conditional processing
b. The current index in a for statement
c. Basic way of generating an HTML fragment containing fields.
d. Template function that iterates through the values in scope.
e. A templated component that can be used in place of a `ColumnConfig`
f. Special HTML tag for enclosing template functions
g. A flexible list that uses XTemplates and CSS to display options

Summary

We have looked at templates and seen how they can be used both on their own and to provide more power to components we have come across before. We covered both the basic `Template` and the more powerful `XTemplate`. We also looked at the `ListView` that gives us a very versatile way of displaying data.

We have now built several components for our application. In the next chapter, we will start joining them together using GXT's model view controller functionality.

7
Model View Controller

In the previous chapters, we have mainly been dealing with individual components in isolation.In this chapter, we will look at GXT's Model View Controller framework and how it can allow components to communicate in larger applications

Specifically, we will cover the following classes:

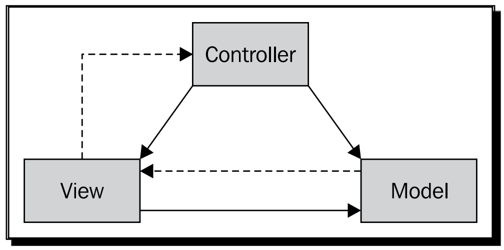
- `AppEvent`
- `EventType`
- `Controller`
- `View`
- `Dispatcher`

The need for good application structure

When building an application with GXT, it is important to think carefully about how it is constructed. Once an application starts growing, it is easy to run into problems very quickly. As components get more and more inter-dependent or coupled, it becomes very difficult to keep track of what is going on. This leads to a potent maintenance nightmare.

A standard solution to the problem of structuring GUI applications both on the desktop and the Web has been to use a framework that implements a **Model View Controller** (**MVC**) pattern. Fortunately, GXT includes an MVC implementation, and using this can save a lot of headaches.

The classic Model View Controller pattern



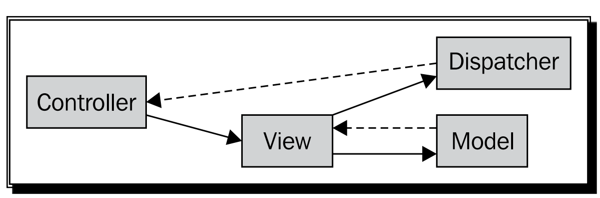
Model View Controller is a popular design pattern. It has several variations, but ultimately it is concerned with dividing up responsibilities into three parts:

- Model: It holds state, data, and application logic. It provides an interface that allows its state to be retrieved and changed. Observers can register so that they are notified when the model's state changes.
- View: This is the user interface. It responds to state changes from the model by requesting data and presenting it to the user. When the user interacts with the user interface, the view fires events that can be observed by the controller. The view does not normally have any knowledge of the controller.
- Controller: The controller observes events from the view and either makes a change to the model or the view as a result.

The strength of this pattern is that the model does not need to know anything about the controller or the view, and so is not dependent on either.

The GXT Model View Controller

The GXT Model View Controller is a bit different from the class MVC pattern, but it is still very useful.



- Model: It takes the form of the `ModelData` objects in a `Store`, as covered in previous chapters. Individual `ModelData` objects can be retrieved from stores and manipulated through their get and set methods.

- View: It organizes the UI components. As with the classic MVC pattern, data-backed components observe a model and respond to changes. Unlike the classic MVC model, components in a view can make changes to the model by loading the data into it. The view uses the dispatcher to fire the events that can be observed by the controller. GXT is designed in such a way that the view has knowledge of the controller as a `Controller` object is passed to the constructor of the `View` class. However, it is good practice for the view to not communicate with the controller directly, as this would break the MVC pattern.
- Controller: The classic MVC pattern responds to events received from the view via the dispatcher. It can then either perform an operation on the model or forward an event onto a view.
- Dispatcher: Instead of the controller observing the view directly, the view fires the events using the dispatcher. `Dispatcher` is a class with static methods that can be called to forward events to controllers. The controller then registers with the dispatcher to receive specific event types.

The AppEvent class

The messages that pass between controllers and views are instances of the `AppEvent` class. Each `AppEvent` object has a specific type defined by an `EventType` object.

Optionally, an `AppEvent` can contain a payload of one or more items of data by using the `setData` methods. This is useful for passing the state information. If we want to include more than one data object in an `AppEvent`, we need to pass a key as a `String` to allow us to retrieve that object later.

Another option is to use the `setHistoryEvent` method to set the `AppEvent` as a history event. This means that when the event is passed to the dispatcher, a history item is created for it. The consequence of this is that the dispatcher can be queried for a history of the events fired.

The EventType class

An `EventType` defines a custom type of `Event` that can be used to set the type of an `AppEvent`.

Typically, we will define each `EventType` a static field in an `AppEvents` class.
We will now define two `EventType` objects for the example application.

Time for action – defining application events

1. Create a new class named `AppEvents` in a new package named `client.mvc.events`.
2. In the newly created class, define two event type fields—one named `Init` and the other `Error`.

```
public class AppEvents {
  public static final EventType Init = new EventType();
  public static final EventType Error = new EventType();
}
```

What just happened?

We created a class to hold our application's events and defined an `Init` and an `Error` event type in it.

Controller class

A `Controller` processes and responds to events in the application.

A `Controller` must register the event types it wishes to observe in its constructor. The `registerEventTypes` method is used for this and takes `EventType` objects as parameters.

Time for action – creating a controller

1. Create a new class named `AppController` that extends `Controller` in a new package named `client.mvc.controllers`.
2. In the constructor of the `Controller`, register to respond to both event types we defined in the `AppEvents` class.

```
public AppController() {
  registerEventTypes(AppEvents.Init);
  registerEventTypes(AppEvents.Error);
}
```

What just happened?

We created a `Controller` and registered the `Init` and `Error` `EventType` objects for the `Controller` to respond to.

When creating a `Controller`, it is necessary to implement the `handleEvent` method. This method defines how the `Controller` will handle each `EventType`.

If we want to make a *query* about whether a `Controller` can handle a particular `AppEvent`, we can use the `canHandle` method.

There are a number of different actions that we can take as a response to an event.

- Handle it in the controller
- Delegate it to a child controller
- Forward it onto a view for further action
- A combination of all three

We will now implement `handleEvent` in our `AppController`.

Time for action – handling events

1. In the `AppController` class, override the `handleEvent` method.

```
@Override
public void handleEvent(AppEvent event) {
```

2. At the moment, we do not need to handle any events in the `Controller`. We just want to pass all events onto the `View`. However, we have yet to create a `View`. So let's just define the `View` for now.

```
private View appView;
```

3. With the `View` defined, we can forward all events to the `View` in the `handleEvent` method.

```
@Override
public void handleEvent(AppEvent event) {
forwardToView(appView, event);
}
```

What just happened?

We implemented the `handleEvent` method of the `AppController` so that it forwards all the events to the associated `View`. However, while running, this code will cause an error as we have not created a `View` class.

The View class

The `View` class is the part of the GXT MVC framework that provides the user interface. It is responsible for displaying components and reacting to events forwarded from the `Controller`. It also responds to user actions by forwarding the `AppEvents` to the `Dispatcher`.

Like controllers, views are required to implement the `handleEvent` method. To keep the code tidy, it is helpful to create an `on<EventType>` method for each `EventType`.

For example, if we wanted to handle the `Init EventType`, we would check for the `EventType` and create and call a method named `onInit`.

```
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.Init)) {
    onInit(event);
  }
}
```

We will now create a `View` for our example application.

Time for action – creating a View

1. Create a new class named `AppView` that extends `View` in a new package named `client.mvc.views`

2. Create a constructor for the class which takes an `AppController` as an argument, and with this, calls the constructor of the super class.

   ```
   public AppView(AppController appController) {
     super(appController);
   }
   ```

3. As we are interested in both the `Init` and `Error` event types, we created two methods named `onInit` and `onError`.

   ```
   private void onInit(AppEvent event) {}

   private void onError(AppEvent event) {}
   ```

4. Now implement the `handleEvent` method to call the correct method based on the `EventType` of the `AppEvent`.

   ```
   @Override
   protected void handleEvent(AppEvent event) {
         EventType eventType = event.getType();
   ```

```
        if (eventType.equals(AppEvents.Init)) {
            onInit(event);
        } else if (eventType.equals(AppEvents.Error)) {
            onError(event);
        }
    }
```

5. Finally, we can tell the `AppController` and `AppView` about each other. We do this by implementing the `initialize` method in `AppController`.

```
@Override
public void initialize() {
  super.initialize();
  appView = new AppView(this);
}
```

What just happened?

We created a `View` and created a framework to allow it to handle events.

As we had both a `View` and a `Controller`, we used the `initialize` method of the `AppController` to relate `AppController` and `AppView` to each other.

> Note that the `Controller` has a reference to the `View` and the `View` has a reference to the `Controller`. It is unusual for a `View` in an MVC pattern to have a reference to the `Controller`. However, this is the way GXT works.
>
> Even though the `View` does have a reference to the `Controller`, do not be tempted to call the methods in the `Controller` directly from the `View`. Forward an `AppEvent` to the `Dispatcher` from the `View` and have the `Controller` observe them instead. This avoids making the `View` dependent on the `Controller`, or in other words, makes them loosely coupled. This means that it is easier to make changes to the `Controller` without having knock-on effects for the `View`.

Dispatcher

`Dispatcher` is a singleton class, a class that is limited to a single instance that is available across the application. It has static methods that can be used to forward `AppEvent` objects. A `Controller` registers with the `Dispatcher` to observe `AppEvent` objects of a specific `EventType`. When an `AppEvent` is dispatched all `Controller` objects that have registered to observe the `EventType` will be notified with the `AppEvent`.

An event can be fired from anywhere in the application using one of the static `forwardEvent` methods of the `Dispatcher`. There are four convenient methods to the `forwardEvent` method that take different arguments.

forwardEvent Method	**Description**
`forwardEvent(AppEvent event)`	Takes an existing `AppEvent` and forwards it to the `Dispatcher`.
`forwardEvent(EventType eventType)`	Creates a new `AppEvent` of the specified `EventType` and forwards it.
`forwardEvent(EventType eventType, java.lang.Object data)`	Creates a new `AppEvent` of the specified `EventType` with the specified data object as the payload and forwards it.
`forwardEvent(EventType eventType, java.lang.Object data, boolean historyEvent)`	Creates a new `AppEvent` of the specified `EventType` with the specified data object as the payload and allows us to create the `AppEvent` as a history event and forwards it.

As well as having static `forwardEvent` methods, the `Dispatcher` also has non-static `dispatch` methods that perform the same function. In fact, the static `forwardEvent` methods call the `dispatch` methods. The only difference is that there is not a version of the `dispatch` method that allows a history event to be created.

> If the `Dispatcher` has multiple controllers registered, it will service them in the order in which they were added to the `Dispatcher`. When using multiple controllers, it may be important to be aware of this to manage which `Controller` gets to handle an `AppEvent` first.

Pop quiz: MVC fundamentals

In GXT's MVC implementation, which component or components do the following?

1. Dispatches `AppEvents`
2. Observes the Dispatcher
3. Handles events
4. Defines a type of `AppEvent`
5. Can forward `AppEvents` to the dispatcher

6. Can Dispatch events
7. Can add `Controllers`
8. Can register to receive `AppEvents` of a specified `EventType`

a. Dispatcher
b. Controller
c. View
d. EventType

Incorporating MVC

As we are now using the GXT MVC framework, this gives us an opportunity to re-factor the code to make the individual components more self-contained.

In order for a `Controller` to start receiving events, it needs to be registered with the `Dispatcher`. This is normally done in the `EntryPoint` class and that is what we are now going to do in our example application.

Time for action – registering a Controller with the Dispatcher

1. In the `onModuleLoad` method of the `RSSReader` remove all the existing code apparent from the line that registers the `FeedService`.

```
public void onModuleLoad() {
Registry.register(RSSReaderConstants.FEED_SERVICE, GWT.
create(FeedService.class));
}
```

2. In its place, retrieve the `Dispatcher` instance.

```
public void onModuleLoad() {
       Registry.register(RSSReaderConstants.FEED_SERVICE, GWT.
                           create(FeedService.class));
       Dispatcher dispatcher = Dispatcher.get();
}
```

3. Now register the `AppController` with the `Dispatcher`

```
public void onModuleLoad() {
      Registry.register(RSSReaderConstants.FEED_SERVICE, GWT.
                  create(FeedService.class));
      Dispatcher dispatcher = Dispatcher.get();
      dispatcher.addController(new AppController());
}
```

What just happened?

We registered the `AppController` with the `Dispatcher`. When the `Dispatcher` receives an `AppEvent`, it will check each `Controller` registered with it. If the `AppEvent` is of an `EventType` that the `Controller` is registered to observe, the dispatcher will call the `handleEvent` method of the `Controller`.

We also just removed the code that laid out the UI from the `onModuleLoad` method. Now to replace this, we are going to lay out the UI in our `AppView` class in response to an `AppEvent` of the `Init EventType`, initiated from the `onModuleLoad` method.

Time for action – refactoring UI setup

1. In the `onModuleLoad` method of the `RSSReader` class, use the `Dispatcher` to dispatch an event with the `Init EventType`.

```
public void onModuleLoad() {
      Registry.register(RSSReaderConstants.FEED_SERVICE, GWT.
                        create(FeedService.class));
      Dispatcher dispatcher = Dispatcher.get();
      dispatcher.addController(new AppController());
      dispatcher.dispatch(AppEvents.Init);
}
```

2. In the `AppView` class, create two new instances—`ContentPanel` and `Viewport`.

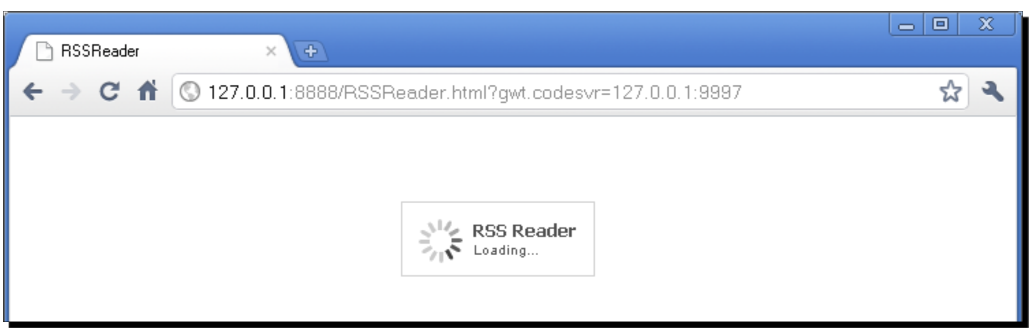
```
private final ContentPanel mainPanel = new ContentPanel();
private final Viewport viewport = new Viewport();
```

3. In the `onInit` method of the `AppView` class, insert the following UI creation code, which is similar to the code that we removed from the `onModuleLoad` method.

```
private void onInit(AppEvent event) {
  final BorderLayout borderLayout = new BorderLayout();
  viewport.setLayout(borderLayout);

  HTML headerHtml = new HTML();
```

```
  headerHtml.setHTML("<h1>RSS Reader</h1>");
  BorderLayoutData northData = new
BorderLayoutData(LayoutRegion.NORTH,20);
  northData.setCollapsible(false);
  northData.setSplit(false);
  viewport.add(headerHtml, northData);

  BorderLayoutData centerData = new BorderLayoutData(LayoutRegion.
CENTER);
  centerData.setCollapsible(false);

  RowLayout rowLayout = new RowLayout(Orientation.VERTICAL);
  mainPanel.setHeaderVisible(false);
  mainPanel.setLayout(rowLayout);
  viewport.add(mainPanel, centerData);
}
```

4. Start the application now and you will only see the loading message, as we are not adding the `Viewport`. We will do this in response to a separate `AppEvent`.



5. In the `AppEvents` class, add a new event named `UIReady`.

```
public static final EventType UIReady = new EventType();
```

6. In the constructor of the `AppController` class, register the `UIReady` event type.

```
public AppController() {
  registerEventTypes(AppEvents.Init);
  registerEventTypes(AppEvents.Error);
  registerEventTypes(AppEvents.UIReady);
}
```

7. In the `AppView` class, create a new method named `onUIReady` that adds the `Viewport` to the `RootPanel`.

```
private void onUIReady(AppEvent event) {
  RootPanel.get().add(viewport);
}
```

8. In the `handleEvent` method of `AppView`, respond to the `UIReady EventType` by calling the `onUIReady` method.

```
@Override
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.Init)) {
    onInit(event);
  } else if (eventType.equals(AppEvents.Error)) {
    onError(event);
  } else if (eventType.equals(AppEvents.UIReady)) {
    onUIReady(event);
  }
}
```

9. In the `onModuleLoad` method of the `RSSReader` class, dispatch a `UIReady AppEvent`.

```
public void onModuleLoad() {
       Registry.register(RSSReaderConstants.FEED_SERVICE, GWT.
                           create(FeedService.class));
       Dispatcher dispatcher = Dispatcher.get();
       dispatcher.addController(new AppController());
       dispatcher.dispatch(AppEvents.Init);
       dispatcher.dispatch(AppEvents.UIReady);
}
```

10. Start the application again. This time the `Viewport` will be added, but there will be no components on the screen.

What just happened?

We moved the UI setup code from the `onModuleLoad` method of the `EntryPoint` into the `onInit` method of the `AppView` class. We also added the `Viewport` to the `RootPanel` in response to the `UIReady` event.

In the `EntryPoint` class, we dispatched an `AppEvent` with the `Init EventType`. The `AppController` handled this event by forwarding it to the `AppView`. `AppView`, in turn, handled the event by calling the `onInit` method and the basics of the UI were set up. What it didn't do, however, was that it did not add components to the UI.

What we can do now is that we can create a separate `Controller` and `View` to manage each of the main components of the application independently, starting with the navigation component.

- The `Controller` for the navigation component will handle an `Init EventType` by forwarding the event onto the `View`.
- The `View` for the navigation component will handle events of the `Init EventType` by forwarding an `AppEvent` of the type `NavPanelReady` to the `Dispatcher`. The data payload of the event will contain an instance of `NavPanel`.
- The `AppController` will observe this `EventType`, and when one is received, will forward it on the `AppView`.
- The `AppView` will handle the `NavPanelReady EventType` by adding the `NavPanel` contained in the events' data payload to the `Viewport`.

By the time that the `UIReady` event is dispatched by the `onModuleLoad` method of the `EntryPoint` class, the `NavPanel` will have been added to the `Viewport` and will display.

Time for action – creating the navigation Controller and View

1. In the `AppEvents` class, define a new `EventType` named `NavPanelReady`.

```
public static final EventType NavPanelReady = new EventType();
```

2. Create a new class named `NavController` that extends `Controller` in the package `client.mvc.controllers`.

```
public class NavController extends Controller {}
```

3. Create a new class named `NavView` that extends `View` in the package `client.mvc.views`. This view should have a constructor that takes a `NavController` object as a parameter.

```
public class NavView extends View {

  public NavView(NavController navController) {
    super(navController);
  }
}
```

4. In the constructor of `NavController`, register the `Init EventType`.

```
public NavController() {
  registerEventTypes(AppEvents.Init);
}
```

5. Define a field for a `NavView` instance and implement the initialize method to create a new instance of `NavView` with the `NavController` as the `Controller`.

```
private NavView navView;

@Override
public void initialize() {
  super.initialize();
  navView = new NavView(this);
}
```

6. Implement the `handleEvent` method so that all events are forwarded to the `NavView`.

```
@Override
public void handleEvent(AppEvent event) {
  forwardToView(navView, event);
}
```

7. Rename the existing `RssNavigationPanel` class to `NavPanel`. Then create a new instance of the `NavPanel` in the `NavView` class.

```
private final NavPanel navPanel = new NavPanel();
```

8. Implement the `handleEvent` method so that if an event of the `EventType Init` is received, an `AppEvent` of the `EventType NavPanelReady` is dispatched with the `NavPanel` instance as the event's data.

```
@Override
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
```

```
  if (eventType.equals(AppEvents.Init)) {
    Dispatcher.forwardEvent(new AppEvent(AppEvents.NavPanelReady,
    navPanel));
  }
}
```

9. In the constructor of the `AppController` class, register the `NavPanelReady EventType`.

```
public AppController() {
  registerEventTypes(AppEvents.Init);
  registerEventTypes(AppEvents.Error);
  registerEventTypes(AppEvents.UIReady);
  registerEventTypes(AppEvents.NavPanelReady);
}
```

10. In the `AppView` class, create a new method named `onNavPanelReady`. This will retrieve the `Component`, in this case, the `NavPanel` from the data payload of the event and add it to the `Viewport`.

```
private void onNavPanelReady(AppEvent event) {
  BorderLayoutData westData = new BorderLayoutData(LayoutRegion.
WEST,
        200, 150, 300);
  westData.setCollapsible(true);
  westData.setSplit(true);
  Component component = event.getData();
  viewport.add(component, westData);
}
```

11. In the `handleEvent` method of the `AppView`, add a condition to call the `onNavPanelReady` method if an `AppEvent` of the `NavPanelReady EventType` is received.

```
@Override
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.Init)) {
    onInit(event);
  } else if (eventType.equals(AppEvents.Error)) {
    onError(event);
  } else if (eventType.equals(AppEvents.UIReady)) {
    onUIReady(event);
  } else if (eventType.equals(AppEvents.NavPanelReady)) {
    onNavPanelReady(event);
  }
}
```

12. Finally, in the `onModuleLoad` method of the `RSSReader EntryPoint` class, add a new instance of the `NavController` to the dispatcher, taking care to add it after the `AppController`, as we want `AppController` to receive the `Init AppEvent` first.

```
public void onModuleLoad() {
  final FeedServiceAsync feedService =
    GWT.create(FeedService.class);
  Registry.register(RSSReaderConstants.FEED_SERVICE, feedService);
  Dispatcher dispatcher = Dispatcher.get();
  dispatcher.addController(new AppController());
  dispatcher.addController(new NavController());
  dispatcher.dispatch(AppEvents.Init);
  dispatcher.dispatch(AppEvents.UIReady);
}
```

13. Start the application and the `NavPanel` will now be visible.

What just happened?

We created a `Controller` and a `View` for the `NavPanel` component, making it completely decoupled from the rest of the application. When the `NavPanel` had been created, this was announced using an `AppEvent` of an `EventType` the `AppController` had registered to observe. This `AppEvent` was forwarded to the `AppView`, which was able to handle the event by adding the `NavPanel` to the `Viewport`.

We will now do almost exactly the same for the `FeedPanel` component.

