SAMPLE CHAPTER

GWT IN ACTION

SECOND EDITION

Adam Tacy
Robert Hanson
Jason Essington
Anna Tökke

MANNING

GWT in Action
Second Edition

by Adam Tacy
Robert Hanson
Jason Essington
Anna Tökke

Chapter 2

brief contents

PART 3 ADVANCED 457

2 Building a GWT application: saying "Hello World!"

This chapter covers

- Understanding GWT terminology
- Using the Google Plugin for Eclipse
- Running an application in development mode
- Compiling an application for production
- Exploring common debugging techniques

Following directly from chapter 1, we'll now put the development tools and plug-ins you installed to use. In the next two chapters you'll create and run the GWT application shown in the mock-up in figure 2.1.

Figure 2.1 shows a fairly simple application with a logo, a tabbed panel with some content, a search button, and a slide-in feedback tab. Although simple, it's typical of a common web application, and it will allow you to see the use of widgets, panels, events, styling, adding items to the browser page, and managing history (clicking the browser's Back and Forward buttons) in action. Don't worry if those terms mean nothing to you now; by the end of chapter 3 it will all be second nature.

We'll also show how a typical GWT application is developed in two steps:

1. Use some tools to create a basic framework GWT application.
2. Enhance the framework GWT application to become the application you want.

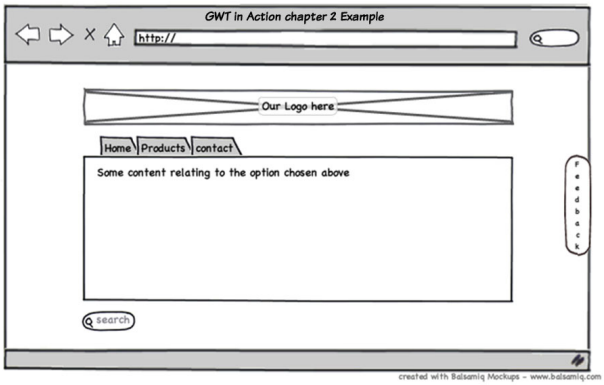

Figure 2.1 A mock-up of the application you'll build in chapters 2 and 3. In this chapter you create the base application using GWT tools, and in chapter 3 you extend it to implement this mock-up.

This chapter covers step 1, and we'll go through the process of creating a simple `HelloWorld` application using some of the wizards from the Google Plugin for Eclipse (GPE). The resulting application is the simplest GWT application that can be built that places some text on a web page. You'll use this to explore the wizards, understand some GWT terminology, get familiar with application code layout and the files involved, and learn how to run and debug the application in Eclipse as well as compile it to JavaScript—the basics of GWT development.

Chapter 3 covers step 2, but rather than take the steps together we'll present a brand-new application, `BasicProject`, and examine the changes that allow it to have the functionality of figure 2.1. `BasicProject` will allow us to examine functionality such as web page (DOM) manipulation, widgets and panels, event handling, history management, and styling—typical techniques in a nontrivial GWT application.

Unusually, we'll hold off discussing client/server interaction until chapters 7, 8, 9, and 12—because GWT gives you a number of approaches and we want to demonstrate these useful client-side techniques, such as client bundles and declarative UI (UiBinder) first.

You could download the results of this chapter from our book's download site, but we recommend instead that you take the steps with us to get some hands-on experience in creating a basic GWT application. The example is available for download in case you decide to experiment with what we're doing along the way and you want a reference to refer to.

When you look at the code you'll create, you may think this simple hello world example contains a lot of overhead—after all, you can do the same in JavaScript with a few lines of code. This is true, but GWT is designed to support pushing the boundary of web applications and user experience, as you'll see from chapter 4 onward. This quality in GWT makes the overhead fade into insignificance because of the benefits of simpler team development, industrial-strength concepts, and reduced and simplified maintenance.