Time for action – creating the FeedPanel Controller and View

1. In the `AppEvents` class, define a new `EventType` named `FeedPanelReady`.

```
public static final EventType FeedPanelReady = new EventType();
```

2. Create a new class named `FeedController` that extends `Controller` in the package `client.mvc.controllers`.

```
public class FeedController extends Controller {}
```

3. Create a new class named `FeedView` that extends `View` in the package `client.mvc.views`. This view should have a constructor that takes a `FeedController` instance as a parameter.

```
public FeedView(FeedController feedController) {
	super(feedController);
}
```

4. In the `FeedController` constructor, register the `Init EventType`.

```
public FeedController() {
	registerEventTypes(AppEvents.Init);
}
```

5. Define a field for a `FeedView` instance and implement the initialize method to create a new instance of `FeedView` with the `FeedController` as the `Controller`.

```
private FeedView feedView;

@Override
public void initialize() {
	super.initialize();
	feedView = new FeedView (this);
}
```

6. Implement the `handleEvent` method so that all events are forwarded to the `FeedView`.

```
@Override
public void handleEvent(AppEvent event) {
	forwardToView(feedView, event);
}
```

7. Rename the existing `RssMainPanel` class to `FeedPanel`. Then create a new instance of the `FeedPanel` in the `FeedView` class.

```
private final FeedPanel feedPanel = new FeedPanel();
```

8. In the `FeedView`, create a method named `onInit` that dispatches an `AppEvent` of the type `FeedPanelReady` with the `FeedPanel` instance as the event's data.

```
private void onInit(AppEvent event) {
       Dispatcher.forwardEvent(new AppEvent(AppEvents.
FeedPanelReady, feedPanel));
}
```

9. Implement the `handleEvent` method so that if an event of the `Init EventType` is received, the `onInit` method is called.

```
@Override
protected void handleEvent(AppEvent event) {
       EventType eventType = event.getType();
       if (eventType.equals(AppEvents.Init)) {
              onInit(event);
       }
}
```

10. In the constructor of the `AppController` class, register the `FeedPanelReady` `EventType`.

```
public AppController() {
       registerEventTypes(AppEvents.Init);
       registerEventTypes(AppEvents.Error);
       registerEventTypes(AppEvents.UIReady);
       registerEventTypes(AppEvents.NavPanelReady);
       registerEventTypes(AppEvents.FeedPanelReady);
}
```

11. In the `AppView` class, create a new method named `onFeedPanelReady`. This will retrieve the `Component`, in this case, the `FeedPanel` from the data payload of the event and add it to the `Viewport`.

```
private void onFeedPanelReady(AppEvent event) {
       RowData rowData = new RowData();
       rowData.setHeight(.5);
       Component component = event.getData();
       mainPanel.add(component, rowData);
}
```

12. In the `handleEvent` method of the `AppView`, add a condition to call the `onFeedPanelReady` method if an `AppEvent` of the `NavPanelReady EventType` is received.

```
@Override
protected void handleEvent(AppEvent event) {
        EventType eventType = event.getType();
        if (eventType.equals(AppEvents.Init)) {
                onInit(event);
        } else if (eventType.equals(AppEvents.Error)) {
                onError(event);
        } else if (eventType.equals(AppEvents.UIReady)) {
                onUIReady(event);
        } else if (eventType.equals(AppEvents.NavPanelReady)) {
                onNavPanelReady(event);
        } else if (eventType.equals(AppEvents.FeedPanelReady)) {
                onFeedPanelReady(event);
        }
}
```

13. Finally, in the `onModuleLoad` method of the `RSSReader EntryPoint` class, add a new instance of the `FeedController` to the `Dispatcher`.

```
public void onModuleLoad() {
        final FeedServiceAsync feedService = GWT.create(FeedService.
                                              class);
        Registry.register(RSSReaderConstants.FEED_SERVICE,
                        feedService);
        Dispatcher dispatcher = Dispatcher.get();
        dispatcher.addController(new AppController());
        dispatcher.addController(new NavController());
        dispatcher.addController(new FeedController());
        dispatcher.dispatch(AppEvents.Init);
        dispatcher.dispatch(AppEvents.UIReady);
}
```

14. Start the application and the `FeedPanel` will now be visible.

What just happened?

As with the `NavPanel`, we created a `Controller` and `View` for the `FeedPanel` component, and using the MVC mechanisms, allowed it to be added to the UI.

Have a go hero – creating the item Controller and View

We have just created two very similar `Controller` and `View` pairs for the `NavPanel` and `FeedPanel` components. Now we need the same thing for the `ItemPanel` component. Create an `ItemPanelReady` `EventType`, an `ItemController`, and an `ItemView`. Then register the `AppController` to handle the `ItemPanelReady` method and the `AppView` to add the `ItemPanel` to the `Viewport` as a second row of the main panel.

The objective is to have the example application showing all the three components as follows:

Solution:

AppEvents class:

```
public class AppEvents {
  public static final EventType Init = new EventType();
  public static final EventType Error = new EventType();

  public static final EventType UIReady = new EventType();

  public static final EventType NavPanelReady = new EventType();
  public static final EventType FeedPanelReady = new EventType();
  public static final EventType ItemPanelReady = new EventType();
}
```

AppController class:

```
public class AppController extends Controller {

  private AppView appView;

  public AppController() {
    registerEventTypes(AppEvents.Init);
    registerEventTypes(AppEvents.Error);
    registerEventTypes(AppEvents.UIReady);
    registerEventTypes(AppEvents.NavPanelReady);
    registerEventTypes(AppEvents.FeedPanelReady);
    registerEventTypes(AppEvents.ItemPanelReady);
  }
}
```

RSSReader class:

```
public class RSSReader implements EntryPoint {

  public void onModuleLoad() {
    final FeedServiceAsync feedService = GWT.create(FeedService.
                                         class);
    Registry.register(RSSReaderConstants.FEED_SERVICE, feedService);
    Dispatcher dispatcher = Dispatcher.get();
    dispatcher.addController(new AppController());
    dispatcher.addController(new NavController());
    dispatcher.addController(new FeedController());
    dispatcher.addController(new ItemController());
    dispatcher.dispatch(AppEvents.Init);
    dispatcher.dispatch(AppEvents.UIReady);
  }
}
```

ItemController class:

```
public class ItemController extends Controller {

  private ItemView itemView;

  public ItemController() {
    registerEventTypes(AppEvents.Init);
  }

  @Override
  public void handleEvent(AppEvent event) {
```

```
    forwardToView(itemView, event);
  }

  @Override
  public void initialize() {
    super.initialize();
    itemView = new ItemView(this);
  }
}
```

ItemView class:

```
public class ItemView extends View {

  private final ItemPanel itemPanel = new ItemPanel();

  public ItemView(ItemController itemController) {
    super(itemController);
  }

  @Override
  protected void handleEvent(AppEvent event) {
    EventType eventType = event.getType();
    if (eventType.equals(AppEvents.Init)) {
      Dispatcher.forwardEvent(new AppEvent(AppEvents.ItemPanelReady,
itemPanel));
    }
  }
}
```

AppView class:

```
public class AppView extends View {

  @Override
  protected void handleEvent(AppEvent event) {
    EventType eventType = event.getType();
    if (eventType.equals(AppEvents.Init)) {
      onInit(event);
    } else if (eventType.equals(AppEvents.Error)) {
      onError(event);
    } else if (eventType.equals(AppEvents.UIReady)) {
      onUIReady(event);
    } else if (eventType.equals(AppEvents.NavPanelReady)) {
      onNavPanelReady(event);
    } else if (eventType.equals(AppEvents.FeedPanelReady)) {
```

```
        onFeedPanelReady(event);
      } else if (eventType.equals(AppEvents.ItemPanelReady)) {
        onItemPanelReady(event);
    }

    private void onItemPanelReady(AppEvent event) {
      RowData rowData = new RowData();
      rowData.setHeight(.5);
      Component component = event.getData();
      mainPanel.add(component, rowData);
    }
```

Allowing viewing of multiple feeds

Previously, we had only displayed one feed by displaying a single `ItemGrid` in the `RssMainPanel`. Now we are going to use a `TabPanel` to manage multiple `TabItem` objects, each using an `ItemGrid` to display the items of a feed.

Time for action – adding tabs

1. In the `FeedPanel` class, create a new `TabPanel` field.

```
private final TabPanel tabPanel = new TabPanel();
```

2. Create a new public method named addTab that takes a `TabItem` as an argument. Set the `Layout` to `FitLayout`, the icon to the RSS icon we defined previously and the scroll mode to auto so that scroll bars appear if necessary.

```
public void addTab(TabItem tabItem) {
  tabItem.setLayout(new FitLayout());
  tabItem.setIcon(Resources.ICONS.rss());
  tabItem.setScrollMode(Scroll.AUTO);
}
```

3. We only want one `TabItem` for each feed, so if a feed already had a `TabItem` on the `TabPanel`, we want to switch to that; otherwise switch to the existing one.

```
public void addTab(TabItem tabItem) {
        tabItem.setLayout(new FitLayout());
        tabItem.setIcon(Resources.ICONS.rss());
        tabItem.setScrollMode(Scroll.AUTO);
        String tabId = tabItem.getId();
        TabItem existingTab = tabPanel.findItem(tabId, false);
        if (existingTab == null) {
              tabPanel.add(tabItem);
```

```
            tabPanel.setSelection(tabItem);
        } else {
            tabPanel.setSelection(existingTab);
        }
}
```

4. Remove everything from the constructor apart from the `setHeading` and `setLayout` calls and then add the `TabPanel` to the underlying `ContentPanel`.

```
public FeedPanel() {
        setHeading("Main");
        setLayout(new FitLayout());
add(tabPanel);
}
```

What just happened?

We added a `TabPanel` to the `FeedPanel`. This means that we can now display multiple feeds on each of the `TabItem` objects in the `TabPanel`.

Wiring it together

We have all the components on the UI. Now we need to get them to respond to selections of feeds and items appropriately.

We can pass the `ModelData` items that are selected in the different components in the same way that we passed the components in the data payload of events.

- When a user selects a `Feed` in the `FeedList`, a `FeedSelected` `AppEvent` is dispatched with the selected `Feed` as the data.
- When the `FeedSelected` `AppEvent` is dispatched, the `FeedView` creates or switches to a tab displaying the items of the feed in an `ItemGrid`.
- When a user selects an `Item` in the `ItemGrid`, an `ItemSelected` `AppEvent` is dispatched with the selected `Item` as the data.
- When the `ItemSelected` `AppEvent` is dispatched, the `ItemView` renders the item in the `ItemPanel`.
- When a tab is selected by the user, the `TabSelected` `AppEvent` is dispatched with the `Feed` the tab is displaying as data.
- When the `TabSelected` `AppEvent` is dispatched, the `FeedList` will select the appropriate `Feed`.

Time for action – responding to selections

1. In the `AppEvents` class, define the three new events.

```
public static final EventType FeedSelected = new EventType();
public static final EventType ItemSelected = new EventType();
```

2. In the `onRender` method of the `FeedList` class, create a **SelectionChange** `Listener` so that it forwards a `FeedSelected` `AppEvent` with the selected `Feed` attached using the `Dispatcher`.

```
feedList.addSelectionChangedListener(new SelectionChangedListener<
BeanModel>() {
  @Override
  public void selectionChanged(SelectionChangedEvent<BeanModel>
se)   {
    Feed feed = se.getSelectedItem().getBean();
    if (feed != null) {
      Dispatcher.forwardEvent(AppEvents.FeedSelected, feed);
    }
  }
});
```

3. Register the `FeedSelected` `EventType` in the `FeedController`.

```
public FeedController() {
  registerEventTypes(AppEvents.Init);
  registerEventTypes(AppEvents.FeedSelected);
}
```

4. In the `ItemGrid` class define a new `Feed` field and modify the constructor so that it takes a `Feed` as a parameter and uses that to set the `Feed` field.

```
private final Feed feed;

public ItemGrid(Feed feed) {
       setLayout(new FitLayout());
       this.feed = feed;
}
```

5. Create a new field for the `Grid` and use this in place of the `Grid` in the `onRender` method.

```
private Grid<ModelData> grid;
@Override
protected void onRender(Element parent, int index) {
…
grid = new Grid<ModelData>(itemStore, columnModel);
}
```

6. In the `onRender` method remove the `TEST_DATA_FILE` constant and in the call to the loadItems method of the FeedService replace the reference to `TEST_DATA_FILE` with the `UUID` of the `Feed` object.

```
RpcProxy<List<Item>> proxy = new RpcProxy<List<Item>>() {
       @Override
       protected void load(Object loadConfig,
                    AsyncCallback<List<Item>> callback) {
              feedService.loadItems(feed.getUuid(), callback);
       }
};
```

7. Again in the `ItemGrid` class define a new `resetSelection` method that resets the selection of the underlying `Grid`.

```
public void resetSelection() {

       grid.getSelectionModel().deselectAll();

}
```

8. In the `FeedView` class, create an `onFeedSelected` event that creates a new `ItemGrid` using the `Feed` object extracted from the event and wrap it in a `TabItem` and add the `TabItem` to the `FeedPanel`.

```
private void onFeedSelected(AppEvent event) {
  Feed feed = event.getData();
  final ItemGrid itemGrid = new ItemGrid(feed);
  TabItem tabItem = new TabItem(feed.getTitle());
  tabItem.setId(feed.getUuid());
  tabItem.setData("feed", feed);
  tabItem.add(itemGrid);
  tabItem.addListener(Events.Select, new Listener<TabPanelEvent>()
{
    @Override
    public void handleEvent(TabPanelEvent be) {
      itemExpanderGrid.resetSelection();
    }
  });
  tabItem.setClosable(true);
  feedPanel.addTab(tabItem);
}
```

9. Modify the `handleEvent` method so that when a `FeedSelected` `AppEvent` is received, the `onFeedSelected` method is called.

```
@Override
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.Init)) {
    onInit(event);
  } else if (eventType.equals(AppEvents.FeedSelected)) {
    onFeedSelected(event);
  }
}
```

10. In the `onRender` method of the `ItemGrid` class, create a `SelectionChange Listener` so that it forwards a `ItemSelected` `AppEvent` with the selected `Item` attached using the `Dispatcher`.

```
grid.getSelectionModel().addListener(Events.SelectionChange,
  new Listener<SelectionChangedEvent<Item>>() {
    public void handleEvent(SelectionChangedEvent<Item> be) {
      Item item = (Item) be.getSelection().get(0);
      Dispatcher.forwardEvent(AppEvents.ItemSelected, item);
      }
    });
```

11. Register the `ItemSelected` `EventType` in the `ItemController`.

```
public ItemController() {
  registerEventTypes(AppEvents.Init);
  registerEventTypes(AppEvents.ItemSelected);
}
```

12. In the `ItemView` class, create a method named `onItemSelected` that displays the `Item` from the `ItemSelected` `AppEvent` in the `ItemPanel`.

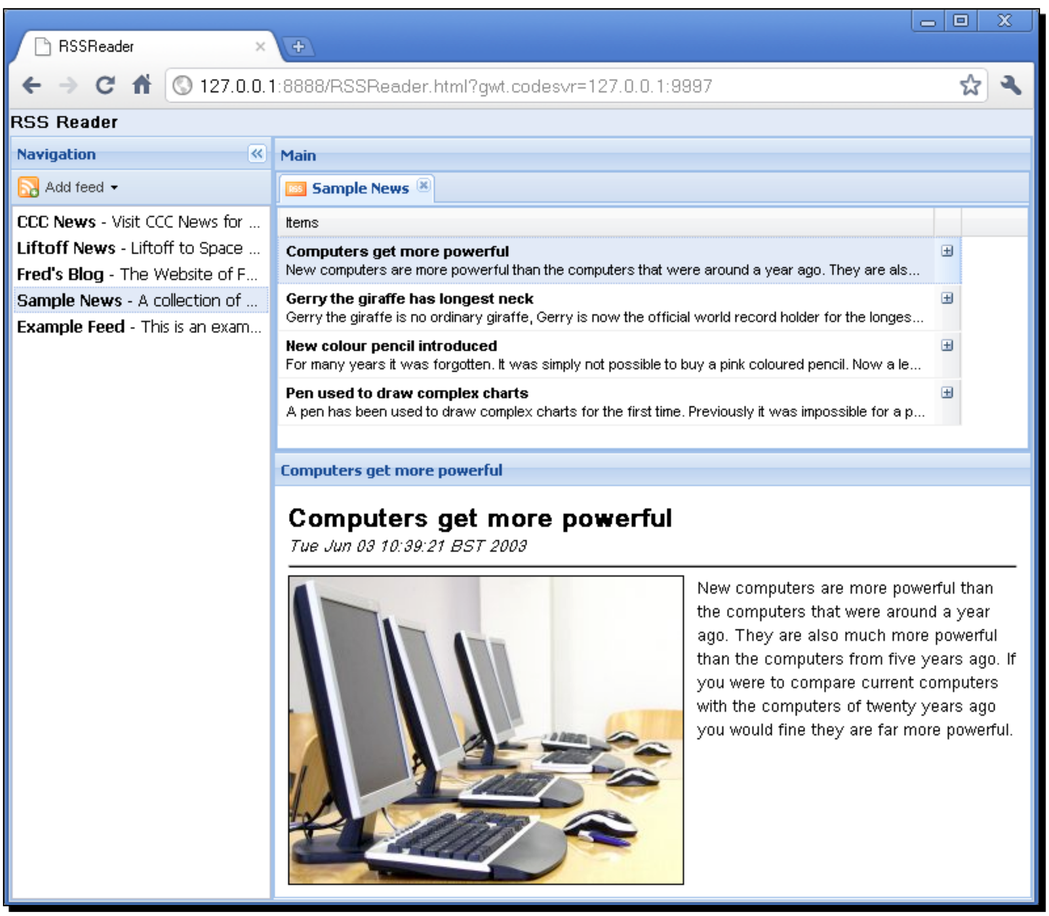
```
private void onItemSelected(AppEvent event) {
  Item item = (Item) event.getData();
  itemPanel.displayItem(item);
}
```

13. Modify the `handleEvent` method so that when an `ItemSelected` `AppEvent` is received, the `onItemSelected` method is called.

```
@Override
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.In it)) {
```

```
        Dispatcher.forwardEvent(new AppEvent(AppEvents.ItemPanelReady,
itemPanel));
  } else if (eventType.equals(AppEvents.ItemSelected)) {
    onItemSelected(event);
  }
}
```

14. Start the application, select a feed, then an item, and see how the `FeedPanel` and `ItemPanel` components update respectively. Please refer to the following screenshot:



What just happened?

We have created events to pass selections between the `NavPanel`, `FeedPanel`, and `ItemPanel` components. We can now select feeds and items and have the components updated automatically.

Keeping things in sync

We now need to make sure the list of feeds updates correctly when a user adds a new feed. We also need to make sure that the correct feed is shown as selected in the list when a user selects a feed tab.

To do this we will create events that will fire when a feed is added and a tab is selected and make the feed list respond appropriately.

Time for action – responding to a Feed being added

1. In the `AppEvents` class, define two new `EventType` object named `FeedAdded` and `TabSelected` respectively

```
public static final EventType TabSelected = new EventType();
public static final EventType FeedAdded = new EventType();
```

2. In the constructor of the `NavController` class, register the `TabSelected` `EventType` and

```
public NavController() {
       registerEventTypes(AppEvents.Init);
       registerEventTypes(AppEvents.FeedAdded);
       registerEventTypes(AppEvents.TabSelected);
}
```

3. In the `FeedList` class take the `ListField` and the `ListLoader` from the `onRender` method and redefine them as fields.

```
private final ListField<BeanModel> feedList = new
ListField<BeanModel>();
private ListLoader<ListLoadResult<BeanModel>> loader;
```

4. Define a new method named `reloadFeeds` that calls the `load` method of the `Loader`. This will reload the `Feed` objects into the `Store`.

```
public void reloadFeeds() {
       loader.load();
}
```

5. Define a second new method named `selectFeed` that takes a `Feed` object and uses it to select the appropriate entry in the `ListField`.

```
public void selectFeed(Feed feed)
{
	BeanModelFactory beanModelFactory = BeanModelLookup.get().
getFactory(feed.getClass());
	feedList.setSelection(Arrays.asList(beanModelFactory.
createModel(feed)));
}
```

6. In the `NavPanel` make the FeedList defined in the constructor into a feed.

```
private FeedList feedList = new FeedList();

	public NavPanel() {
		setHeading("Navigation");
		setLayout(new FitLayout());
		initToolbar();
		add(feedList);
	}
```

7. Define `selectFeed` and `reloadFeeds` methods that expose the methods of the same name in the `FeedList`.

```
public void reloadFeeds()
{
	feedList.reloadFeeds();
}

public void selectFeed(Feed feed)
{
	feedList.selectFeed(feed);
}
```

8. In the `NavView` class create an `onTabSelected` method that extracts the `Feed` from an `AppEvent` and uses it to call the `selectFeed` event or the `NavPanel`.

```
private void onTabSelected(AppEvent event) {
	Feed feed = (Feed) event.getData();
	navPanel.selectFeed(feed);
}
```

9. Again in the `NavView` class create an `onFeedAdded` method that calls the `reloadFeeds` method of the `NavPanel`.

```
private void onFeedAdded(AppEvent event) {
	navPanel.reloadFeeds();
}
```

10. Now modify the `handleEvent` method call the `onTabSelected` and `onFeedAdded` methods in response to the `TabSelected` and `FeedAdded` `EventType` respectively.

```
@Override
protected void handleEvent(AppEvent event) {
       EventType eventType = event.getType();
       if (eventType.equals(AppEvents.Init)) {
              Dispatcher.forwardEvent(new AppEvent(AppEvents.
NavPanelReady,
                            navPanel));
       } else if (eventType.equals(AppEvents.TabSelected)) {
              onTabSelected(event);
       } else if (eventType.equals(AppEvents.FeedAdded)) {
              onFeedAdded(event);
       }
}
```

11. In the `addFeed` method of `LinkFeedPopup` class there is a called to the `addExistingFeed` method of the `FeedService`. In the `onSuccess` method use the `Dispatcher` to forward a `FeedAdded` `AppEvent`.

```
@Override
public void onSuccess(Void result) {
       tfUrl.clear();
       Info.display("RSS Reader", "Feed at " + FeedUrl + " added
successfully");
       Dispatcher.forwardEvent(AppEvents.FeedAdded);
       hide();
}
```

12. Similarly in the `save` method of the `FeedForm` class, in the `onSuccess` method of the call to the `saveFeed` method of the `FeedService`, again forward a `FeedAdded` `AppEvent`.

```
@Override
public void onSuccess(Void result) {
       Info.display("RSS Reader", "Feed " + feed.getTitle() + "
saved sucessfully");
       Dispatcher.forwardEvent(AppEvents.FeedAdded);
}
```

13. Finally in the `onFeedSelected` method of the `FeedView` class forward a `TabSelected` `AppEvent` in the existing `Listener`.

```
tabItem.addListener(Events.Select, new Listener<TabPanelEvent>() {
       @Override
```

```
            public void handleEvent(TabPanelEvent be) {
                  itemGrid.resetSelection();
                  Dispatcher.forwardEvent(new AppEvent(AppEvents.
                    TabSelected, feed));
            }
      });
```

What just happened

We have made the FeedList automatically respond to a new feed being added by refreshing the list. The selected item of the list will also update in response to an open tab being selected.

An `AppEvent` does not just have to be consumed by one `Controller`, it can be consumed by multiple controllers.

For example, we want to add in a `StatusToolbar` component to provide the user with feedback on what is happening in the application. We can use a `Controller` and `View` to make that happen.