Places to get help

If you do run into problems, don't forget you have a number of places where you can go for help:

The GWT Discussion Group:

http://mng.bz/3a44

The GWT Issue List (to see if someone else is having a similar issue, or if a solution exists):

http://mng.bz/8Kyj

The *GWT in Action* forum:

www.manning-sandbox.com/forum.jspa?forumID=659

If you're a little more technical or interested in what the future holds for GWT, we also recommend the GWT Contributor's Group: http://mng.bz/2BW1

One exercise you might like to do is to take your results from this chapter and expand them to have the functionality of `BasicProject` or whatever functionality you wish by adding your own widgets, panels, or event handling, or altering the styling.

It's almost time to get into the action and create your first GWT application, but before you do that we want to briefly discuss what we mean by a GWT application—it will be good to know what it is from a user's and developer's perspective—and the different ways you can create one (and why we're going to use the Google Plugin for Eclipse).

2.1 What's a GWT application?

We'll try to describe a typical GWT application. We'll highlight the key aspects in this chapter and then expand on them throughout the book. To do this, we'll consider the application from the user's view next; later we'll cover it from the developer's perspective.

2.1.1 Seeing the user's view

The user runs your application by requesting its start web page (which is usually an HTML page but could be a JSP, or a PHP, or whatever). Figure 2.2 shows what happens when a user requests a GWT application's web page.

First, the user enters the URL of your application, which triggers the browser to request the application's HTML file. The HTML downloads a specifically named nocache.js file—the so-called *bootstrap file*. The bootstrap code determines which specific permutation of your application in JavaScript is required and then requests it. When the permutation is loaded, the bootstrap calls the compiled `onModuleLoad` method from the `EntryPoint` and the application starts.

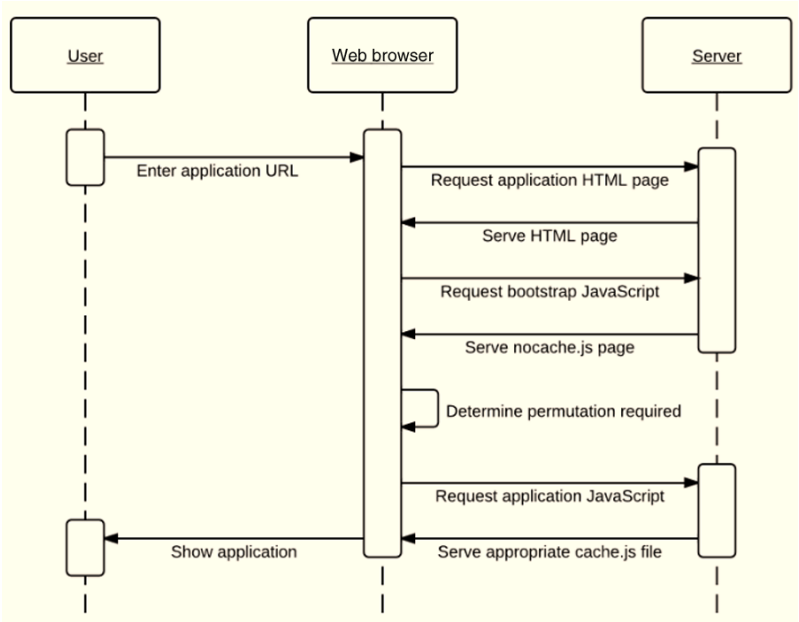

Figure 2.2 A sequence diagram showing a user request to start a GWT application

This is the typical case. You could alter the bootstrapping slightly by applying different linkers (which we'll briefly consider in section 2.8.1), but we'll assume in this book that the applications are sticking to the plain-vanilla approach described.

It's good to know this, and we'll come back to it later, particularly when we talk about deferred binding, but you're probably more interested in the developer's view at this point.

2.1.2 Examining the developer's view

A typical GWT application consists of client-side code written in Java and server-side code written in any language you like. One option for the server side is to use Java code and link to the client via GWT-RPC; other options include using GWT's RequestFactory approach, or have normal Java servlets, or PHP code, or you choose. As you'll see, GWT is remarkably flexible.

In your IDE the code of a complete application is split into two parts, as shown in figure 2.3:

- The uncompiled part of the application (your Java code).
- The deployable part of the application, created using web technologies such as Cascading Style Sheets, HTML, servlets, and when appropriate, your compiled application in JavaScript, all in a Java-compliant web archive that you can drop into a server for deployment. You can also keep any non-Java aspects of your server code in this part, such as PHP code.

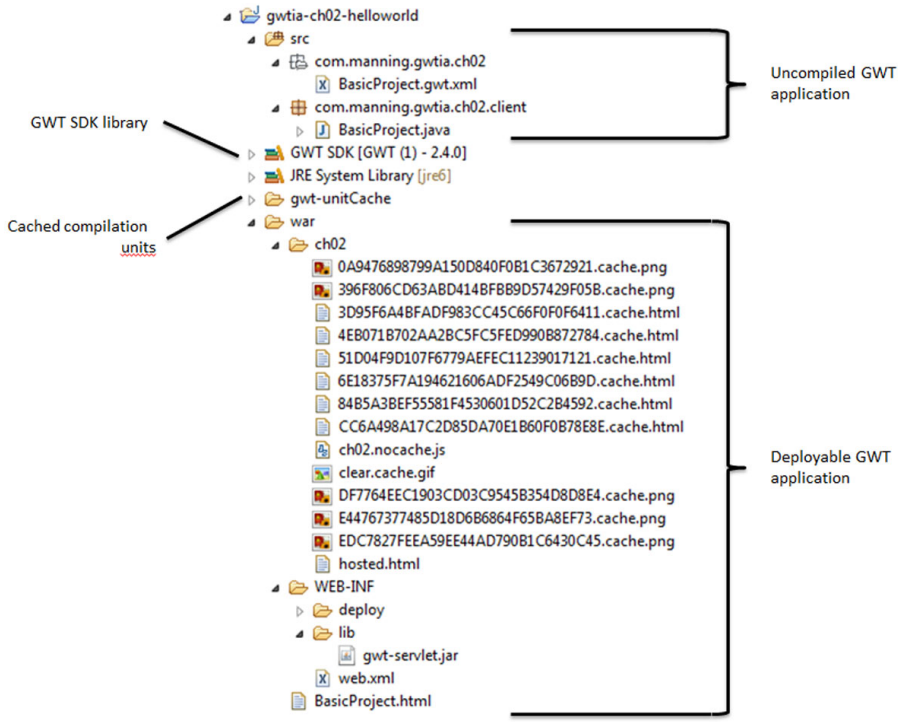

Figure 2.3 The structure of a GWT application as you'd see it in Eclipse project view. We're mainly interested in the uncompiled and deployable GWT application parts in this chapter.

The Java code is compiled to JavaScript and moved to the *deployable part* during compilation. (In figure 2.3 you can see the compiled code in the deployable part as the strangely named .cache.html files.)

Because the deployable part is a Java-compliant web archive (.war) file, it's already packaged and ready to be deployed to your favorite web container—could it be simpler?

Both of these parts are important to your application at different points in their life, and it's good that you know about that now. But we think it's better for you to get some practice under your belt before diving into the theory, so let's park this discussion for now and return to it at the end of the chapter.

If we're agreed on parking the theory, let's walk through the different modes in which you can run your application: development and web. (If we're not in agreement, you can shift to the more theoretical parts in section 2.6 and then come back here.)

2.1.3 Understanding development vs. web mode

Currently, you can run your application in two modes: development and web.

> **NOTE** Where is hosted mode? You might see references in other (older) sources to GWT's hosted mode. It has been replaced with the massively more flexible development mode.