We want our `StatusController` to report when:

- A Feed is selected
- An Item is selected

Time for action – creating a status toolbar Controller and View

1. In the `AppEvents` class, define a new `EventType` named `StatusToolbarReady`.

```
public static final EventType StatusToolbarReady = new
EventType();
```

2. Create a new class named `StatusController` that extends `Controller` in the package `client.mvc.controllers` and register it to observe the `Init`, `Error`, `UIReady`, `FeedSelected`, and `ItemSelected` events.

```
public class StatusController extends Controller {
{
      public StatusController() {
            registerEventTypes(AppEvents.Init);
            registerEventTypes(AppEvents.Error);
            registerEventTypes(AppEvents.UIReady);
            registerEventTypes(AppEvents.FeedSelected);
            registerEventTypes(AppEvents.ItemSelected);
      }
}
```

3. Create a new class named `StatusView` that extends `View` in the package `client.mvc.views`. This `View` should have a constructor that takes a `StatusController` instance as a parameter.

```
public class StatusView extends View {

       public StatusView(StatusController statusController) {
              super(statusController);
       }
}
```

4. Define two new fields one for a `Status` object, a second for a `ToolBar` and define a `setStatus` method that takes a String and uses it to set the text of the `Status` object.

```
private final Status status = new Status();
private final ToolBar toolBar = new ToolBar();

public void setStatus(String message) {
       status.setText(message);
}
```

5. Create a new `onInit` method that sets up the `Status` object, adds it to the ToolBar and then forwards a `StatusToolbarReady` `AppEvent` with the `ToolBar` attached.

```
private void onInit() {

       status.setWidth("100%");
       status.setBox(true);
       toolBar.add(status);
       Dispatcher.forwardEvent(new AppEvent(AppEvents.
         StatusToolbarReady,toolBar));
}
```

6. Implement the `handleEvent` method to call the onInit method in response to an `Init` `AppEvent` and then call `setStatus` method to display "Init" in the `ToolBar`

```
@Override
protected void handleEvent(AppEvent event) {
       EventType eventType = event.getType();
       if (eventType.equals(AppEvents.Init)) {
              onInit();
              setStatus("Init");
       }
}
```

7. In the constructor of the `AppController` class, register the `StatusPanelReady` `EventType`.

```
public AppController() {
       registerEventTypes(AppEvents.Init);
       registerEventTypes(AppEvents.Error);
       registerEventTypes(AppEvents.UIReady);
       registerEventTypes(AppEvents.NavPanelReady);
       registerEventTypes(AppEvents.FeedPanelReady);
       registerEventTypes(AppEvents.ItemPanelReady);
       registerEventTypes(AppEvents.StatusToolbarReady);
}
```

8. In the `AppView` class, create a new method named `onStatusToolbarReady` that adds the `StatusToolbar` contained in the `AppEvent` as the bottom component to the main `ContentPanel`.

```
private void onStatusToolbarReady(AppEvent event) {
       Component component = event.getData();
       mainPanel.setBottomComponent(component);
}
```

9. Modify the `handleEvent` method so that when an `AppEvent` of the `StatusToolbarReady` `EventType` is received, the `onStatusToolbarReady` method is called.

```
@Override
protected void handleEvent(AppEvent event) {
       EventType eventType = event.getType();
       if (eventType.equals(AppEvents.Init)) {
             onInit(event);
       } else if (eventType.equals(AppEvents.ItemPanelReady)) {
             onItemPanelReady(event);
       } else if (eventType.equals(AppEvents.StatusToolbarReady)) {
             onStatusToolbarReady(event);
       }
}
```

10. In the `onModuleLoad` method of the `RSSReader` `EntryPoint` class, add a new instance of the `StatusController`.

```
public void onModuleLoad() {
       Registry.register(RSSReaderConstants.FEED_SERVICE, GWT.
                   create(FeedService.class));
       Dispatcher dispatcher = Dispatcher.get();
       dispatcher.addController(new AppController());
       dispatcher.addController(new NavController());
```

```
        dispatcher.addController(new FeedController());
        dispatcher.addController(new ItemController());
        dispatcher.addController(new StatusController());
        dispatcher.dispatch(AppEvents.Init);
        dispatcher.dispatch(AppEvents.UIReady);
}
```

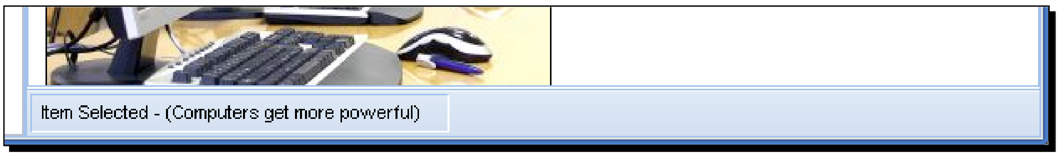
11. Returning to the `handleEvent` method of the `StatusView`, handle a `FeedSelected` `AppEvent` by extracting the `Feed` object from the `AppEvent` and displaying the `Feed` name.

```
@Override
protected void handleEvent(AppEvent event) {
        EventType eventType = event.getType();
        if (eventType.equals(AppEvents.Init)) {
                onInit();
                setStatus("Init");
        } else if (eventType.equals(AppEvents.FeedSelected)) {
                Feed feed = event.getData();
                setStatus("Feed Selected - (" + feed.getTitle() +
                  ")");
        }
}
```

12. Similarly, extract the `Item` object from the `AppEvent` and display the `Item` name when an `ItemSelected` `AppEvent` is received.

```
@Override
protected void handleEvent(AppEvent event) {
        EventType eventType = event.getType();
        if (eventType.equals(AppEvents.Init)) {
                onInit();
                setStatus("Init");
        } else if (eventType.equals(AppEvents.FeedSelected)) {
                Feed feed = event.getData();
                setStatus("Feed Selected - (" + feed.getTitle() +
                   ")");
        } else if (eventType.equals(AppEvents.ItemSelected)) {
                Item item = event.getData();
                setStatus("Item Selected - (" + item.getTitle() +
                   ")");
        }
}
```

13. Run the application and you will now see a message appear in the `StatusToolbar` when a `Feed` or `Item` is selected.



What just happened?

We created an additional `Controller` that monitored events and reported the status to the user. This showed us how `AppEvents` can be observed by multiple controllers.

Summary

In this chapter, we saw how the GXT framework can allow us to uncouple the different components of the application, and instead of being dependent on each other, they can just respond to events.

In the next chapter, we will take the concept of independent components further, by looking at the portal and the drag-and-drop functionality of GXT.

8

Portal and Drag-and-Drop

This chapter covers the portal and drag-and-drop features of GXT. We will start by learning how to use the `Portal` *layout and* `Portlet` *and then move on to making use of GXT's drag-and-drop features in a practical way.*

Specifically, we will cover the following topics:

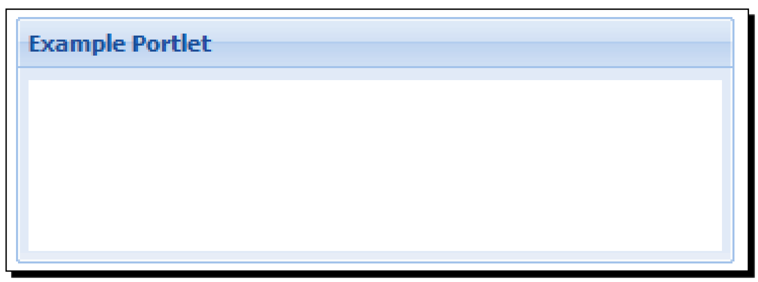
- `Portal`
- `Portlet`
- `Draggable`
- `DragSource`
    - `GridDragSource`
    - `ListViewDragSource`
    - `TreeGridDragSource`
    - `TreePanelDragSource`
- `DropTarget`
    - `GridDropTarget`
    - `ListViewDropTarget`
    - `TreeGridDropTarget`
    - `TreePanelDropTarget`
- `ColumnLayout`
- `RowLayout`

Portlet class

The `Portlet` class extends `ContentPanel` to provide a special type of panel that can be repositioned in the `Viewport` by the user with a `Portal` container. It may appear similar to a window in a desktop application. Creating a `Portlet` is similar to creating other containers. This code:

```
Portlet portlet = new Portlet();
portlet.setHeight(150);
portlet.setHeading("Example Portlet");
```

creates a Portlet like this:



A `Portlet` can be excluded from being repositioned by pinning it using:

```
portal.setPinned(true);
```

Apart from that, a `Portlet` inherits all the features of a standard `ContentPanel`.

The Portal class

A `Portal` is a special container for `Portlet` components. In fact, it is a `Container` containing a collection of `LayoutContainer` components arranged using `ColumnLayout`. Each of those `LayoutContainer` components in turn is able to contain `Portlet` components, arranged using a `RowLayout`.

`Portal` also supports dragging and dropping of `Portlet` components, both in terms of changing the row it is in within a column and the column within the `Portal`.

When creating a `Portal`, we need to set the number of columns the `Portal` should create in the constructor. We also need to set the widths of each column before using the `setColumnWidth` method of the `Portal`.

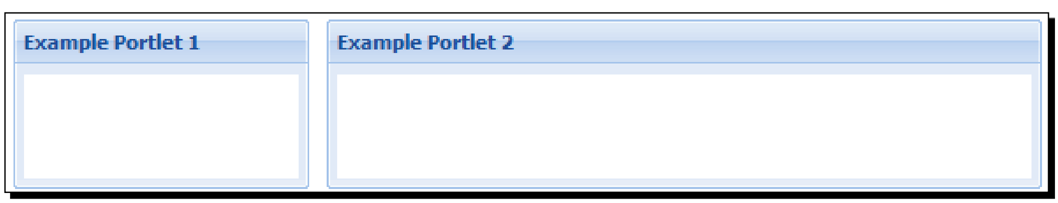
So to create a `Portal` with two columns, (one using 30 percent of the width and the second 70 percent) we would define it as follows:

```
Portal portal = new Portal(2);
portal.setColumnWidth(0, 0.3);
portal.setColumnWidth(1, 0.7);
```

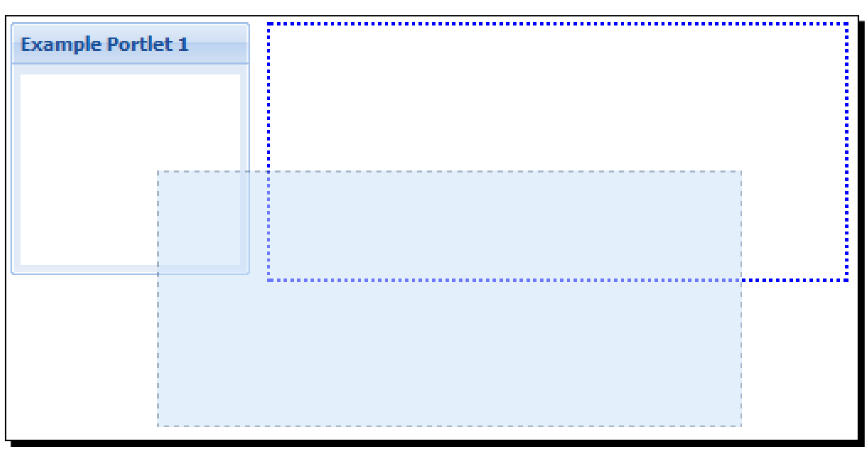
We can then add a `Portlet` to each column like this:

```
Portlet portlet1 = new Portlet();
portlet1.setHeight(150);
portlet1.setHeading("Example Portlet 1");
portal.add(portlet1, 0);

Portlet portlet2 = new Portlet();
portlet2.setHeight(150);
portlet2.setHeading("Example Portlet 2");
portal.add(portlet2, 1);
```

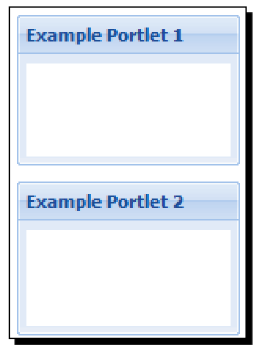
This will produce the following output:



Both `Portlet` components can be dragged and dropped into different positions. The `Portlet` turns into a blue box while being dragged as shown in the following screenshot:



A `Portlet` will automatically resize and fit into the column in which it is dropped, as seen in the next screenshot:

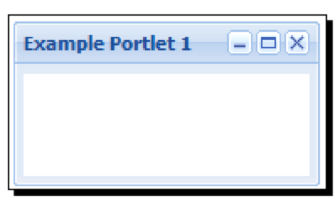


ToolButton

Like `ContentPanel` that `Portlet` extends, we can add `ToolButton` components to the header. These can be very useful for making a `Portlet` look and behave even more like windows in a desktop application.

```
portlet.getHeader().addTool(new ToolButton("x-tool-minimize"));
portlet.getHeader().addTool(new ToolButton("x-tool-maximize"));
portlet.getHeader().addTool(new ToolButton("x-tool-close"));
```

The output can be seen as shown in the following screenshot:



At the moment, we are using `ContentPanel` components in our example application and laying them out using a `BorderLayout`. We shall now see that it does not take much to change the `ContentPanel` components into `Portlet` components and manage them using a `Portal`.

`Portlet` components are ideally suited to being independent, self-contained user interface elements that respond to the data passed to them. Rather than tying them into a `Portal` directly, we can use the MVC components to cause the `Portal` to respond to the creation of a new `Portlet` to preserve that independence.

Time for action – creating a Portal Controller and a Portlet View

1. The first thing we need to do is add a new `EventType` to the existing `AppEvents` class named `NewPortletCreated`. We will fire this when we create a new `Portlet`.

```
public static final EventType NewPortletCreated = new EventType();
```

2. Create a new class named `PortalController` that extends `Controller`.

```
public class PortalController extends Controller {
```

3. Create a new class named `PortalView` that extends `View`.

```
public class PortalView extends View {
```

4. Create a constructor that sets the `Controller` of the `PortalView`.

```
public PortalView(PortalController portalController) {
  super(portalController);
}
```

5. Returning to `PortalController`, create a variable to hold the `PortalView` and override the initialize method to set the view.

```
private PortalView portalView;

@Override
public void initialize() {
  super.initialize();
portalView = new PortalView(this);
}
```

6. Create a constructor that registers each `EventType` the `PortalController` should observe, specifically `NewPortletCreated` creation and `Error`.

```
public PortalController() {
  registerEventTypes(AppEvents.NewPortletCreated );
  registerEventTypes(AppEvents.Error);
}
```

7. Override the `handleEvent` method to forward any events to the `View` apart from errors which for the time being we will just log to the GWT log.

```
@Override
public void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.error)) {
   GWT.log("Error", (Throwable) event.getData());
  } else {
    forwardToView(portalView, event);
  }
}
```

8. Returning to `PortalView`, create a new portal field consisting of a `Portal` component with two columns.

```
private final Portal portal = new Portal(2);
```

9. Override the `initialize` method to set the width of the two columns, the first to 30 percent of the width of the `Portal` and the second to 70 percent.

```
@Override
protected void initialize() {
  portal.setColumnWidth(0, 0.3);
  portal.setColumnWidth(1, 0.7);
}
```

10. Now create a `Viewport`, set the layout to `FitLayout`, add the `Portal`, and then add the `Viewport` to GWT's `RootPanel`.

```
@Override
protected void initialize() {
  portal.setColumnWidth(0, 0.3);
  portal.setColumnWidth(1, 0.7);

  final Viewport viewport = new Viewport();
  viewport.setLayout(new FitLayout());
  viewport.add(portal);
  RootPanel.get().add(viewport);
}
```

11. We also need to implement the `handleEvent` method of the `View`. For now, we will catch the `NewPortletCreated` event, but we will not do anything with it yet.

```
@Override
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.NewPortletCreated )) {

  }
}
```

12. Finally, go to the `onModuleLoad` method of the `EntryPoint` `RSSReader` class and instead of creating an `AppController`, create a `PortalController`, and remove the line that forwards an `Init` `AppEvent`, as we will not be using it. The `onModuleLoad` method will now look like this:

```
public void onModuleLoad() {
  final FeedServiceAsync feedService =
        GWT.create(FeedService.class);
  Registry.register(RSSReaderConstants.FEED_SERVICE, feedService);
  Dispatcher dispatcher = Dispatcher.get();
  dispatcher.addController(new PortalController());
}
```

What just happened?

We created the basic framework for a `Portal` layout of our application. However, if we started it now, we would just get a blank screen. What we need to do is add `Portlet` components.

The actual `Portlet` components are not too complicated, as most of the work is done by components that we created in the previous chapters. The `Portlet` components will just act as wrappers.

Time for action – creating the Navigation Portlet

1. Create a new class named `NavPortlet` that extends `Portlet`.

```
public class NavPortlet extends Portlet {
```

2. Create a constructor and set the heading, layout, and height of the `Portlet`.

```
public NavPortlet()
{
      setHeading("Navigation");
      setLayout(new FitLayout());
      setHeight(610);
}
```

3. In the `RSSReaderConstants` class, add a new constant to act as the ID for this `Portlet`.

```
public static final String NAV_PORTLET = "navPortlet";
```

4. Back in the constructor of `NavPortlet`, set the ID of the `Portlet` to be the `NAV_PORTLET` constant.

```
public NavPortlet()
{
  setHeading("Navigation");
  setLayout(new FitLayout());
  setHeight(610);
  setId(RSSReaderConstants.NAV_PORTLET);
}
```

5. Now create a new instance of the `NavPanel` class to provide the content of the `Portlet`. As the `Portlet` already has a title, hide the header of the `NavPanel` and add it to the `Portlet`.

```
public NavPortlet() {
  setHeading("Navigation");
  setLayout(new FitLayout());
  setHeight(610);
  setId(RSSReaderConstants.NAV_PORTLET);
  NavPanel navPanel = new NavPanel();
  navPanel.setHeaderVisible(false);
  add(navPanel);
}
```

6. We now need to tell the `Portal` that this new `Portlet` has been created. We will do that by forwarding an `AppEvent` of the `NewPortletCreated` `EventType` with this `Portlet` as the data payload using the `Dispatcher`.

```
public NavPortlet()
{
  setHeading("Navigation");
  setLayout(new FitLayout());
  setHeight(610);
  setId(RSSReaderConstants.NAV_PORTLET);
  NavPanel navPanel = new NavPanel();
  navPanel.setHeaderVisible(false);
  add(navPanel);
  Dispatcher.forwardEvent(AppEvents.NewPortletCreated , this);
}
```

7. Now we have to respond to the NewPortletCreated event in the PortalView. So in PortalView, create a method called onNewPortletCreated and implement it so that if the NavPortlet is contained in the data of the AppEvent, it will be added to the first column of the Portal. All the other Portlet components will be added to the second column.

```
private void onNewPortletCreated (AppEvent event) {
  final Portlet portlet = (Portlet) event.getData();
  if (portlet.getId() == RSSReaderConstants.NAV_PORTLET) {
    portal.add(portlet, 0);
  } else {
    portal.add(portlet, 1);
  }
}
```

8. In the handleEvent method, call the onNewPortletCreated method when an AppEvent with the NewPortletCreated EventType is handled.

```
@Override
protected void handleEvent(AppEvent event) {
  EventType eventType = event.getType();
  if (eventType.equals(AppEvents.NewPortletCreated )) {
    onNewPortletCreated (event);
  }
}
```

9. All we need to do now is go back to the onModuleLoad method of the RSSReader EntryPoint class and create a new instance of NavPortlet and the MVC events will take care of the rest.

```
public void onModuleLoad() {
  final FeedServiceAsync feedService =
        GWT.create(FeedService.class);
  Registry.register(RSSReaderConstants.FEED_SERVICE, feedService);
  Dispatcher dispatcher = Dispatcher.get();
  dispatcher.addController(new PortalController());
  new NavPortlet();
}
```

10. Finally, start the application and you will see a `Portlet` complete with a list of feeds.

What just happened?

We have created a new navigation `Portlet` and constructed the framework to automatically add it to the `Portal`. With this in place, it is now straightforward to create two more portlets, one for displaying feeds and one for displaying items.

Time for action – creating more portlets

1. Create two new constants in `RSSReaderConstants` for the two new `Portlet` components we are going to create, namely, `FEED_PORTLET` and `ITEM_PORTLET`.

```
public static final String FEED_PORTLET = "feedPortlet";
public static final String ITEM_PORTLET = "itemPortlet";
```

2. Create a new class named `FeedPortlet`, extending `Portlet`, and build a constructor in the same way as we did with `NavPortlet`, this time setting the ID of the `Portlet` to the `FEED_PORTLET` constant.

```
public FeedPortlet() {
setHeading("Feed");
  setLayout(new FitLayout());
  setHeight(350);
  setId(RSSReaderConstants.FEED_PORTLET);
}
```

3. Create a new `FeedPanel` field, set its header to invisible in the constructor of the `FeedPortlet`, and add it to the underlying `Portlet`.

```
private final FeedPanel feedPanel = new FeedPanel();

public FeedPortlet()
{
  setHeading("Feed");
  setLayout(new FitLayout());
  setHeight(350);
  setId(RSSReaderConstants.FEED_PORTLET);
  feedPanel.setHeaderVisible(false);
  add(feedPanel);
}
```

4. As before, tell the Portal about this new Portlet by forwarding an `AppEvent` of the `NewPortletCreated` `EventType` with the `Portlet` as the data payload, using the `Dispatcher`.

```
public FeedPortlet()
{
  setHeading("Feed");
  setLayout(new FitLayout());
  setHeight(350);
  setId(RSSReaderConstants.FEED_PORTLET);
  feedPanel.setHeaderVisible(false);
  add(feedPanel);
  Dispatcher.forwardEvent(AppEvents.NewPortletCreated, this);
}
```

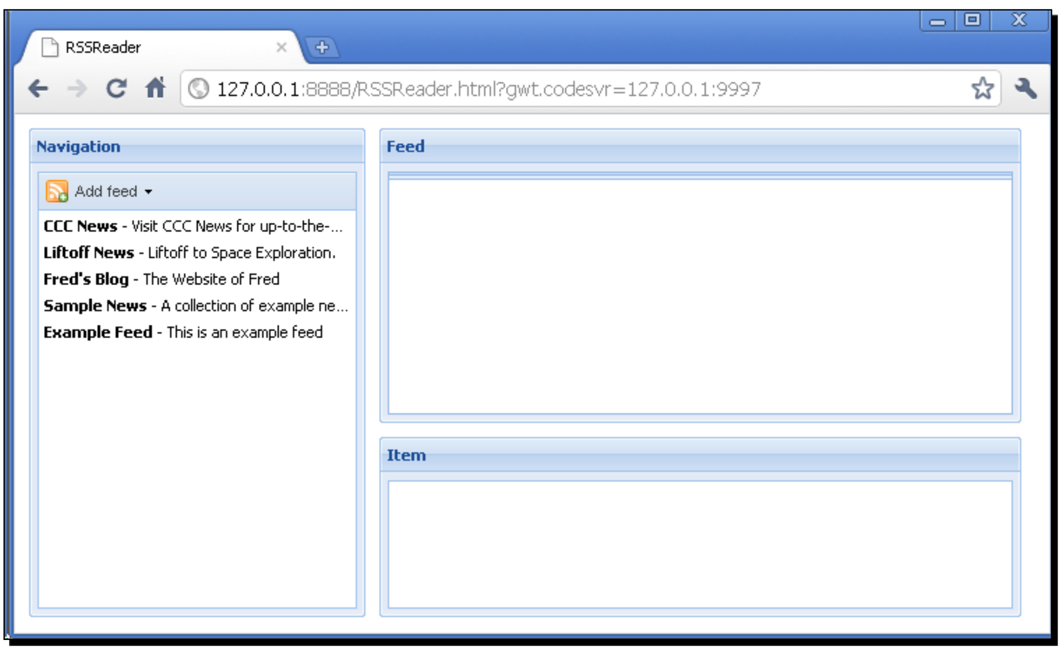
5. Create a new class called `ItemPortlet` again extending `Portlet` and with a similar constructor to the other `Portlet` components, but this time using an `ItemPanel` as the content.

```
  public ItemPortlet()
  {
    setHeading("Item");
    setLayout(new FitLayout());
    setHeight(250);
    setId(RSSReaderConstants.ITEM_PORTLET);
    final ItemPanel itemPanel = new ItemPanel();
    itemPanel.setHeaderVisible(false);
    add(itemPanel);
    Dispatcher.forwardEvent(AppEvents.NewPortletCreated, this);
  }
```

6. With the new portlets defined, we can now create new instances of each in the `onModuleLoad` method of the `RSSReader EntryPoint` class.

```
public void onModuleLoad() {
  Registry.register(RSSReaderConstants.FEED_SERVICE, GWT
           create(FeedService.class));
  Dispatcher dispatcher = Dispatcher.get();
  dispatcher.addController(new PortalController());
  new NavPortlet();
  new FeedPortlet();
  new ItemPortlet();
}
```

7. Now start the application, and you will see that there are three `Portlet` components in the `Portal`.



What just happened?

We now have three `Portlet` components in our `Portal`. However, two are blank, and selecting a feed from the list will not do anything because there is nothing to pass the data to the other portlets. We are going to solve this in a different way by using drag-and-drop.

Drag-and-drop

Drag-and-drop is another built-in feature of GXT that is useful and flexible. Like other GXT features, drag-and-drop is a feature common in desktop applications but unusual in web applications.

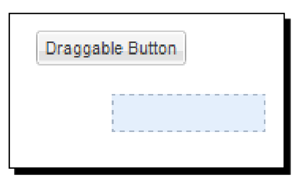
Many GXT components already have specific drag support, but you can extend this to any component you like by implementing the `Draggable` class.

The Draggable class

The `Draggable` class is used to add drag behavior to any component by providing a wrapper around it. For example, if we wanted to make a `Button` draggable, we would do the following:

```
Button dragButton = new Button("Draggable Button");
Draggable draggaable = new Draggable(dragButton);
```

Now the user will be able to drag the `Button`. The new location for the `Button` will be shown by a blue "ghost" rectangle of the `Button` like this:



By default, a draggable component can be dragged in any direction. However, this can be constrained to not allow horizontal or vertical dragging using `setConstrainVertical` and `setConstrainHorizontal` respectively.

```
Button dragButton = new Button("Draggable Button");
Draggable draggable = new Draggable(dragButton);
draggable.setConstrainVertical(true);
```

The DragSource class

The `DragSource` class identifies a component that drag and drops can be initiated from.

A `DragSource` is used to define the data that will be dragged during the drag-and-drop operation. Data can either be moved or copied from the source component. As this setting is only set when the data reaches the target, a `DragSource` also needs to be able to remove data from the source component.

The data can be set using the setData method of the DragSource. When the drag starts, a new DNDEvent is created. Alternatively, data can be set at this point by using the setData method on DNDEvent itself by overriding the onDragStart method.

```
DragSource source = new DragSource(component) {
  @Override
  protected void onDragStart(DNDEvent event) {
    event.setData(component);
  }
};
```

DragSource implementations

All the data-backed controls—Grid, ListView, TreeGrid, and TreePanel—have ready-made DragSource implementations. These support both single and multi-selection.

The implementations for each component are as follows:

Component	DragSource
Grid	GridDragSource
ListView	ListViewDragSource
TreeGrid	TreeGridDragSource
TreePanel	TreePanelDragSource

The DropTarget class

DropTarget is the other end of the drag-and-drop operation. DropTarget identifies a component that can receive the data from a drag-and-drop operation.

A DropTarget is responsible for a number of things. The first is determining if the object that is dragged over it is valid for a drop and showing a visual indication.

Data is obtained from the DropTarget by overriding the onDragDrop method and calling the getData method of the DNDEvent.

```
DropTarget target = new DropTarget(component) {
  @Override
  protected void onDragDrop(DNDEvent event) {
    super.onDragDrop(event);
    Object data = event.getData();
  }
```

It can also specify the DND.Operation, which is either COPY or MOVE. If it is moved (the default), the corresponding DragSource needs to remove the data from its component.

```
target.setOperation(DND.Operation.MOVE);
```

DropTarget implementations

As with `DragSource`, there are specific ready-made implementations for `Grid`, `ListView`, `TreeGrid`, and `TreePanel`.

Component	DropTarget
`Grid`	`GridDropTarget`
`ListView`	`ListViewDropTarget`
`TreeGrid`	`TreeGridDropTarget`
`TreePanel`	`TreePanelDropTarget`

Grouping sources and targets

Both `DragSource` and `DropTarget` classes can be put into groups to constrain where data can be dragged and dropped to. This is useful for avoiding the user dropping data into a component that is unable to handle data of that type.

Simply use the `setGroup` method with a `String` parameter to identify the group of both the `DragSource` and `DropTarget` classes to put them in the same group. Once a `DropTarget` is in a group, it will only accept data from a `DragSource` in the same group.

Pop Quiz – Quick Q&A

Match the description to the correct component, method, or concept.

1. Special panel that extends ContentPanel and can be repositioned in the Viewport
2. Prevents a Portlet being repositioned.
3. The two things we must do when creating a Portal.
4. Class that allows any Component to be dragged.
5. The two places source data can be set in a drag operation.
6. `DragSource` for Grid.
7. `DropTarget` for `TreePanel`.
8. Prevents vertical dragging.

a. Set the number of columns and the column widths
b. `GridDragSource`
c. `Draggable`
d. `setPinned` (true)

e. setData of the DragSource and setData of the DNDEvent
f. setConstrainVertical(true)
g. Portlet
h. TreePanelDropTarget

Using drag-and-drop

We can use drag-and-drop with the Portal layout of our example application. This will give an example of how to use built-in and custom DragSource and DropTarget components.

The first thing we are going to do is allow users to drag a feed from the FeedList in the NavPortlet. When dropped on the FeedPortlet, this will cause the items in the feed to be displayed in an ItemGrid.

Time for action – dragging and dropping of feeds

1. In the RSSReaderConstants class, create a new constant named FEED_DD_GROUP to act as an ID for the drag-and-drop group for feeds.

```
public static final String FEED_DD_GROUP = "feedDDGroup";
```

2. At the end of the onRender method of the FeedList class, create a new DragSource object that wraps the FeedList.

```
DragSource source = new DragSource(feedList);
```

3. Override the onDragStart method of the DragSource so that the BeanModel object is selected from the FeedList and is attached to the DNDEvent as data.

```
DragSource source = new DragSource(feedList) {
  @Override
  protected void onDragStart(DNDEvent event) {
    event.setData(feedList.getSelection());
  }
};
```

4. Set the group of the DragSource to FEED_DD_GROUP.

```
DragSource source = new DragSource(feedList) {
  @Override
  protected void onDragStart(DNDEvent event) {
    event.setData(feedList.getSelection());
  }
};
source.setGroup(RSSReaderConstants.FEED_DD_GROUP);
```

5. Now in `FeedPortlet`, create a new method named `onFeedsDropped`. This should extract the `BeanModel` objects contained in the data of the `DNDEvent`, and with each of them, create a new ItemGrid for each `Feed` in the same way as we did in the `FeedView`.

```
private void onFeedsDropped(DNDEvent event) {
  List<BeanModel> beanModels = event.getData();
  for (BeanModel beanModel : beanModels) {
    Feed feed = beanModel.getBean();
    final ItemGrid itemGrid = new ItemGrid(feed);
    TabItem tabItem = new TabItem(feed.getTitle());
    tabItem.setId(feed.getUuid());
    tabItem.setData("feed", feed);
    tabItem.add(itemGrid);
    tabItem.addListener(Events.Select, new
Listener<TabPanelEvent>() {
      @Override
      public void handleEvent(TabPanelEvent be) {
        itemGrid.resetSelection();
      }
    });
    tabItem.setClosable(true);
    feedPanel.addTab(tabItem);
  }
}
```

6. Override the `onRender` method of `FeedPortlet`, and in it, create a new `DropTarget` using the actual `FeedPortlet` itself as the target component.

```
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  DropTarget target = new DropTarget(this);
}
```

7. Override the `onDragDrop` method of the `DropTarget` so that it passes the `DNDEvent` to the `onFeedsDropped` method.

```
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  DropTarget target = new DropTarget(this) {
    @Override
    protected void onDragDrop(DNDEvent event) {
      super.onDragDrop(event);
      onFeedsDropped(event);
    }
  };
}
```

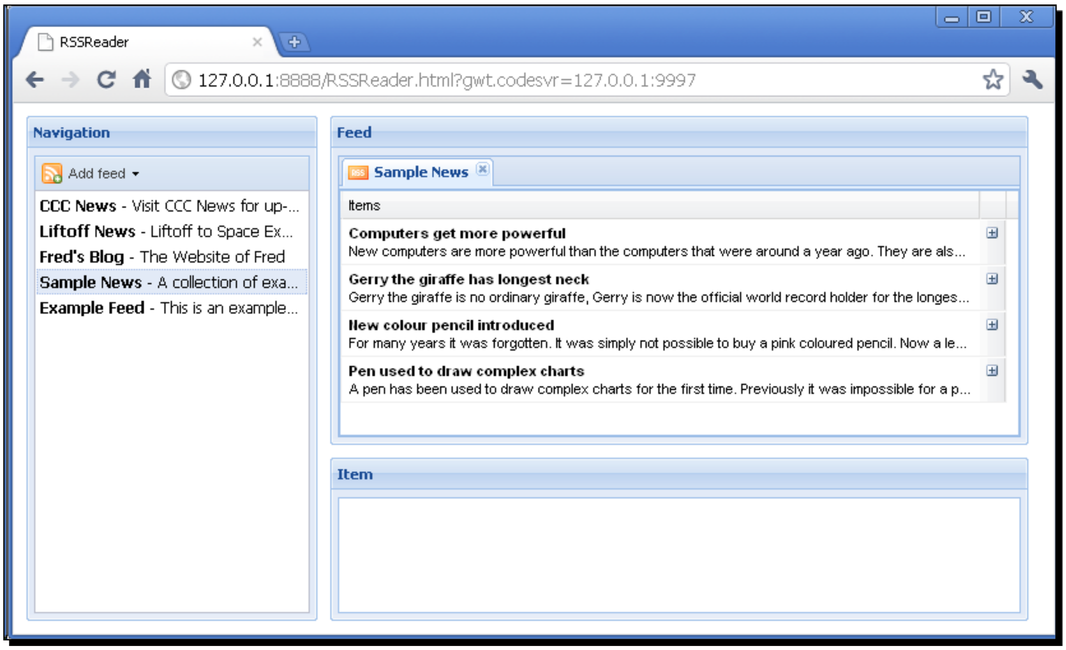
8. Set the operation of the `DropTarget` to be `DND.Operation.COPY` so that the selected feeds are not removed from the `FeedList` when the data is dropped.

```
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  DropTarget target = new DropTarget(this) {
    @Override
    protected void onDragDrop(DNDEvent event) {
      super.onDragDrop(event);
      onFeedsDropped(event);
    }
  };
  target.setOperation(DND.Operation.COPY);
}
```

9. Set the group of the `DropTarget` to `FEED_DD_GROUP`, so that it is in the same group as the `DragSource` we defined earlier.

```
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  DropTarget target = new DropTarget(this) {
    @Override
    protected void onDragDrop(DNDEvent event) {
      super.onDragDrop(event);
      onFeedsDropped(event);
    }
  };
  target.setOperation(DND.Operation.COPY);
  target.setGroup(RSSReaderConstants.FEED_DD_GROUP);
}
```

10. Start the application and drag a feed from the `FeedList` to the `FeedPortlet` to display the content of the feed in an `ItemGrid`.



What just happened?

We used a `DragSource` together with a custom `DropTarget` to allow the drag-and-drop of the feeds to view as a list of items.

We can now implement drag-and-drop in a similar way to allow items to be dragged from the `FeedPortlet` `ItemGrid` to the `ItemPortlet` to display them.

Time for action – dragging and dropping items

1. In the `RSSReaderConstants` class, create a new constant named `ITEM_DD_GROUP` to act as an ID for the drag-and-drop group for items.

```
public static final String ITEM_DD_GROUP = "itemDDGroup";
```

2. At the end of the `onRender` method of the `ItemGrid`, but before the `Grid` is added, create a new `GridDragSource` using the `Grid` as the source component.

```
GridDragSource source = new GridDragSource(grid);
```

3. Set the group of the `GridDragSource` to `ITEM_DD_GROUP`.

```
GridDragSource source = new GridDragSource(grid);
source.setGroup(RSSReaderConstants.ITEM_DD_GROUP);
```

4. In the constructor of the `ItemPortlet` class, create a `DropTarget` where the `ItemPortlet` is the target.

```
DropTarget target = new DropTarget(this);
```

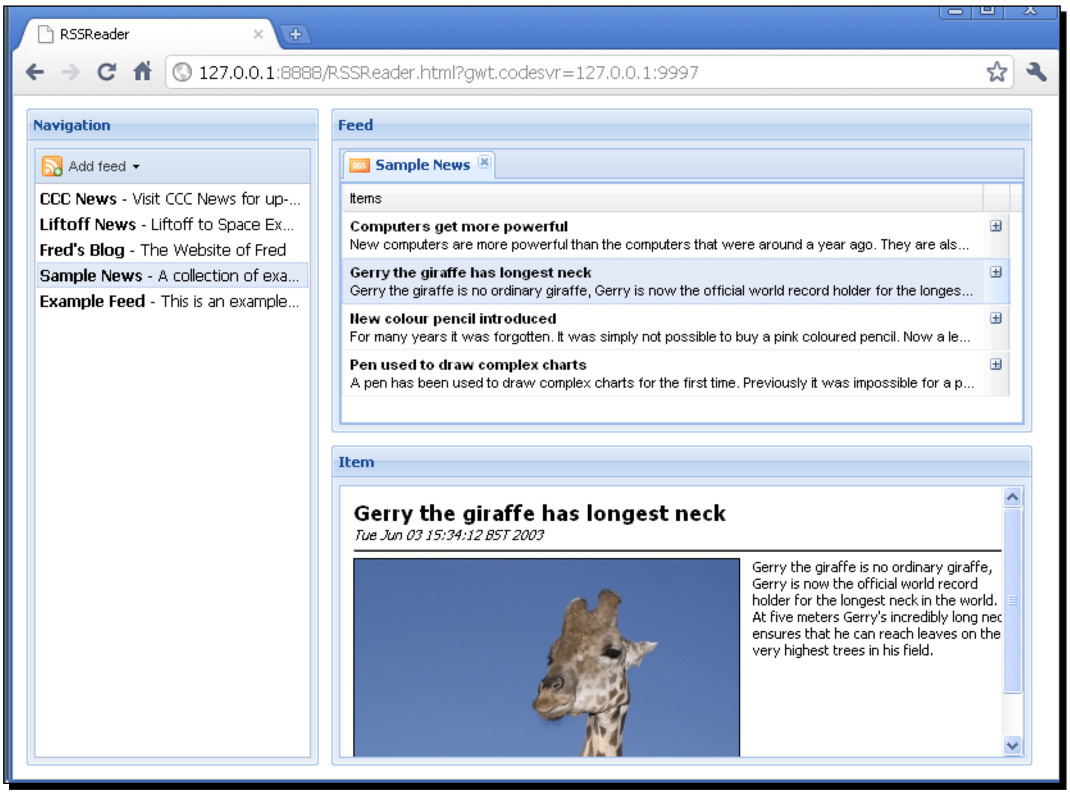
5. Override the `onDragDrop` method of the `DropTarget` to retrieve the list of `Item` objects and then call the `displayItem` method of the `ItemPanel` to display the first `Item` object in the list.

```
DropTarget target = new DropTarget(this) {
  @Override
  protected void onDragDrop(DNDEvent event) {
    super.onDragDrop(event);
    List<Item> items = event.getData();
    itemPanel.displayItem(items.get(0));
  }
};
```

6. Again we need to set the operation of the target to `COPY` to avoid removing the `Item` objects from the `Grid` and use `setGroup` to put the `DropTarget` in the same group as its `DragSource`. This will prevent the user from being able to drop a feed into the `ItemPortlet`.

```
DropTarget target = new DropTarget(this) {
  @Override
  protected void onDragDrop(DNDEvent event) {
    super.onDragDrop(event);
    List<Item> items = event.getData();
    itemPanel.displayItem(items.get(0));
  }
};
target.setOperation(DND.Operation.COPY);
target.setGroup(RSSReaderConstants.ITEM_DD_GROUP);
```

7. Now start the application, drop a `Feed` from the `NavPortlet` to the `FeedPortlet`, and then an `Item` from the `FeedPortlet` to the `ItemPortlet`.



What just happened?

We implemented drag-and-drop between the `FeedPortlet` and the `ItemPortlet`. We now have three portlets that are completely independent and just respond to the data that is dragged into them.

Have a go hero – creating an overview portlet

In the previous chapter, we created a `FeedOverviewView` that displayed a summary icon for a feed. Create a new `OverviewPortlet` that contains the `FeedOverviewView`.

Modify the view so instead of loading `Feed` objects from the `FeedService`, it adds and renders them in the `ListView` when `Feed` objects are dragged into the `OverviewPortlet`.

Solution:

OverviewPortlet:

```
public class OverviewPortlet extends Portlet {

  public OverviewPortlet() {
    setHeading("Overview");
    setLayout(new FitLayout());
    setHeight(250);
    setId(RSSReaderConstants.OVERVIEW_PORTLET);
    final FeedOverviewView feedOverviewView = new FeedOverviewView();
    add(feedOverviewView);

    DropTarget target = new DropTarget(this) {
      @Override
      protected void onDragDrop(DNDEvent event) {
        super.onDragDrop(event);
        List<BeanModel> beanModels = event.getData();
        feedOverviewView.addFeeds(beanModels);
      }
    };
    target.setOperation(DND.Operation.COPY);
    target.setGroup(RSSReaderConstants.FEED_DD_GROUP);

    Dispatcher.forwardEvent(AppEvents.NewPortletCreated, this);
  }
}
```

Modified FeedOverviewView:

```
public class FeedOverviewView extends LayoutContainer {

  private final ListStore<BeanModel> feedStore = new
ListStore<BeanModel>();
  private ListView<BeanModel> listView = new ListView<BeanModel>();

  public void addFeeds(List<BeanModel> feeds)
  {
    feedStore.add(feeds);
  }

  private String getTemplate() {
    StringBuilder sb = new StringBuilder();
    sb.append("<tpl for=\".\">");
```

```
    sb.append("<div class=\"feed-box\">");
    sb.append("<h1>{title}</h1>");
    sb.append("<tpl if=\"imageUrl!=''\">");
    sb.append("<img class=\"feed-thumbnail\" src=\"{imageUrl}\"
       title=\"{shortTitle}\">");
    sb.append("</tpl>");
    sb.append("<p>{shortDescription}</p>");
    sb.append("<ul>");
    sb.append("<tpl for=\"items\">");
    sb.append("<tpl if=\"xindex &lt; 3\">");
    sb.append("<li>{title}</li>");
    sb.append("</tpl>");
    sb.append("</tpl>");
    sb.append("</ul>");
    sb.append("</div>");
    sb.append("</tpl>");
    return sb.toString();
  }

  @Override
  protected void onRender(Element parent, int index) {
    super.onRender(parent, index);
    setScrollMode(Scroll.AUTOY);
    listView = new ListView<BeanModel>() {
      @Override
      protected BeanModel prepareData(BeanModel feed) {
        feed.set("shortTitle", Format.ellipse((String) feed
            .get("title"), 50));
        feed.set("shortDescription", Format.ellipse((String) feed
            .get("description"), 100));
        return feed;
      }
    };
    listView.setStore(feedStore);
    listView.setTemplate(getTemplate());
    listView.setItemSelector("div.feed-box");
    listView.getSelectionModel().addListener(Events.SelectionChange,
new Listener<SelectionChangedEvent<BeanModel>>() {
      public void handleEvent(SelectionChangedEvent<BeanModel> be) {
        BeanModel feed = (BeanModel) be.getSelection().get(0);
        Info.display("Feed selected", (String)feed.get("title"));
      }
    });
```

```
      add(listView);
    }
  }
```

Modified RSSReader:

```
  public class RSSReader implements EntryPoint {

    public void onModuleLoad() {
      Registry.register(RSSReaderConstants.FEED_SERVICE, GWT
          .create(FeedService.class));
      Dispatcher dispatcher = Dispatcher.get();
      dispatcher.addController(new PortalController());
      new NavPortlet();
      new OverviewPortlet();
      new FeedPortlet();
      new ItemPortlet();
    }
  }
```

Summary

We have looked at GXT's drag-and-drop features and used `Portlet` components to create components that independently respond to the data that is dragged and dropped into them.

In the next chapter, we will look at GXT's charting capabilities.

9

Charts

In this chapter, we will look at the GXT charting plugin. We will explore the wide range of charts available, avoid the pitfalls of the plugin, and see how we can use charts with existing data.

Specifically, we will cover the following classes:

- `Chart`
- `ChartModel`
- `ChartConfig`
- `BarChart`
- `CylinderBarChart`
- `FilledBarChart`
- `SketchBarChart`
- `HorizontalBarChart`
- `PieChart`
- `LineChart`
- `AreaChart`

Charts are a bit different from the other parts of GXT. Rather than being a core part of the framework, they are an add-in based on Open Flash Charts 2, an LGPL Flash-based charting system. More information on Open Flash Charts is available here: `http://teethgrinder.co.uk/open-flash-chart-2/`.

As charts are a plug-in to GXT, several configuration steps are required to get them to work and these are not obvious. This means that it is easy to run into problems. Therefore, before we get started properly with charts, let's set up our example application step-by-step so that it can use them. We will also look at the error message that will appear if we miss out a step, as this should help if you encounter problems in your own applications.

Time for action – including the chart module

1. In the project's module file, RSSReader.gwt.xml, add a line to include the charts module so that it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='rssreader'>
…
  <!-- Other module inherits -->
  <inherits name='com.extjs.gxt.ui.GXT' />
  <inherits name='com.extjs.gxt.charts.Chart' />
…
</module>
```

2. If you miss this step, you will see an error message like this:

```
17:47:28.468 [ERROR] [rssreader] Line 26: No source code is
available for type com.extjs.gxt.charts.client.Chart; did you
forget to inherit a required module?
```

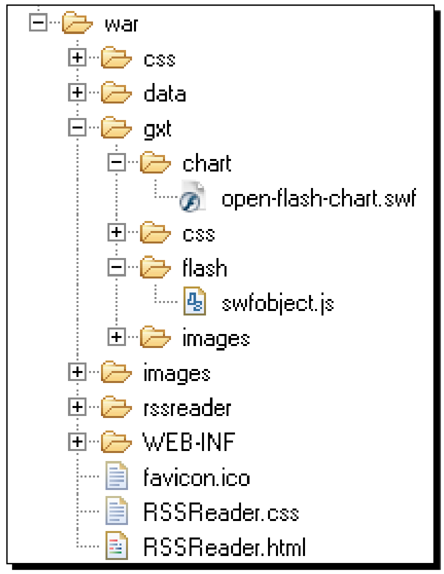
What just happened?

We added the charts module to the example application. This includes the chart source code and makes charts available to the application.

The charts themselves are displayed using Flash and JavaScript. The code is contained in resource files and not the chart module itself. These resource files need to be included in the project.

Time for action – including the chart resources

1. In the resources folder of the GXT distribution, locate the chart folder and copy it to the project's war\gxt folder.
2. Also locate the flash folder in the resources folder and again copy it to the project's war\gxt folder.
3. The war folder should now look like this:



4. If you forget to include the `chart` folder, you will get an error like this on the console in Eclipse:

```
[WARN] 404 - GET /gxt/chart/open-flash-chart.swf (127.0.0.1) 1416
bytes
```

5. If you forget to include the `flash` folder, you will get an error like this:

```
18:27:08.015 [ERROR] [rssreader] Unable to load module entry point
class
  com.danielvaughan.rssreader.client.RSSReader (see associated
exception for
  details)
com.google.gwt.core.client.JavaScriptException: (TypeError):
Cannot call
  method 'embedSWF' of undefined
 stack: TypeError: Cannot call method 'embedSWF' of undefined
```

as well as the following message on the Java console:

```
[WARN] 404 - GET /gxt/flash/swfobject.js (127.0.0.1) 1408 bytes
```

What just happened?

We added the chart resources, specifically a flash file and a JavaScript library to the example project.

Finally, we need to load the JavaScript library.

Time for action – loading the chart JavaScript library

1. In the header of the `RSSReader.html` file, add a script tag that loads the JavaScript library for GXT's charts:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-
  8">
<link type="text/css" rel="stylesheet" href="RSSReader.css">
<link type="text/css" rel="stylesheet" href="css/item.css">
<link type="text/css" rel="stylesheet" href="gxt/css/gxt-all.css">
<script language='javascript' src='gxt/flash/swfobject.js'>
</script>
<title>RSSReader</title>
</head>
```

2. If you forget to do this, you will again get the following error. However, you will not get the 404 error on the console:

```
18:27:08.015 [ERROR] [rssreader] Unable to load module entry point
class
  com.danielvaughan.rssreader.client.RSSReader (see associated
exception for
  details)
com.google.gwt.core.client.JavaScriptException: (TypeError):
Cannot call
  method 'embedSWF' of undefined
 stack: TypeError: Cannot call method 'embedSWF' of undefined:
```

What just happened?

We added a script tag in the example application's HTML file to load the chart JavaScript library.

Now that we have set up the example application to use charts, let's create a simple example.

Chart class

`Chart` is the Java class that wraps the Open Flash Chart library and allows it to be treated as a GXT component. It needs to be provided with an URL that corresponds with the location of the `open-flash-chart.swf` that is contained in the `chart` folder that we copied from GXT's resources. This must be correct, otherwise the chart will simply not render. No error will be displayed, apart from a 404 error on the console.

For example, let's say that instead of:

```
Chart chart = new Chart("gxt/chart/open-flash-chart.swf");
```

We wrote:

```
Chart chart = new Chart("wrong/path/open-flash-chart.swf");
```

We would get the message on the console:

```
[WARN] 404 - GET /wrong/path/open-flash-chart.swf (127.0.0.1) 1417
bytes
```

As an example of using a chart, let's create a new `Portlet` for the example application that contains a chart.

Time for action – creating a chart Portlet

1. Create a container for the chart named `FeedChart` that extends `LayoutContainer` in a package named `client.charts`:

```
public class FeedChart extends LayoutContainer {
```

2. Create a new chart property using the URL for the `open-flash-chart.swf` file:

```
private final Chart chart = new Chart("gxt/chart/open-flash-
  chart.swf");
```

3. Override the `onRender` method to use `FitLayout` for the container, add borders to the chart, and add the chart to the container:

```
@Override
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  setLayout(new FitLayout());
  chart.setBorders(true);
  add(chart);
}
```

4. Create a new class that extends `Portlet` in the `client.portlet` package and name it `ChartPortlet`:

```
public class ChartPortlet extends Portlet {
```

5. Create a new instance of `FeedChart`:

```
private final FeedChart feedChart = new FeedChart();
```

6. In the constructor of the `Portlet`, define the title, layout, and height of the `Portlet` and set the ID to a new `RSSReaderConstants` named `CHART_PORTLET`:

```
public ChartPortlet()
{
  setHeading("Chart");
  setId(RSSReaderConstants.CHART_PORTLET);
  setLayout(new FitLayout());
  setHeight(250);
}
```

7. Add the `FeedChart` to the `Portlet` and dispatch an event to notify the application that the new `Portlet` has been created:

```
public ChartPortlet()
{
  setHeading("Chart");
  setId(RSSReaderConstants.CHART_PORTLET);
  setLayout(new FitLayout());
  setHeight(250);
  add(feedChart);
  Dispatcher.forwardEvent(AppEvents.NewPortletCreated, this);
}
```

8. We would like the `ChartPortlet` to be shown in the first column of the `Portal`. We also need to modify the `onAddPortlet` method of the `PortalView` class to do this:

```
private void onAddPortlet(AppEvent event) {
  final Portlet portlet = (Portlet) event.getData();
  if (portlet.getId() == RSSReaderConstants.NAV_PORTLET
    || portlet.getId() == RSSReaderConstants.CHART_PORTLET) {
    portal.add(portlet, 0);
  } else {
    portal.add(portlet, 1);
  }
}
```

9. We also need to make the height of the `NavPortlet` smaller to make room for the `ChartPortlet`, so change the `setHeight` line in the constructor of the `NavPortlet`:

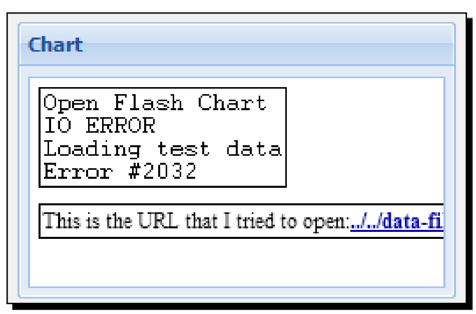
```
public NavPortlet() {
  setHeading("Navigation");
  setLayout(new FitLayout());
  setHeight(350);
  setId(RSSReaderConstants.NAV_PORTLET);
```

```
    NavPanel navPanel = new NavPanel();
    navPanel.setHeaderVisible(false);
    add(navPanel);
    Dispatcher.forwardEvent(AppEvents.NewPortletCreated, this);
  }
```

10. In the `onModuleLoad` method of the `RSSReader EntryPoint` class, create a new instance of the `ChartPortlet` class:

```
public void onModuleLoad() {
  Registry.register(RSSReaderConstants.FEED_SERVICE, GWT
    .create(FeedService.class));
  Dispatcher dispatcher = Dispatcher.get();
  dispatcher.addController(new PortalController());
  new NavPortlet();
  new FeedPortlet();
  new ItemPortlet();
  new ChartPortlet();
}
```

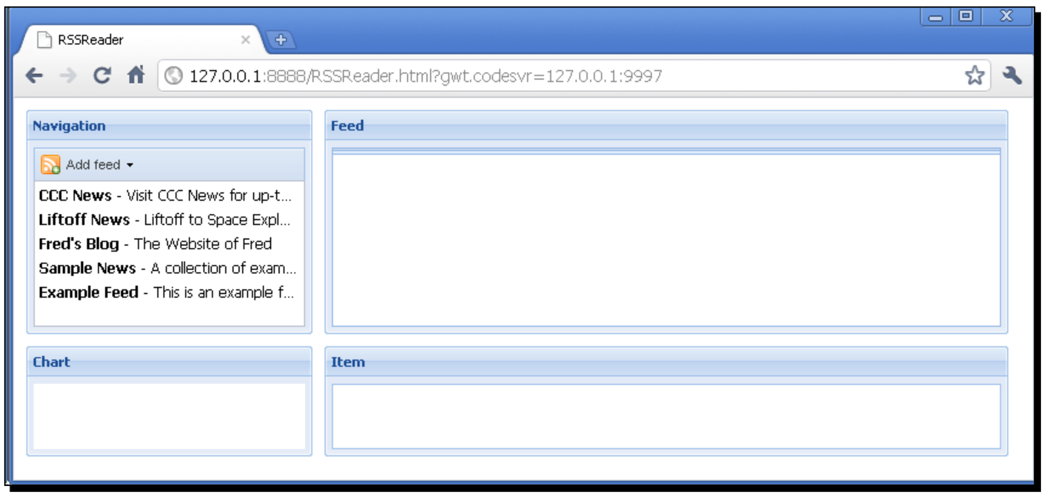
11. Start the application now and you will see the `ChartPortlet` displayed. But it will be displayed with an error from Open Flash Charts like this:



12. The reason for this is that we have not loaded any data for the chart and at this stage we don't want to. To avoid displaying the error, we can make the chart invisible before adding it to the `Portlet` in the `onRender` method of `FeedChart`:

```
@Override
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  setLayout(new FitLayout());
  chart.setBorders(true);
  chart.setVisible(false);
  add(chart);
}
```

13. Start the application again and you will see the empty chart `Portlet` without the error:



What just happened?

We created a chart component and displayed it using a `Portlet`. We then made the chart invisible to avoid Open Flash Charts displaying an error message, as the chart does not have any data.

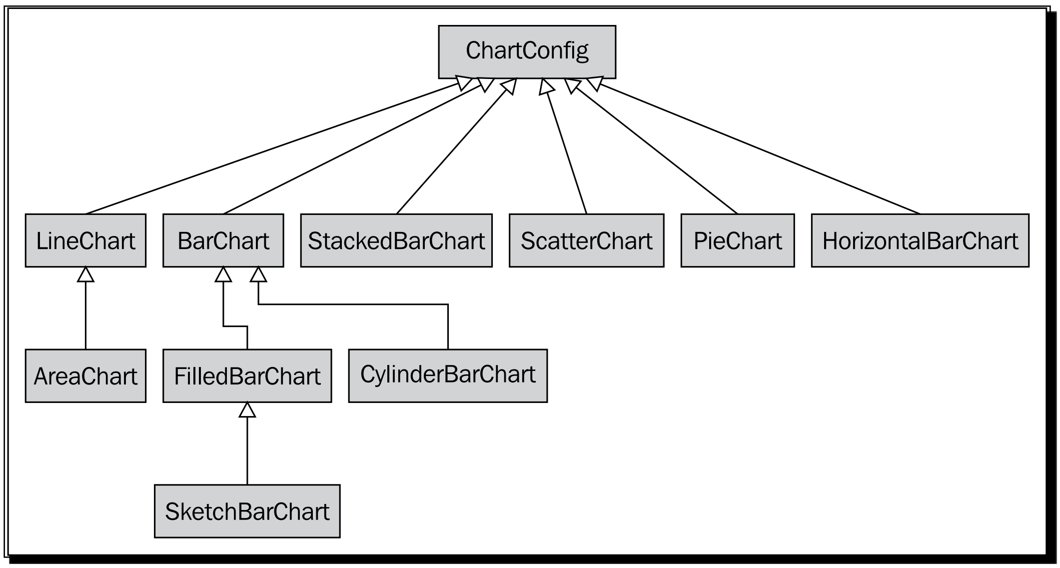
ChartModel class

`ChartModel` extends `BaseModel` to provide a data model compatible with Open Flash Chart's chart model. The `ChartModel` is used to define the type, appearance, and data of a chart. This is where the work is done. A `ChartModel` contains one or more `ChartConfig` objects for different types of charts that can be displayed.

ChartConfig class

`ChartConfig` is an abstract class that again extends `BaseModel`. This class provides the base class for a number of classes that define specific chart types. The classes that extend `ChartConfig` provide a hierarchy of different chart styles.

The following diagram shows the relationships between the different chart types that extend `ChartConfig`:



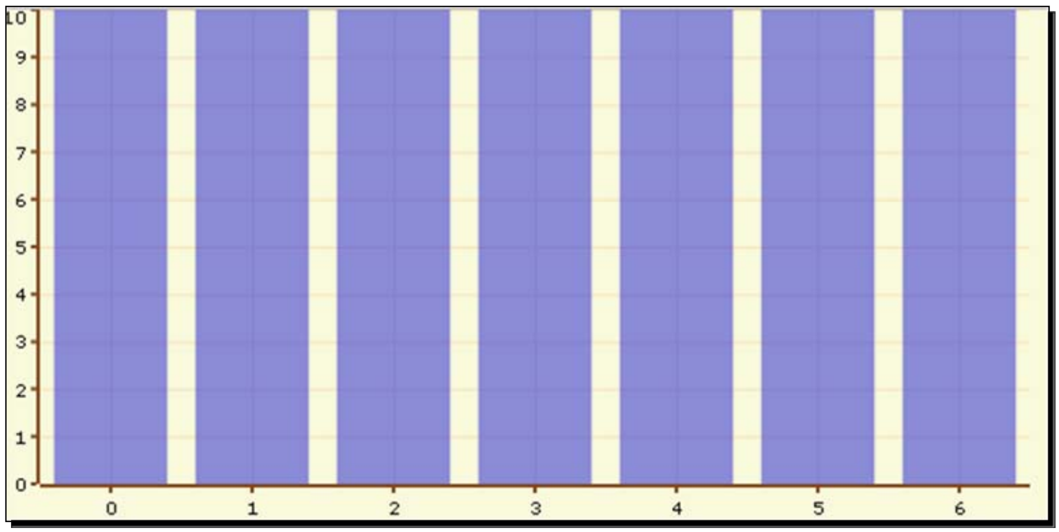
BarChart class

The first `ChartConfig` that we are going to look at is the `BarChart`. In its very simplest form, we can create a `createChartModelData` method as follows:

- Create a `ChartModel`
- Create a `BarChart ChartConfig`
- Add values to the `BarChart`
- Add the `BarChart` to the `ChartModel`
- Return the `ChartModel`

```
private ChartModel createChartModel() {
  ChartModel chartModel = new ChartModel();
  BarChart chartConfig = new BarChart();
  chartConfig.addValues(6936,8628,41832,68376,296,10114,4693);
  chartModel.addChartConfig(chartConfig);
  return chartModel;
}
```

However, this will produce a chart that looks like this:



The main problem is that the values are too large for the Y axis. We can tidy this up as follows:

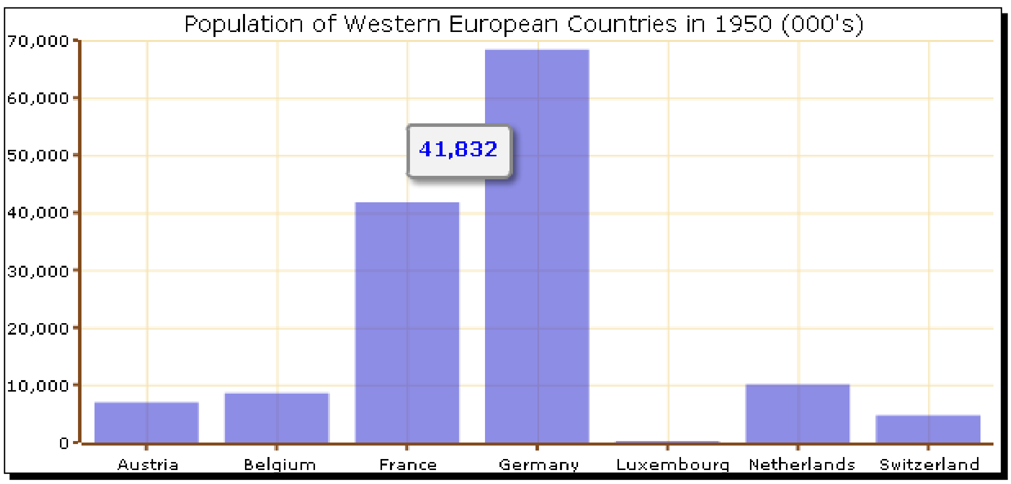
```
private ChartModel getChartModel() {
  ChartModel chartModel = new ChartModel("Population of Western
    European Countries in 1950 (000's)","font-
    size:14px;color:#000000");
  chartModel.setBackgroundColour("#ffffff");
  XAxis xAxis = new XAxis();
  xAxis.addLabels("Austria", "Belgium", "France", "Germany",
    "Luxembourg", "Netherlands", "Switzerland");
  chartModel.setXAxis(xAxis);
  YAxis yAxis = new YAxis();
  yAxis.setRange(0, 70000, 10000);
  chartModel.setYAxis(yAxis);
  BarChart chartConfig = new BarChart();
  chartConfig.addValues(6936,8628,41832,68376,296,10114,4693);
  chartModel.addChartConfig(chartConfig);
  return chartModel;
}
```

Here we are:

- Specifying a title and CSS styling information in the constructor of `ChartModel`.
- Setting the background color

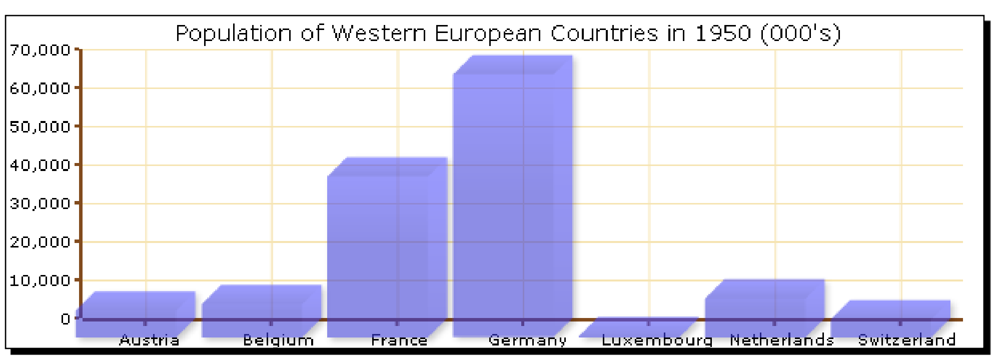
- Defining an `XAxis` and adding labels
- Defining a `YAxis` and setting the range to be large enough to accommodate all our data

These changes produce a much more satisfactory chart. Placing the mouse on a bar causes a pop up showing the bar's value to be displayed:

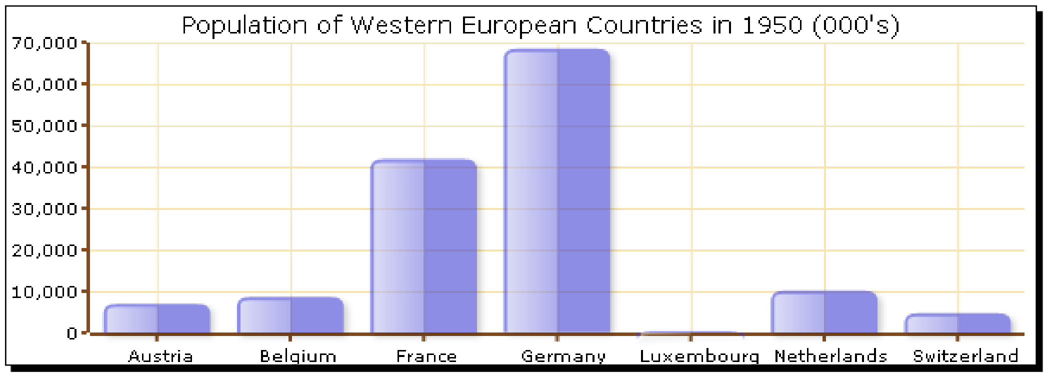


When we create a `BarChart` with no parameters in the constructor, we get a default `BarChart`. However, we can pass a `BarStyle` parameter to display the columns with one of two effects:

```
BarChart chartConfig = new BarChart(BarStyle.THREED);
```



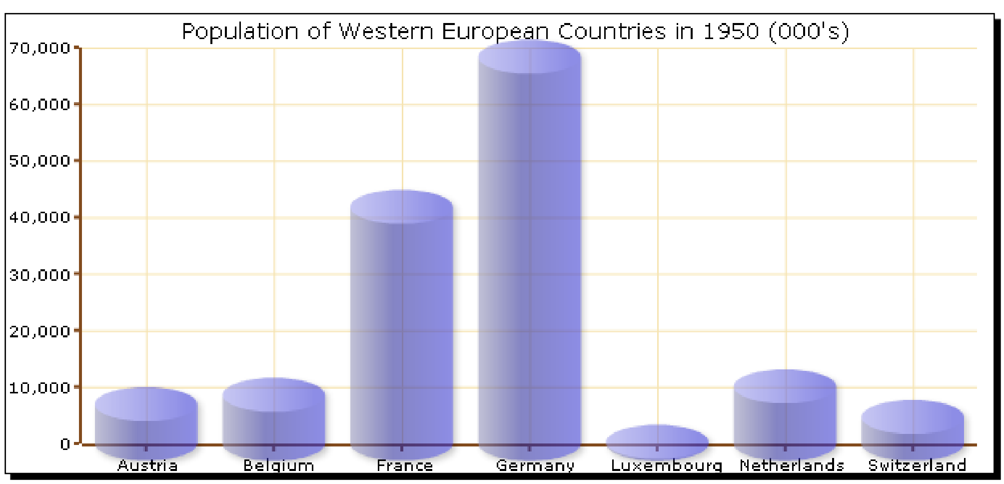
```
BarChart chartConfig = new BarChart(BarStyle.GLASS);
```



However, this is just the start. There are a number of components that extend `BarChart` to give a different look, and using them is simply a question of changing the chart that is created.

CylinderBarChart class

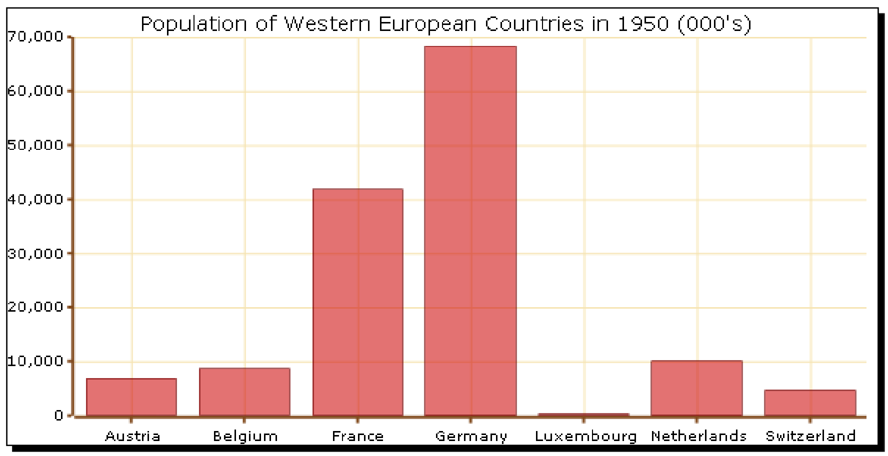
There is the `CylinderBarChart`:




```
BarChart chartConfig = new CylinderBarChart();
chartConfig.addValues(6936,8628,41832,68376,296,10114,4693);
```

FilledBarChart class

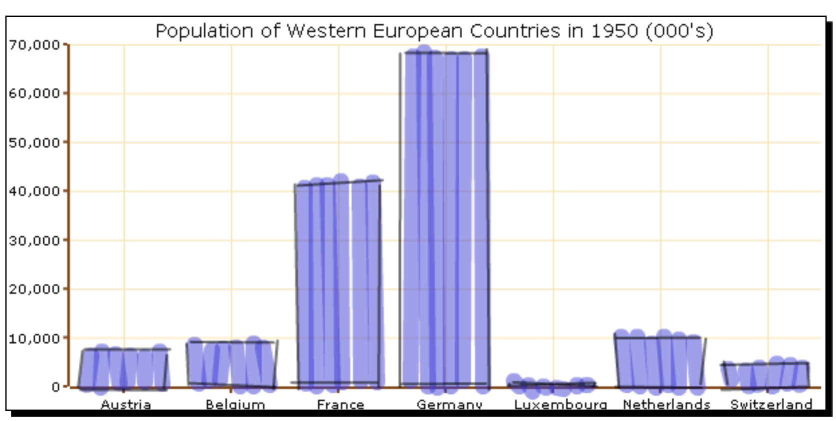
There is the `FilledBarChart` that will look exactly the same as a standard `BarChart` unless we use the `setOutlineColor` method to set a color for an outline around each bar:




```
FilledBarChart chartConfig = new FilledBarChart();
chartConfig.setColour("#cc0000");
chartConfig.setOutlineColour("#660000");
chartConfig.addValues(6936,8628,41832,68376,296,10114,4693);
```

SketchBarChart class

There is also the more casual `SketchBarChart`:



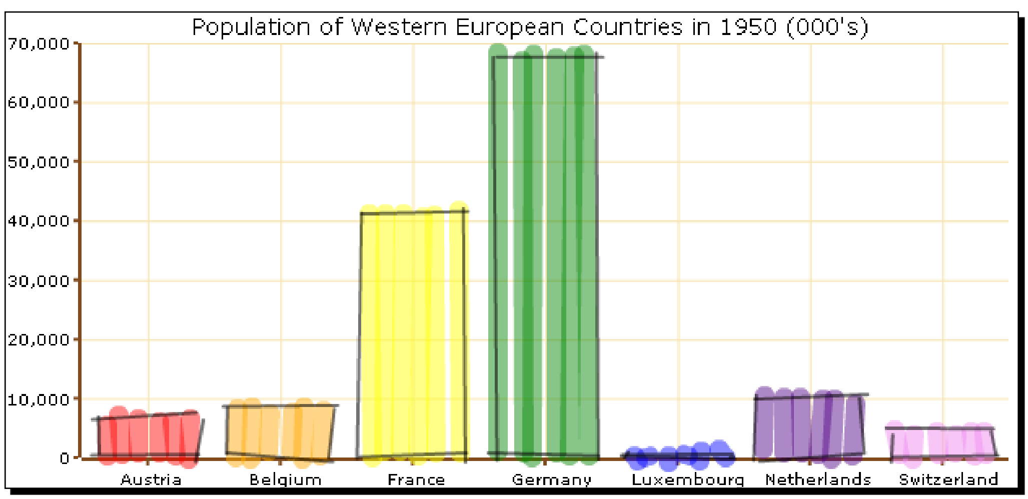

```
BarChart chartConfig = new SketchBarChart();
chartConfig.addValues(6936,8628,41832,68376,296,10114,4693);
```

BarChart.Bar class

Instead of simply adding values to a `BarChart` to produce bars, each `BarChart` class has a `Bar` class that allows us to define the appearance of an individual bar in more detail. For example, we can define different colors for each bar:

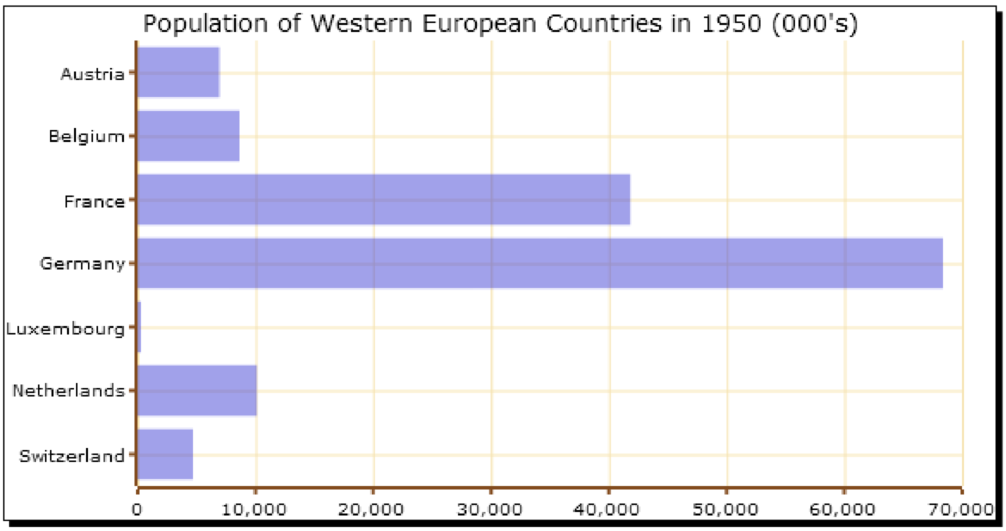
```
BarChart chartConfig = new SketchBarChart();
chartConfig.addBars(new BarChart.Bar(6936, "#FF0000"));
chartConfig.addBars(new BarChart.Bar(8628, "#FFA500"));
chartConfig.addBars(new BarChart.Bar(41832, "#FFFF00"));
chartConfig.addBars(new BarChart.Bar(68376,"#008000"));
chartConfig.addBars(new BarChart.Bar(296,  "#0000FF"));
chartConfig.addBars(new BarChart.Bar(10114,"#4B0082"));
chartConfig.addBars(new BarChart.Bar(4693, "#EE82EE"));
```

This leads to a more colorful chart like this:



HorizontalBarChart class

`HorizontalBarChart` works in the same way as a standard `BarChart`. Of course, the `YAxis` becomes the `XAxis`. It is important to note that the order of the country labels needs to be reversed:



This chart is implemented as follows:

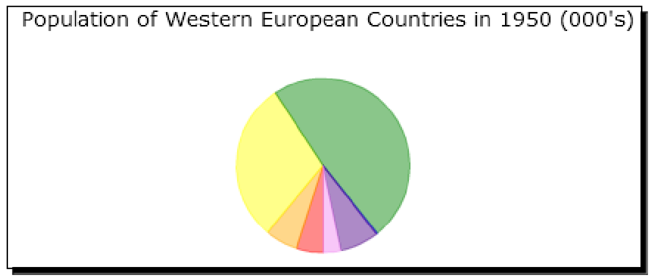
```
YAxis yAxis = new YAxis();
yAxis.addLabels("Switzerland",
  "Netherlands","Luxembourg","Germany","France","Belgium","Austria");
yAxis.setOffset(true);
chartModel.setYAxis(yAxis);
XAxis xAxis = new XAxis();
xAxis.setRange(0, 70000, 10000);
chartModel.setXAxis(xAxis);
HorizontalBarChart chartConfig = new HorizontalBarChart();
chartConfig.addValues(6936,8628,41832,68376,296,10114,4693);
```

PieChart class

Moving away from `BarChart` variants to other charts is also straightforward. In fact, the only change we need to make to move from a `BarChart` to a `PieChart` is to change the definition of `ChartConfig`. It is also a good idea to define a set of colors for the `PieChart` segments using `setColours`:

```
private ChartModel getChartModel() {
  ChartModel chartModel = new ChartModel();
  PieChart chartConfig = new PieChart();
  chartConfig.setColours("#FF0000", "#FFA500", "#FFFF00", "#008000",
    "#0000FF","#4B0082", "#EE82EE");
```

```
        chartConfig.addValues(6936,8628,41832,68376,296,10114,4693);
        chartModel.addChartConfig(chartConfig);
        return chartModel;
    }
```

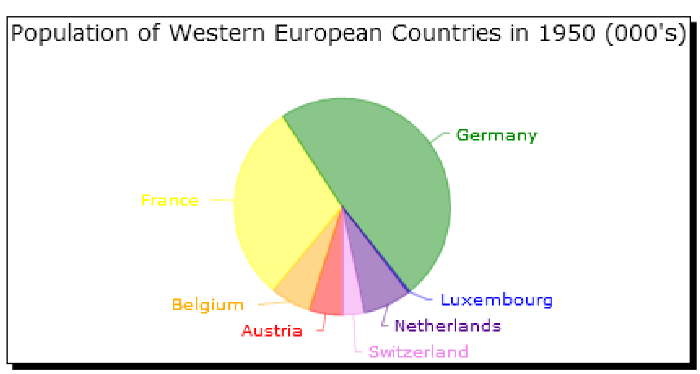


However, this is not very useful as there are no labels on the pie slices.

PieChart.Slice class

As with `BarChart.Bar PieChart`, slices can be individually defined using `PieChart.Slice`. In this case, we can use a `PieChart.Slice` to define both a value and a label:

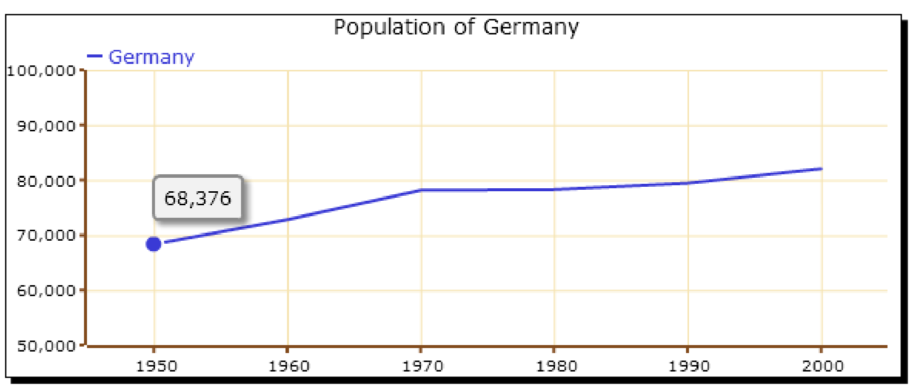
```
PieChart chartConfig = new PieChart();
chartConfig.setColours("#FF0000", "#FFA500", "#FFFF00", "#008000",
  "#0000FF","#4B0082", "#EE82EE");
chartConfig.addSlices(new PieChart.Slice(6936,"Austria"));
chartConfig.addSlices(new PieChart.Slice(8628,"Belgium"));
chartConfig.addSlices(new PieChart.Slice(41832,"France"));
chartConfig.addSlices(new PieChart.Slice(68376,"Germany"));
chartConfig.addSlices(new PieChart.Slice(296,"Luxembourg"));
chartConfig.addSlices(new PieChart.Slice(10114,"Netherlands"));
chartConfig.addSlices(new PieChart.Slice(4693,"Switzerland"));
```



LineChart class

The `LineChart` can be used in a similar manner:

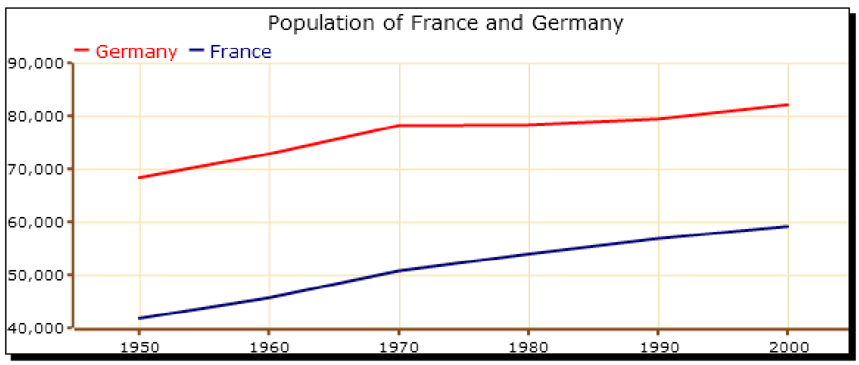
```
private ChartModel getChartModel() {
  ChartModel chartModel = new ChartModel("Population of
    Germany","font-size:14px;color:#000000");
  chartModel.setBackgroundColour("#ffffff");

  XAxis xAxis = new XAxis();
  xAxis.addLabels("1950","1960","1970","1980","1990","2000");
  chartModel.setXAxis(xAxis);

  YAxis yAxis = new YAxis();
  yAxis.setRange(50000, 100000, 10000);
  yAxis.setOffset(true);
  chartModel.setYAxis(yAxis);

  LineChart chartConfig = new LineChart();
  chartConfig.addValues(68376,72815,78169,78289,79433,82075);
  chartConfig.setText("Germany");
  chartModel.addChartConfig(chartConfig);
  return chartModel;
}
```



To add a separate set of data, simply create a separate `ChartConfig` and add it to the model:

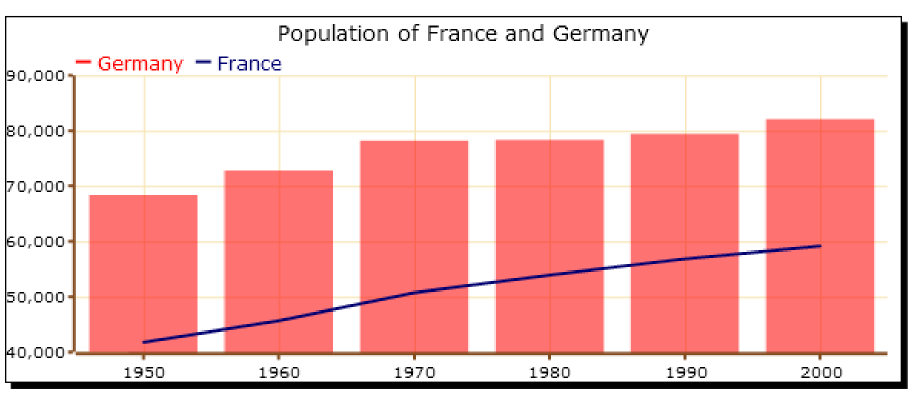
```
LineChart germanyChartConfig = new LineChart();
germanyChartConfig.addValues(68376,72815,78169,78289,79433,82075);
germanyChartConfig.setColour("#ff0000");
germanyChartConfig.setText("Germany");
chartModel.addChartConfig(germanyChartConfig);
```

```
LineChart franceChartConfig = new LineChart();
franceChartConfig.addValues(41832,45674,50771,53950,56842,59128);
franceChartConfig.setColour("#000066");
franceChartConfig.setText("France");
chartModel.addChartConfig(franceChartConfig);
```



When using multiple datasets, the charts do not have to be of the same type. A `LineChart` and a `BarChart` can be displayed together, for example:

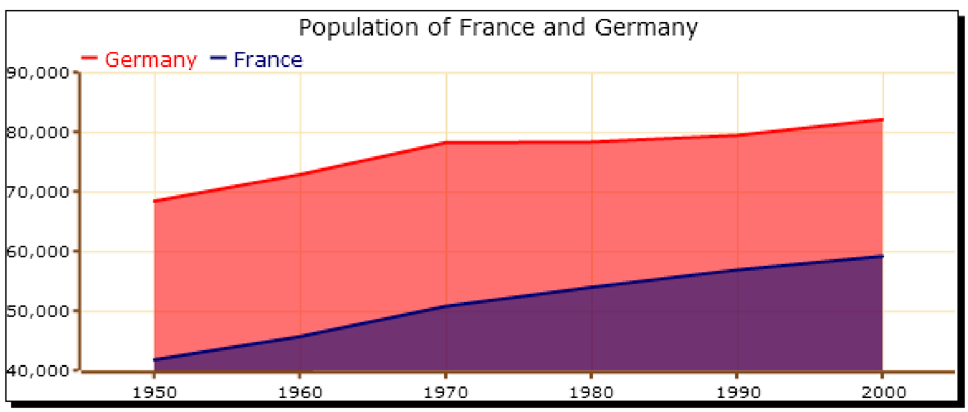
```
BarChart germanyChartConfig = new BarChart();
germanyChartConfig.addValues(68376,72815,78169,78289,79433,82075);
germanyChartConfig.setColour("#ff0000");
germanyChartConfig.setText("Germany");
chartModel.addChartConfig(germanyChartConfig);

LineChart franceChartConfig = new LineChart();
franceChartConfig.addValues(41832,45674,50771,53950,56842,59128);
franceChartConfig.setColour("#000066");
franceChartConfig.setText("France");
chartModel.addChartConfig(franceChartConfig);
```



AreaChart class

`AreaChart` extends `LineChart` and works in the same way. The difference is that instead of a single line, data is displayed as a filled area:

```
AreaChart germanyChartConfig = new AreaChart();
germanyChartConfig.addValues(68376,72815,78169,78289,79433,82075);
germanyChartConfig.setColour("#ff0000");
germanyChartConfig.setText("Germany");

AreaChart franceChartConfig = new AreaChart();
franceChartConfig.addValues(41832,45674,50771,53950,56842,59128);
franceChartConfig.setColour("#000066");
franceChartConfig.setText("France");
```



ScatterChart class

`ScatterChart` is a chart type available in GXT, but not something that is fully supported. We can set the data like this:

```
ScatterChart chartConfig = new ScatterChart();
chartConfig.addPoint(41832, 68376);
chartConfig.addPoint(45674, 72815);
chartConfig.addPoint(50771, 78169);
chartConfig.addPoint(53950, 78289);
chartConfig.addPoint(56842, 79433);
chartConfig.addPoint(59128, 82075);
```

However, there is no way of defining how the actual data will be rendered on the graph so that they do not appear until we pass the mouse over them. To use `ScatterChart` properly, we would need to extend GXT ourselves, and that is beyond the scope of this book.

StackedBarChart class

`StackedBarChart` also is not fully implemented in GXT at the time of writing.

Pop quiz – match the chart feature to the chart

Given the list of chart-related components, match each with the most suitable description:

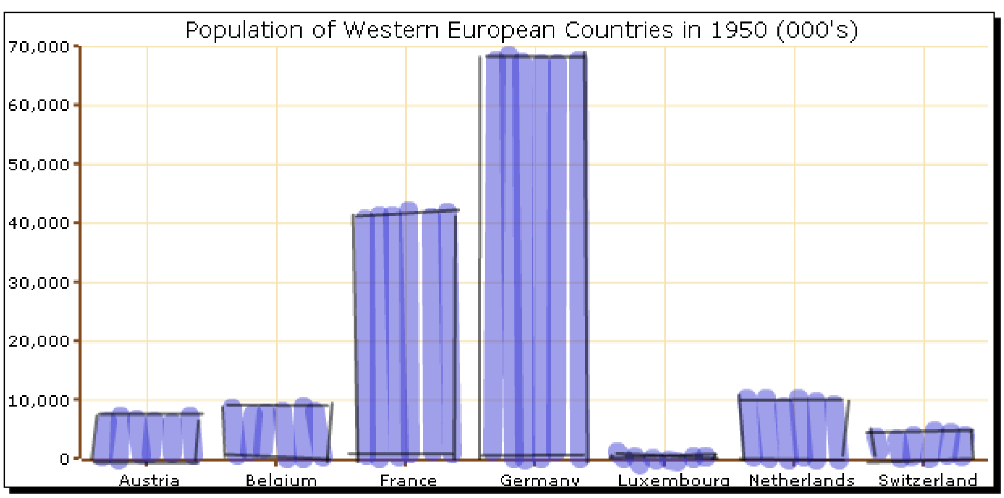
1. What charting system does GXT wrap to produce charts?
2. What would cause this error?

   ```
   [WARN] 404 - GET /gxt/chart/open-flash-chart.swf (127.0.0.1) 1416
   bytes
   ```

3. What would cause this error?

   ```
   [WARN] 404 - GET /gxt/flash/swfobject.js (127.0.0.1) 1408 bytes
   ```

4. Which type of chart has style attributes called `THREED` and `GLASS`?
5. What class is used for defining how the vertical axis of a chart will be displayed?
6. What type of chart is this?



7. What line of code would add a color to a `BarChart` line?
8. What line of code would define colors for a `PieChart`?

a. `SketchBarChart`.

b. `YAxis`.

c. Open Flash Charts 2.

d. Forgetting to include the flash resource folder in the project.

e.
```
chartConfig.setColours("#FF0000", "#FFA500", "#FFFF00",
"#008000",
     "#0000FF","#4B0082", "#EE82EE");
```

f.
```
chartConfig.addBars(new BarChart.Bar(6936, "#FF0000"));
```

g. `BarChart`.

h. Forgetting to include the chart resource folder in the project.

Using a PieChart

Our example application does not display any chart data at the moment. An RSS reader application does not have many uses for charts, but we do have suitable data. Let's create a chart that takes a feed and shows a distribution of the days of the week when items were published, and display it as a `PieChart`.

Time for action – creating PieChart data

1. In `FeedChart`, add the following `prepareData` method. This is not part of the chart itself, but takes a list of items and counts the number of occurrences of each day to provide the data for use in the chart:

```
private HashMap<String, Integer> prepareData(List<Item> items) {
  HashMap<String, Integer> days = new HashMap<String,
    Integer>();
  for (Item item : items) {
    DateTimeFormat fmt = DateTimeFormat.getFormat("EEEE");
    String day = fmt.format(item.getPubDate());
    Integer dayOccurance = days.get(day);
    if (dayOccurance == null) {
      days.put(day, 1);
    } else {
      days.put(day, ++dayOccurance);
    }
  }
  return days;
}
```

2. Create a new method named `createChartModelData` that takes a list of items as a parameter and returns a `ChartModel`:

```
private ChartModel createChartModelData(List<Item> items) {
```

3. Create a new instance of `ChartModel` including the title of the chart and the formatting, set the background color, and return the `ChartModel`:

```
private ChartModel createChartModelData(List<Item> items) {
  ChartModel chartModel = new ChartModel("Posts per week of day",
    "font-size: 14px; font-family: Verdana; text-align: center;");
  chartModel.setBackgroundColour("#ffffff");
  return chartModel;
}
```

4. Create a new `PieChart` `ChartConfig` and set colors for the `PieChart`:

```
private ChartModel createChartModelData (List<Item> items) {
  ChartModel chartModel = new ChartModel("Posts per week of day",
    "font-size: 14px; font-family: Verdana; text-align: center;");
  chartModel.setBackgroundColour("#ffffff");

  PieChart pie = new PieChart();
  pie.setColours("#FF0000", "#FFA500", "#FFFF00", "#008000",
    "#0000FF","#4B0082", "#EE82EE");

  return chartModel;
}
```

5. Retrieve the prepared data using the `prepareData` method, and then for each item in the `HashMap`, add a new `PieChart.Slice`. Then add the `PieChart` `ChartConfig` to the `ChartModel`:

```
private ChartModel createChartModelData (List<Item> items) {
  ChartModel chartModel = new ChartModel("Posts per week of day",
    "font-size: 14px; font-family: Verdana; text-align: center;");
  chartModel.setBackgroundColour("#ffffff");

  PieChart pie = new PieChart();
  pie.setColours("#FF0000", "#FFA500", "#FFFF00", "#008000",
    "#0000FF","#4B0082", "#EE82EE");

  HashMap<String, Integer> days = prepareData(items);
  for (String key : days.keySet()) {
    pie.addSlices(new PieChart.Slice(days.get(key), key));
  }
  chartModel.addChartConfig(pie);

  return chartModel;
}
```

6. Create a new public method named `setFeed`. This should take a `Feed` object and use the `FeedService` to load the `Item` objects for the `Feed`. When it retrieves the `Item` objects, it should use the list as a parameter to the `createChartModelData` method to create a `ChartModel`, and in turn, use that to set the `ChartModel` of the `Chart`:

```
public void setFeed(final Feed feed) {
    final FeedServiceAsync feedService = Registry
        .get(RSSReaderConstants.FEED_SERVICE);
    feedService.loadItems(feed.getUuid(), new
      AsyncCallback<List<Item>>() {
      @Override
      public void onFailure(Throwable caught) {
        Dispatcher.forwardEvent(AppEvents.Error, caught);
      }

      @Override
      public void onSuccess(List<Item> items) {
        chart.setChartModel(createChartModelData(items));
      }
    });
  }
```

7. As the `Chart` now has a `ChartModel`, there will not be a data error in the Open Flash Chart. So as part of this method, we can make the chart visible, if it isn't already visible:

```
public void setFeed(final Feed feed) {
  final FeedServiceAsync feedService = Registry
        .get(RSSReaderConstants.FEED_SERVICE);
  feedService.loadItems(feed.getUuid(), new
    AsyncCallback<List<Item>>() {
    @Override
    public void onFailure(Throwable caught) {
      Dispatcher.forwardEvent(AppEvents.Error, caught);
    }

    @Override
    public void onSuccess(List<Item> items) {
      chart.setChartModel(createChartModelData(items));
    }
  });
  if (!chart.isVisible()) {
    chart.setVisible(true);
  }
}
```

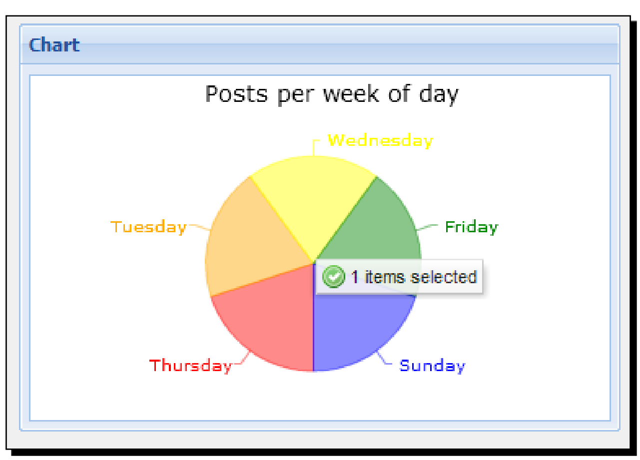
8. In the `ChartPortlet` class, add a method named `onFeedsDropped` method that extracts the feed from the drop event and use it to set the feed for the chart:

```
private void onFeedsDropped(DNDEvent event) {
  List<Feed> feeds = event.getData();
  for (Feed feed : feeds) {
    feedChart.setFeed(feed);
  }
}
```

9. Overwrite the `onRender` method in the same way that we did in the last chapter for the `FeedPortlet` to make the `ChartPortlet` act as another `DropTarget` in the `FEED_DD_GROUP`:

```
@Override
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  DropTarget target = new DropTarget(this) {
    @Override
    protected void onDragDrop(DNDEvent event) {
      super.onDragDrop(event);
      onFeedsDropped(event);
    }
  };
  target.setOperation(DND.Operation.COPY);
  target.setGroup(RSSReaderConstants.FEED_DD_GROUP);
}
```

10. Start the application and drag-and-drop a feed from the `NavPortlet` to the `ChartPortlet` to generate a chart from the data:

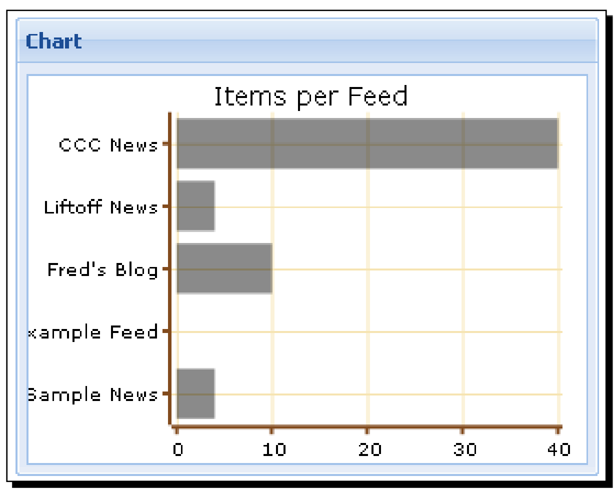


What just happened?

We added a `PieChart` configuration to the `FeedChart` to allow for data to be rendered as a chart and enabled the `ChartPortlet` as a `DropTarget` to pass the data to the `FeedChart`.

Have a go hero – creating an item count bar chart

Using the chart we have just created as a guide. Create a component that uses a `HorizontalBarChart` to display the number of items in each feed. Name the component `ItemCountChart` and use the `loadFeedList` method of the `FeedService` with the `true` parameter to retrieve the raw data. The result should look something like this:



Solution:

```
public class ItemCountChart extends LayoutContainer {

  private final Chart chart = new Chart("gxt/chart/open-flash-
    chart.swf");

  public ItemCountChart() {
    chart.setVisible(false);
    final FeedServiceAsync feedService = Registry
      .get(RSSReaderConstants.FEED_SERVICE);
    feedService.loadFeedList(true, new AsyncCallback<List<Feed>>() {
      @Override
      public void onFailure(Throwable caught) {
        Dispatcher.forwardEvent(AppEvents.Error, caught);
      }
```

```
    @Override
    public void onSuccess(List<Feed> feeds) {
      chart.setChartModel(createChartModelData(feeds));
      chart.setVisible(true);
    }
  });
}

private ChartModel createChartModelData(List<Feed> feeds) {
  ChartModel chartModel = new ChartModel("Items per Feed",
    "font-size:14px;color:#000000");
  chartModel.setBackgroundColour("#ffffff");

  HashMap<String, Integer> data = prepareData(feeds);

  YAxis yAxis = new YAxis();
  for (String key : data.keySet()) {
    yAxis.addLabels(key);
  }
  yAxis.setOffset(true);

  chartModel.setYAxis(yAxis);
  XAxis xAxis = new XAxis();
  xAxis.setRange(0, 50, 10);
  chartModel.setXAxis(xAxis);

  HorizontalBarChart chartConfig = new HorizontalBarChart();
  List<Number> reverseValues = new
    ArrayList<Number>(data.values());
  Collections.reverse(reverseValues);
  chartConfig.addValues(new ArrayList<Number>(reverseValues));
  chartModel.addChartConfig(chartConfig);

  return chartModel;
}

@Override
protected void onRender(Element parent, int index) {
  super.onRender(parent, index);
  setLayout(new FitLayout());
  chart.setBorders(true);
  add(chart);
}
```

```
    private HashMap<String, Integer> prepareData(List<Feed> feeds) {
      HashMap<String, Integer> counts = new HashMap<String, Integer>();
      for (Feed feed : feeds) {
        String feedTitle = feed.getTitle();
        int itemCount = feed.getItems().size();
        counts.put(feedTitle, itemCount);
      }
      return counts;
    }
  }
```

Summary

In this chapter, we have looked into GXT's charting features. We have learnt that charts are more of an extension to GXT than core functionality and examined how to overcome potential pitfalls when setting up charts. We then went on to investigate the different charts available. Finally, we made use of a chart in our example application.

10
Putting It All Together

In the previous chapters, we have developed an example application using GXT. In this chapter, we will learn how to publish it to the world, using Google App Engine. We will then move on to look at how you can take your development further with GXT, and other resources you can turn to, once you have finished this book.

Specifically, we will cover the following subjects:

- Deploying to Google App Engine
- Creating an application shortcut in Google Chrome
- The possibilities offered by Google Gears
- Options for using GXT for mobile applications
- The future of GXT
- Sources of further information

Using Google App Engine

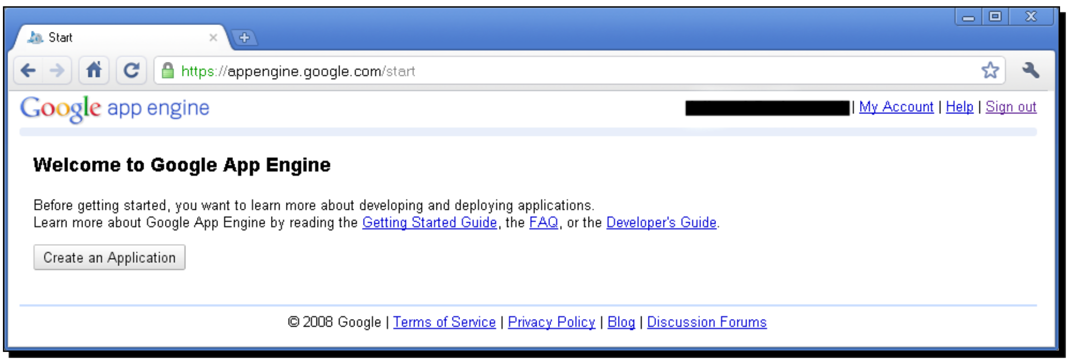
Google App Engine for **Java** (**GAE/J**) is Google's cloud computing platform. It is an excellent companion to GWT as it provides an easy way of hosting applications. It is also a perfect platform for hosting GXT applications such as our RSS Reader.

Before we get started with Google App Engine, we need to create an account and register an application.

Time for action – registering a Google App Engine application

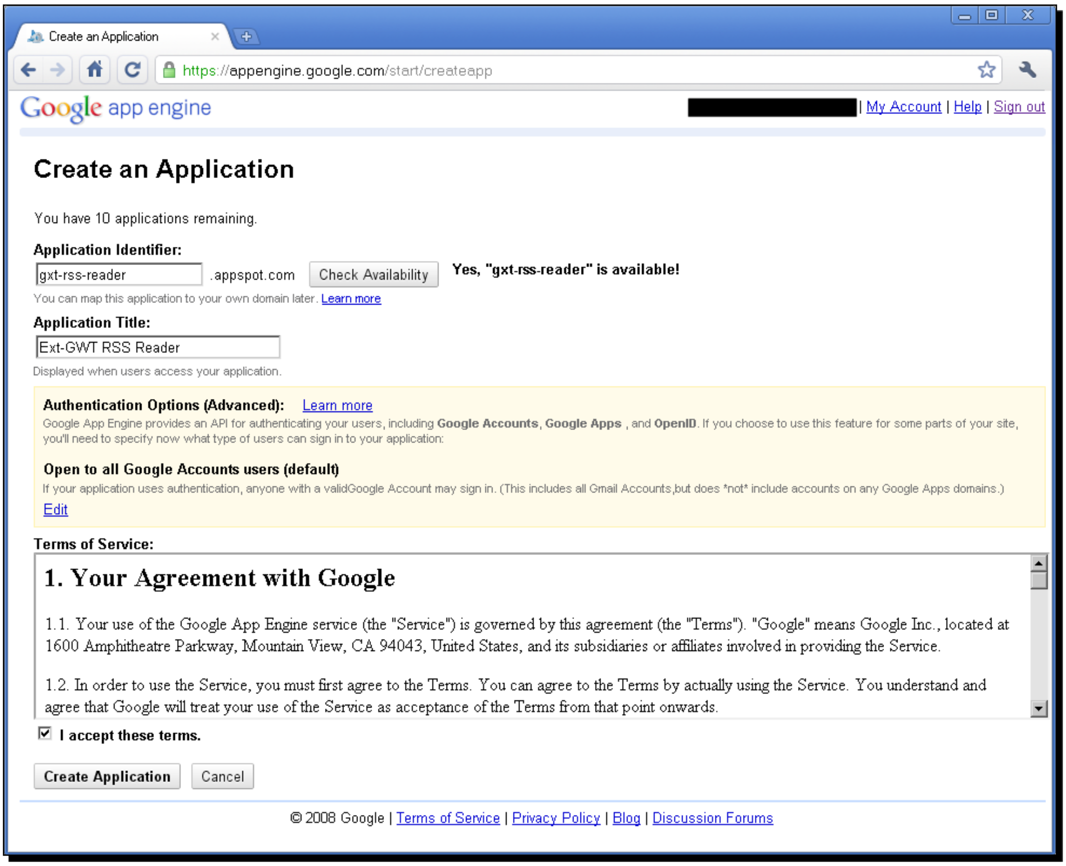
1. Go to the Google App Engine (GAE) website at `http://appengine.google.com/` and log in using your Google ID. If you don't already have an account, you can also sign up for one there.

2. Once you have logged into GAE, you will see the following screen. Click on the **Create an Application** button.



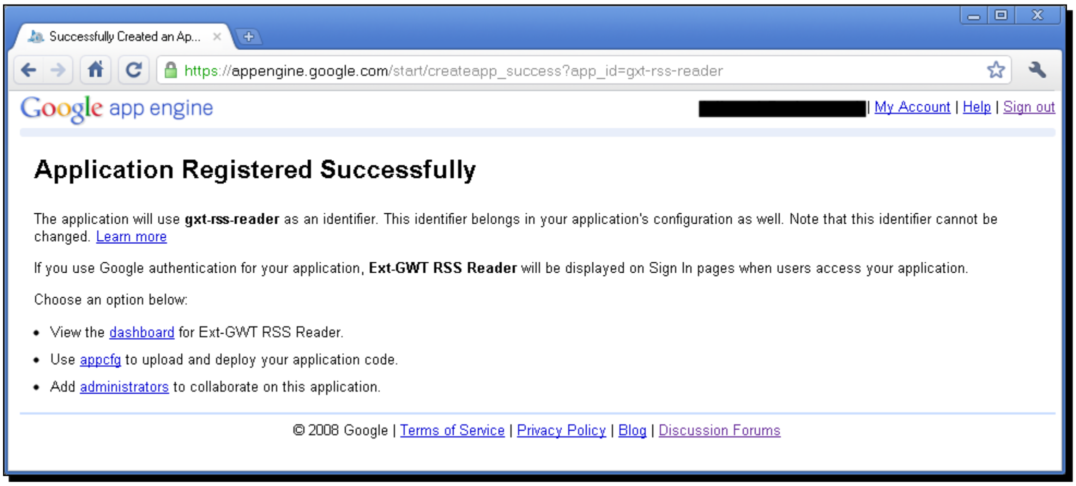
3. After clicking on the **Create an Application** button, you may be asked to verify your account at this point, if you have created a new Google account.

4. You will be presented with the following form. Enter the **Application Identifier**. This is a unique ID for your application across GAE. So while in the screenshot **gxt-rss-reader** is the app ID, your ID will need to be different. In this case, however, the application will be hosted at `http://gxt-rss-reader.appspot.com`. You also need to enter a title for the application.



5. Leave the authentication options as default, read and accept the **Terms of Service**, and finally, click on the **Create Application** button.

6. If successful, you will receive the following message and you will have an application to deploy your code to:



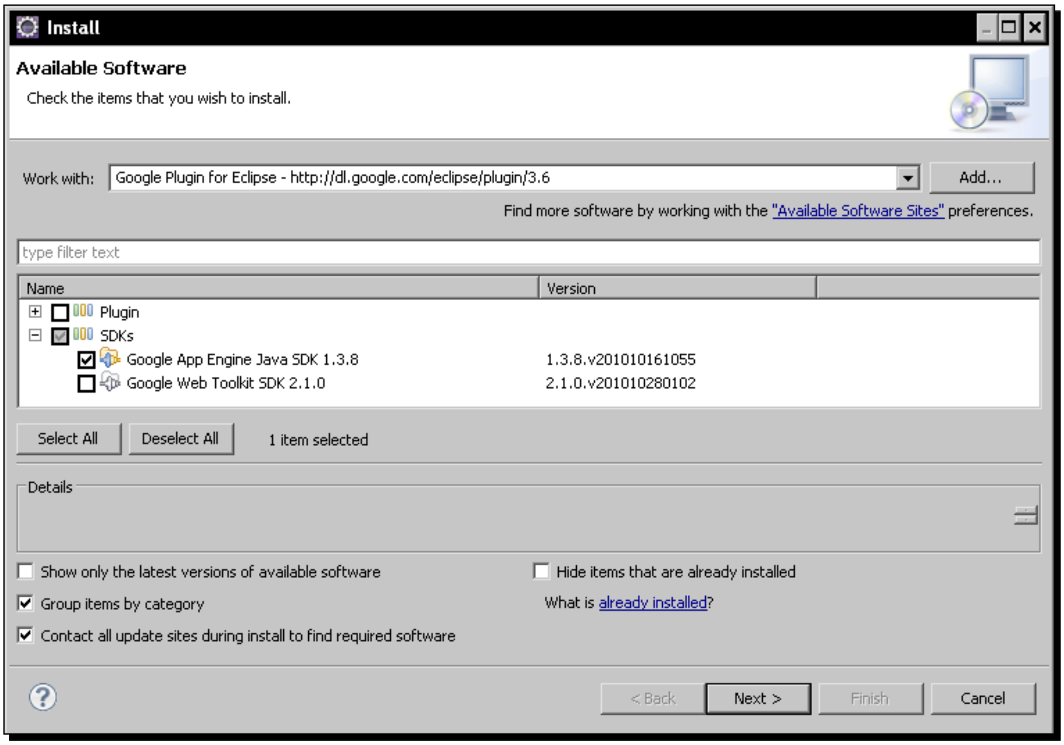
What just happened?

We created an application on Google App Engine, but as yet, it is just an empty container. We need to GAE-enable our example application. However, before we can do that, we need to add the GAE SDK to our Eclipse setup.

In Chapter 1, *Getting Started with Ext GWT*, we installed the Google Plugin for Eclipse and the Google Web Toolkit SDK. Now, in a similar way, we need to install the Google App Engine Java SDK.

Time for action – getting the application ready for GAE

1. In Eclipse, select **Help** | **Install New Software**. The install dialog will be displayed, as shown in the next screenshot:



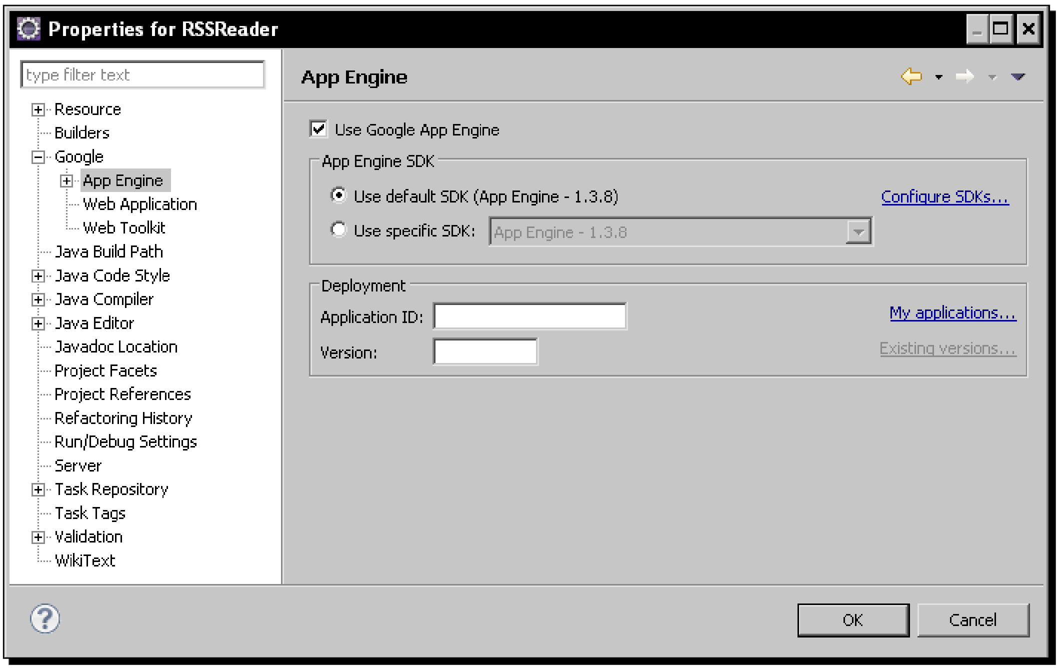
2. Select **Google Plugin for Eclipse** from the **Work with** list. Tick the **Google App Engine Java** option under **SDKs** and click on the **Next** button.

3. Continue till the end of the wizard and restart Eclipse when prompted.

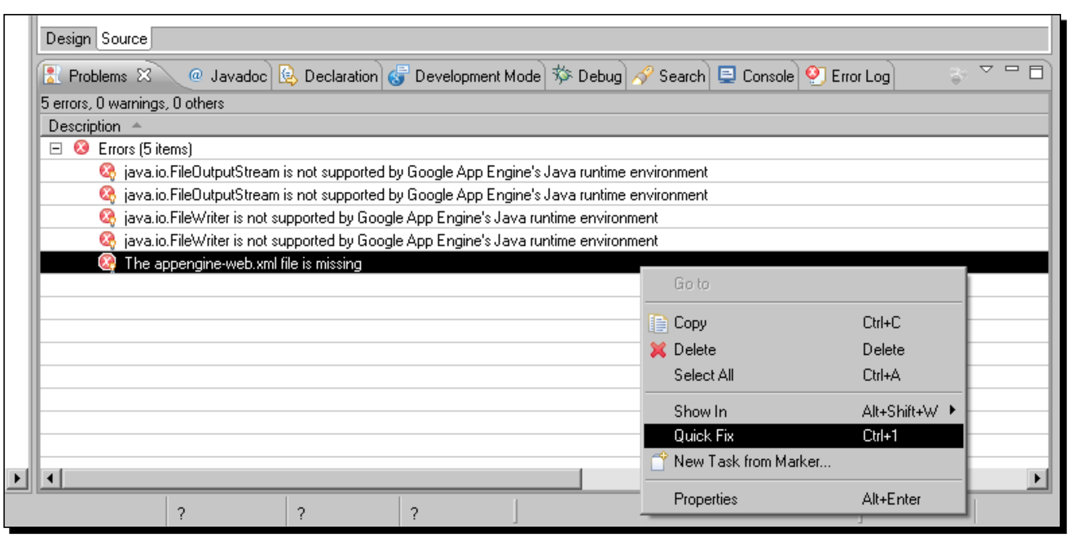
4. When Eclipse restarts, right-click on the example application project and select **Properties**.

5. In the properties dialog, select the **Google** item from the tree and then **App Engine** entry.

6. Tick the **Use Google App Engine** option.



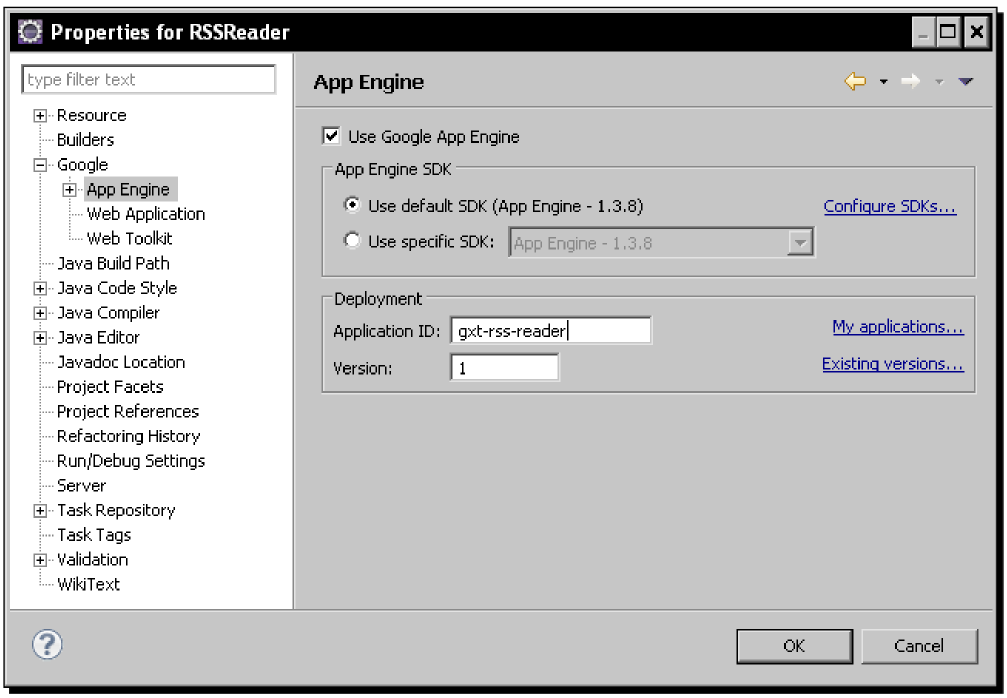
7. Click on the **OK** button. The application will not re-build, but the build will result in a number of errors that will appear in Eclipse's problem panel.



8. The last error, **The appengine-web.xml file is missing**, is easy to fix. Right-click on it and select **Quick Fix**.

9. You will be given an option to create a new file `appengine-web.xml`—**Create a new appengine-web.xml**. Select this option and click on **Finish**. The Google Plugin will create the missing file for you.

10. Return to the projects **App Engine** properties dialog, enter the name of the application identifier of the application you previously registered in the **Application ID** field, and click on the **OK** button.



What just happened?

We installed Google App Engine and enabled it for our example application. However, we still have some compile errors. This is because Google App Engine has some limitations. Fortunately, none of these limitations affect GXT, and we can make a few small changes to the Example Application and it will work perfectly on GAE.

The errors we have are because we cannot write to the filesystem of the GAE server, which is one of the limitations of GAE. However, GAE makes up for this by providing a mechanism for persisting data to a provided data store.

There are several ways of persisting to the data store, but for this application we are going to use JDO, as it is the most straightforward. We are going to create a GAE implementation of our backend persistence interface in our example application.

Note that we will only be implementing the ability to save the feed list and hence we will only be able to save references to the existing feeds and not create new feeds as XML files. This is because the GAE does not give direct access to the filesystem.

Time for action – using the Google App Engine data store

1. First, we need to make sure that the `jdoconfig.xml` file is present in a directory named `META-INF` in the project's source folder. This is normally created automatically when you create a GAE project in Eclipse. However, as we added GAE to the project later, it may be missing. If it is, add the file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<jdoconfigxmlns="http://java.sun.com/xml/ns/jdo/jdoconfig"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="http://java.sun.com/xml/ns/jdo/
jdoconfig">

  <persistence-manager-factory name="transactions-optional">
    <property name="javax.jdo.PersistenceManagerFactoryClass"
          value="org.datanucleus.store.appengine.jdo.
DatastoreJDOPersistenceManagerFactory"/>
    <property name="javax.jdo.option.ConnectionURL"
              value="appengine"/>
    <property name="javax.jdo.option.NontransactionalRead"
              value="true"/>
    <property name="javax.jdo.option.NontransactionalWrite"
              value="true"/>
  <property name="javax.jdo.option.RetainValues" value="true"/>
  <property name="datanucleus.appengine.autoCreateDatastoreTxns"
            value="true"/>
  </persistence-manager-factory>
</jdoconfig>
```

2. We now need to create a simple singleton class to provide an instance of `PersistenceManagerFactory` that we will use for persistence. Create a final class with the name `PMF` in the `server.utils` package, which is implemented as follows:

```
public final class PMF {
  private static final PersistenceManagerFactory pmfInstance =
JDOHelper
      .getPersistenceManagerFactory("transactions-optional");
```

```
  private PMF() {
  }

  public static PersistenceManagerFactory get() {
    return pmfInstance;
  }
}
```

3. Previously, we simply saved the text of the URLs of the RSS feeds directly to a file. In GAE, we must wrap our data in a JavaBean. So create a class named `FeedUrl` and a new package named `server.model` as follows:

```
public class FeedUrl {

  private String url;

  public FeedUrl(String url) {
    this.setUrl(url);
  }

  public void setUrl(String url) {
    this.url = url;
  }

  public String getUrl() {
    return url;
  }
}
```

4. We now need to make the JavaBean persist-able by adding an `@PersistenceCapable` annotation to the class and an `@PrimaryKey` annotation to the primary key field.

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class FeedUrl {
  @PrimaryKey
  private String url;

  public FeedUrl(String url) {
    this.setUrl(url);
  }

  public void setUrl(String url) {
    this.url = url;
```

```
  }

  public String getUrl() {
    return url;
  }
}
```

5. Create a new class named `GaePersistence` in the `server.utils` package that implements the `Persistence` interface.

```
public class GaePersistence implements Persistence {
```

6. In the `GaePersistence` class, implement the `saveFeedList` method so that it takes URL strings past to it, creates a `FeedUrl` JavaBean for each string, and makes it persistent using the persistence manager.

```
@Override
public void saveFeedList(Set<String>feedUrls) {
  PersistenceManager pm = PMF.get().getPersistenceManager();
  try {
    for (String url : feedUrls) {
      FeedUrl feedUrl = new FeedUrl(url);
      pm.makePersistent(feedUrl);
    }
  } finally {
    pm.close();
  }
}
```

7. In a similar way, implement the `loadFeedList` method by using the persistence manager to retrieve the URLs of the feeds from the GAE persistence store.

```
@SuppressWarnings("unchecked")
@Override
public Set<String> loadFeedList() {
  PersistenceManager pm = PMF.get().getPersistenceManager();
  try {
    Set<String>urls = new HashSet<String>();
    Query q = pm.newQuery("select url from " + FeedUrl.class.
getName());
       List ids = (List) q.execute();
    urls.addAll(ids);
    return urls;
  } finally {
    pm.close();
  }
}
```

8. In the `FeedServiceImpl` class, change the persistence field so that it instantiates as a `GaePersistence` object instead of a `FilePersistence` object.

```
public class FeedServiceImpl extends RemoteServiceServlet
implements
  FeedService {

 private final static Logger LOGGER = Logger.
getLogger(FeedServiceImpl.class
   .getName());

 private Map<String, Feed> feeds = new HashMap<String, Feed>();

 private final Persistence persistence = new GaePersistence();
```

9. Finally, remove the now redundant `FilePersistence` class and with it the compile errors.

What just happened?

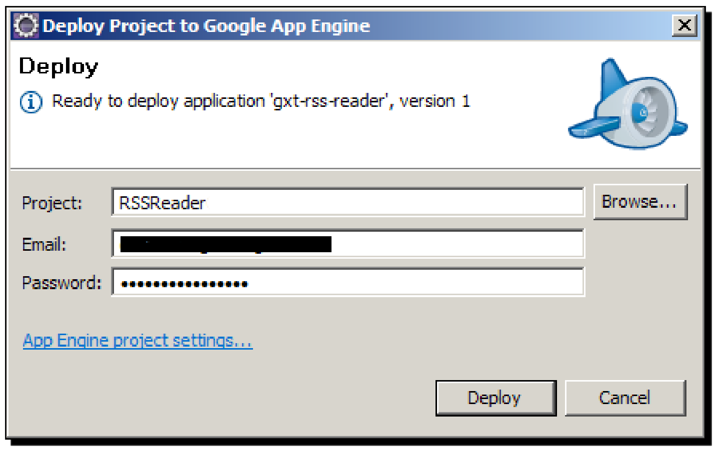
In order to store the data in Google App Engine, we created a persistence solution that makes use of GAE's persistence store instead of the filesystem.

We are now ready to publish the application to Google App Engine, which is surprisingly straightforward.

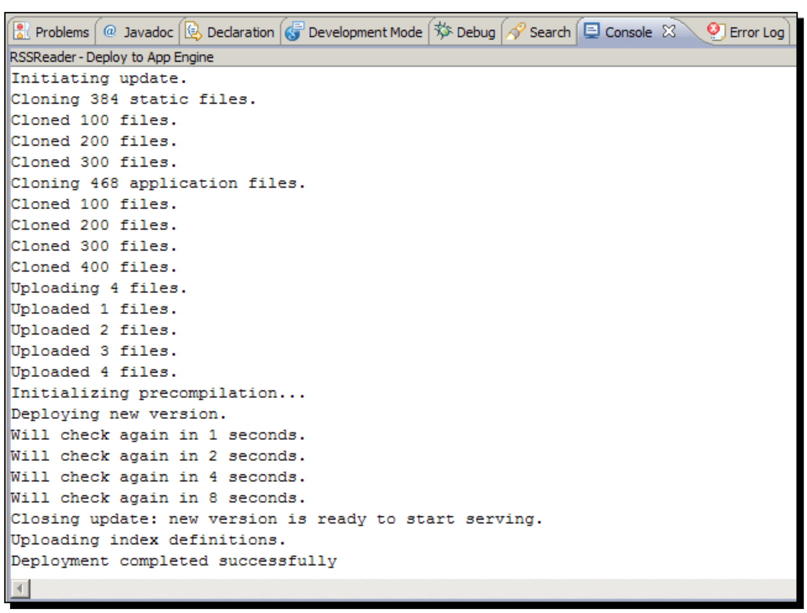
Time for action – publishing the example application

1. In Eclipse, there is a row of toolbar buttons related to the Google Plugin. The third button is **Deploy App Engine Project**. With the example application project selected, click on this button.

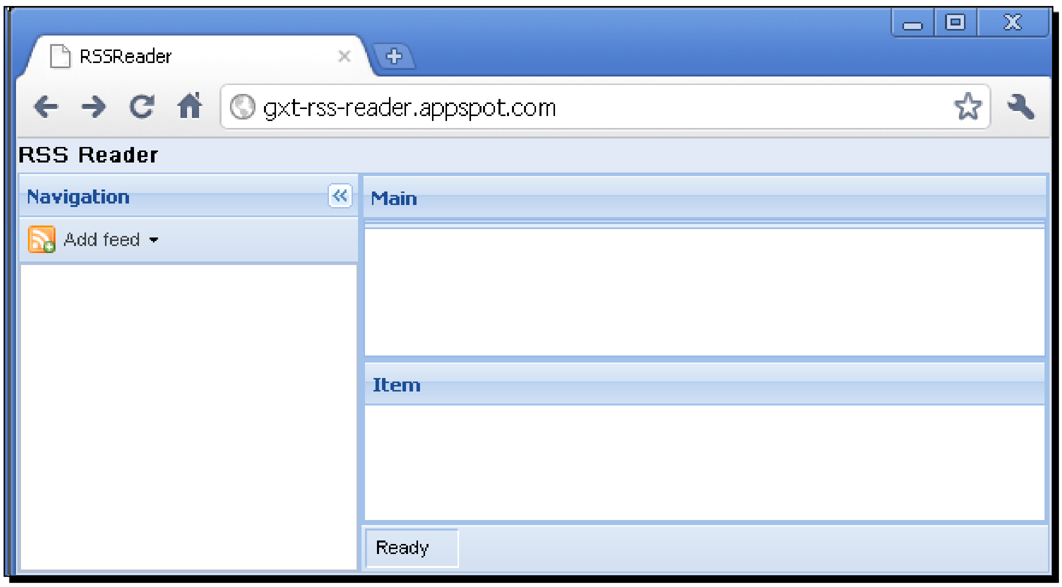
2. You will be prompted with the following dialog box. Enter the e-mail address and password of the Google App Engine account you used earlier in order to register the application and click on the **Deploy** button.



3. The Google Plugin will now automatically compile the application and upload the generated files to Google App Engine. Progress is output to the console, and when finished, the message **Deployment completed successfully** will be displayed.



4. Once deployed, check that the application is available on the web. The URL will be in the format `http:/<application-id>.appspot.com/`, and in this case, `http://gxt-rss-reader.appspot.com/`.



What just happened?

We deployed our example application onto the Google App Engine, making it available publically on the web.

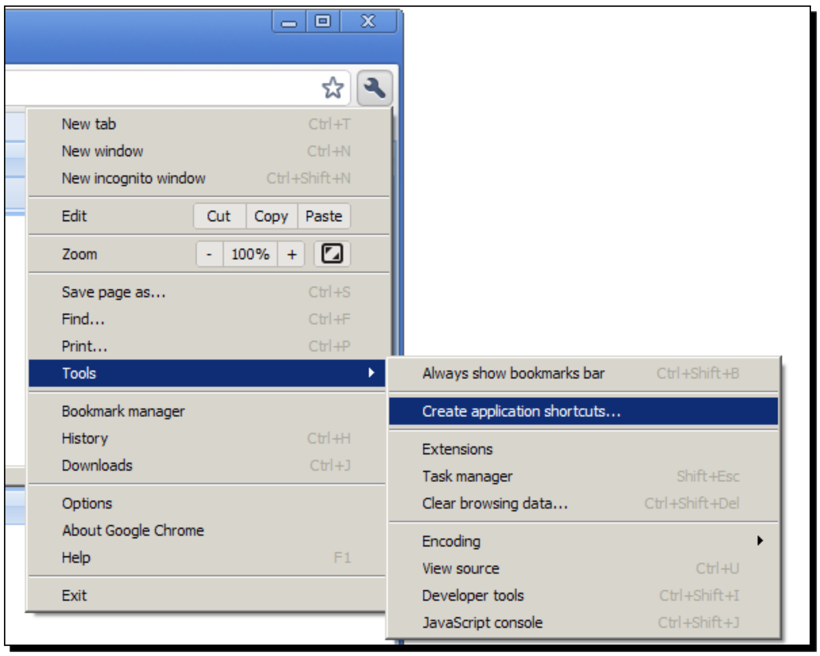
Google Chrome

Google Chrome is Google's own web browser and is optimized for running the JavaScript that GWT and GXT applications consist of. Google Chrome is available for free. You can download it from `http://www.google.com/chrome`.

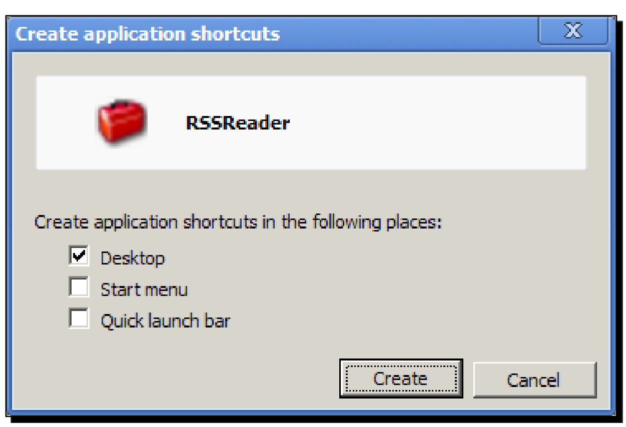
One of the great features of Google Chrome is its ability to create application shortcuts to make web applications appear like normal desktop applications.

Time for action – creating a Google Chrome application shortcut

1. In Google Chrome, browse to your deployed Google App Engine application, click the settings icon, and select **Tools** | **Create application shortcuts...** from the menu.

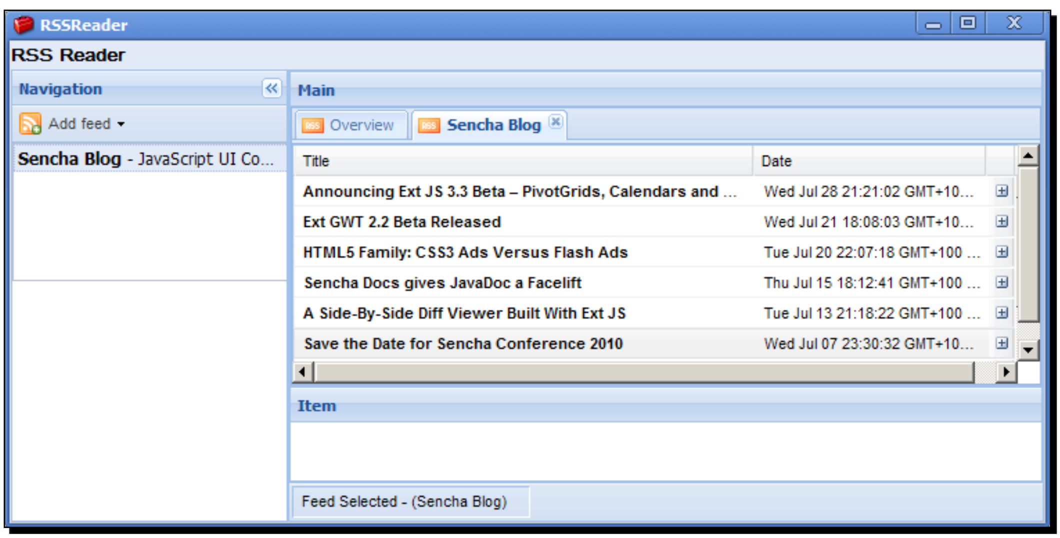


2. From the dialog box that is displayed, choose the types of shortcuts that you want to create, in this case, just a desktop shortcut.



3. Click on the **Create** button and a new shortcut will be created on your desktop.

4. Click on the shortcut and the example application will start. Notice that now it looks more like a desktop application rather than one running in a browser. There is no address bar or other browser user interface around it.



What just happened?

We used the Google Chrome browser to create a desktop shortcut to our example application running on Google App Engine. When started, it looks like a desktop application. Would an average user even realize they are using a web application?

Have a go hero: ideas for doing even more

GWT and GXT is a great platform for building applications. However, there are lots of other technologies out there that can be incorporated to give the platform even more potential.

If you would like to start looking at the possibilities, we suggest the following technologies would be a good start.

Gears

Gears is another open source Google-provided library that adds a selection of new features to web browsers. These allow the application to run offline without an internet connection together with a number of other enhancements. Specifically, Gears includes the following modules.

- Database module: This allows the local storage of data using SQLite
- WorkerPool module: This allows for the parallel execution of JavaScript code
- LocalServer module: caches and serves the HTML, JavaScript, and images of an application locally
- Desktop module: This allows the Web to interact with the desktop of the client machine
- Geolocation module: This allows the web application to determine the geographical location of the user

To use Gears with GWT, download the Gears API library for GWT. It is available at `http://code.google.com/p/gwt-google-apis/`. The GWT library doesn't support every feature of Gears at the time of writing, but it does offer many useful features.

Mobile applications

In the same way as Gear can make the GXT web applications available offline and Chrome can make the same applications appear more like desktop applications, other technologies can do this on mobile devices. These allow web applications to run locally on mobile phones or tablet devices and appear like native mobile applications.

PhoneGap

PhoneGap is an open source mobile application development framework. It allows the developers to take the HTML, JavaScript, and image files like those produced using GXT and build them into native applications for a wide range of mobile phone operating systems.

More information about PhoneGap can be found at `http://www.phonegap.com/`.

Widgets

Widgets are another way of running a web application locally on a device such as a native application rather than on a server. The device could be a mobile phone or devices such as a TV set-top box or just a normal desktop computer. The examples of this technology you might like to investigate are:

- Opera Widgets at `http://widgets.opera.com/`
- Nokia WRT at `http://www.forum.nokia.com/Develop/Web/`

The future for GXT

There is little point in writing about the future of GXT, as this sort of technology moves so fast that anything written will soon be superseded. However, at the time of writing, Sencha had just announced that a touch interface for GXT was under development. This would make GXT a very attractive tool for developers of touchscreen-based mobile phones and tablet devices.

If, at any time, you want to look at what is coming up in GXT, check out the road map on the Sencha website at `http://www.sencha.com/products/gwt/roadmap.php`.

Getting more information

This book is intended to be a comprehensive introduction to GXT but it cannot cover everything. As you continue with GXT, you may find the following resources useful.

GXT Explorer website

The GXT Explorer (at `http://www.sencha.com/examples`) that we introduced in Chapter 2, *The Building Blocks*, is the best place for finding additional examples of how to use the GXT components.

GXT sample code

In the GXT distribution, there is a `samples` directory that contains the source code for the GXT explorer used in the showcase plus additional examples. These can be useful when you want to learn more about how components fit together than what is possible by just looking at the snippets of code provided on the website.

GXT Java doc

The Java doc for GXT is the place to look for detailed information about the API structure of GXT. It is available both in the `docs` folder of the distribution and online in Sencha's Docs application at `http://www.sencha.com/gxtdocs/`.

GXT Help Eclipse plugin

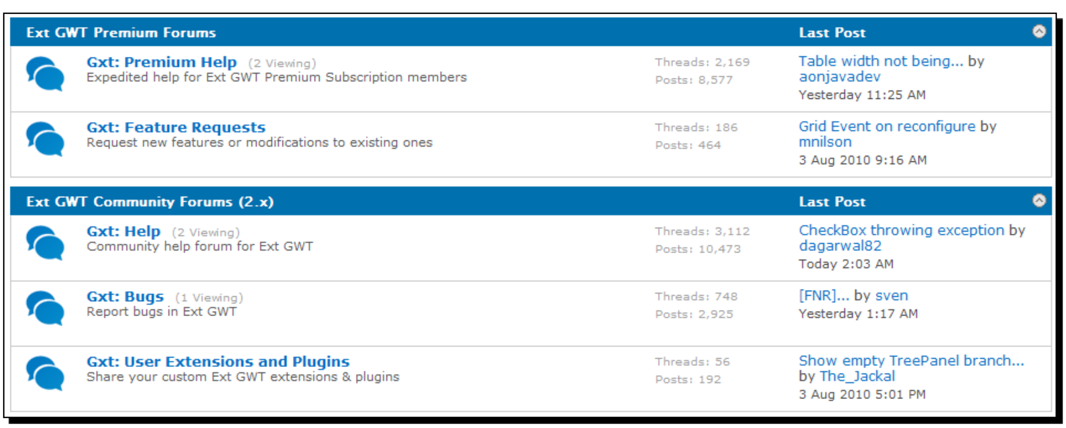
At the time of writing, Sencha has released a basic GXT help plugin for Eclipse. This can be downloaded from the GXT eclipse update site, mentioned here: `http://dev.sencha.com/deploy/gxt-update-site/`. The help is, at present, a little sparse, but should improve as time goes on.

GXT source code

GXT is an open source project meaning that the source code is there for you to look at or even modify, should you desire. Understanding how the source code works is an excellent way to start understanding GXT in depth. It can also be useful when you have a problem and want to see what is going on. The source code can be found in the `src` folder of the distribution.

GXT forums

If you have a specific question about GXT, there are a number of forums on the Sencha website at `http://www.sencha.com/forum/`. There are two sets of forums—**community forums** (where anyone can post questions) and the **premium forums** (where only those with a GXT-support license can post questions). Anyone can view the past questions on both the sets of forums, and it is a good idea to search the forums before posting to make sure your question hasn't been answered already. The forum users include experienced GXT users as well as some of the developers of GXT, so there is a good chance you will get a helpful answer for even the most complex question.



Other programmer forums

Other programming websites also feature questions about GXT. The site `http://stackoverflow.com/` is one of the best sites and GXT questions can be found tagged as follows:

- With the GXT tag at this address: `http://stackoverflow.com/questions/tagged/GXT`
- With the GXT tag at this address:

  `http://stackoverflow.com/questions/tagged/gxt`

PopQuiz: Finding additional information

In addition to this book, where could you look if you required:

1. To find out the parameters of a particular GXT method.
2. To ask a question to the developers of GXT.
3. To ask a question to other developers that use GXT.
4. To search previous answers asked about GXT.
5. To work out in detail how a particular GXT component works.
6. To get an example of how a particular GXT component is used.
7. To find out what new features are planned for GXT.
8. To learn about what components are available.
9. To find out if a problem you are having is a bug in GXT.
10. To attempt to fix a bug in GXT.

a. GXT Explorer website
b. GXT Sample code
c. GXT Java doc
d. GXT help plugin
e. GXT source code
f. GXT forums
g. Other programming website like Stack Overflow
h. The GXT roadmap

Summary

GXT provides a rich set of components that work with GWT to provide a whole new set of opportunities in web application development. It is a powerful and constantly developing platform that opens doors to all sorts of possibilities.

We hope you have enjoyed your journey through GXT and are inspired to create some great Rich Internet Applications.

Pop Quiz Answers

Chapter 1

Introducing GXT

1	Ext JS.
2	Smart GWT.
3	Vaadin.
4	GWT-Ext.
5	Sencha.
6	gxt.jar.
7	Dual GPL and commercial.
8	The GWT module's gwt.xml module file.
9	The GWT module's HTML file.
10	The project's war\WEB-INF\lib folder.

Chapter 2

Matching the component with the description

1	2	3	4	5	6	7	8	9	10
g	c	b	h	a	i	e	d	f	j

Chapter 3

Match the form components with their definitions

1	2	3	4	5	6	7	8	9	10
h	c	i	e	g	f	a	b	j	d

Chapter 4

Right tool for the job

	DataProxy	**DataReader**	**Loader**
1	HttpProxy	XmlLoadResultReader	BaseListLoader
2	RpcProxy	BeanModelReader	BasePagingLoader
3	MemoryProxy	ModelReader	BaseListLoader
4	HttpProxy	JsonLoadResultReader	BasePagingLoader
5	ScriptTagProxy	ScriptTagProxy	BaseListLoader

Chapter 5

Matching the component with the definition

1	2	3	4	5	6	7	8	9	10
b	j	h	i	c	e	f	d	a	g

Chapter 6

What does what?

1	2	3	4	5	6	7
c	f	d	a	b	e	g

Chapter 7

MVC Fundamentals

1	2	3	4	5	6	7	8
a	b	b & c	d	b & c	a	a	b

Chapter 8

Quick Q&A

1	2	3	4	5	6	7	8
g	d	a	c	e	b	h	f

Chapter 9

Match the chart feature to the chart

1	2	3	4	5	6	7	8
c	h	d	g	b	a	f	e

Chapter 10

Finding additional information

1	2	3	4	5	6	7	8	9	10
c, d or e	f	f or g	f or g	e	a or b	h	a, b, d	e or f	c and e

Index

Symbols

E

F

G

H

I

S

T

Thank you for buying

Ext GWT 2.0: Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.