Let's look at these in turn, starting with development mode.

DEVELOPMENT MODE

You use development mode when you're developing your code. In this mode you're running the Java code you've written—via a development mode browser plug-in—in the browser of your choice. GWT translates your Java code into the necessary JavaScript on the fly.

> **DEFINITION** *Development mode*—Running client Java code in the browser (interpreted by a development mode plug-in) and server code on a GWT-provided development server.

Development mode is quite useful. Not only can you quickly check your code in your browser of choice, but you can also run your code in debug mode in your IDE through development mode and step through it as if it was a normal Java program, as shown in figure 2.4.

Development mode starts up two servers. The first is called the *code server* and is the one that the GWT browser plug-in communicates with to enable, among other things, debugging of your application in your IDE while it's running in the browser. The other server is a typical JEE server that runs your servlets and provides your resources in development mode. You'll see later, in table 2.1, that you can alter how, and if, these servers start, giving you some flexibility in development.

It's possible that in the future this mode will become known as "classic" dev mode because there's a new toy emerging: super dev mode.

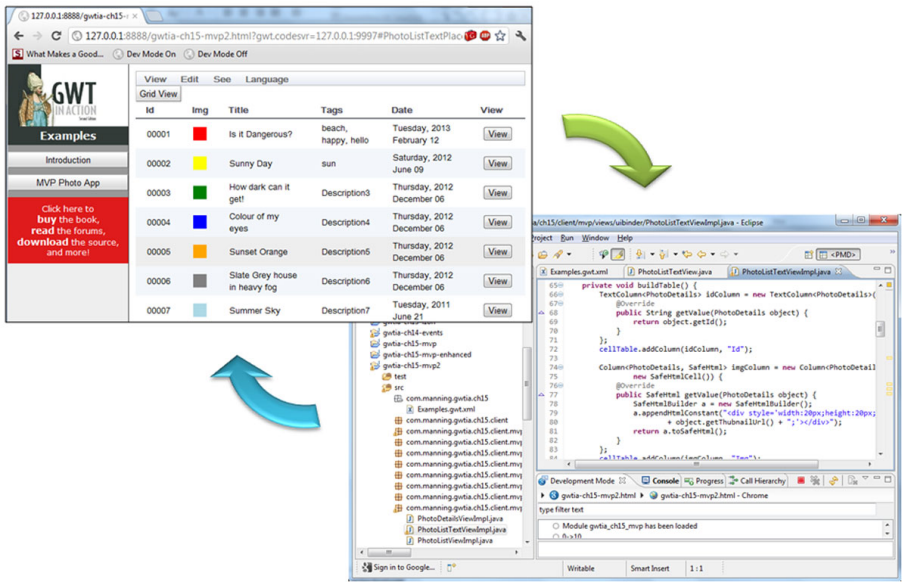

Figure 2.4 Development mode in action showing the round trip between browser, browser plug-in, Java code server, and Java code

SUPER DEV MODE

In the future you can expect to be able to use super dev mode, where you can debug your Java code running directly in the browser, doing away with the need for the current plug-in. The compiler producing source maps[1] and the browsers understanding them enables this.

GWT 2.5 starts down this path, but the functionality is limited to Chrome and Firefox, because at the time of this writing the other browsers don't support source maps, and it's highly experimental—it's subject to change and takes a lot of digging around and patience to use. You can see it running in figure 2.5, and we'll update the book's website on how to use it as it becomes more stable.

We mention this mode because it's sure to cause a buzz and we want to be able to show you where it will fit in the future (and it certainly looks interesting, as you can see from figure 2.5). This figure shows chapter 15's MVP example running in super dev mode inside of Chrome. The interesting part is in Chrome's web developer tool that we've fired up at the bottom of the page. There you can see the Java code of the example's Welcome view; if you have good eyesight, you'll see that we're at a breakpoint in the code. All this can be seen and controlled from directly within the browser.

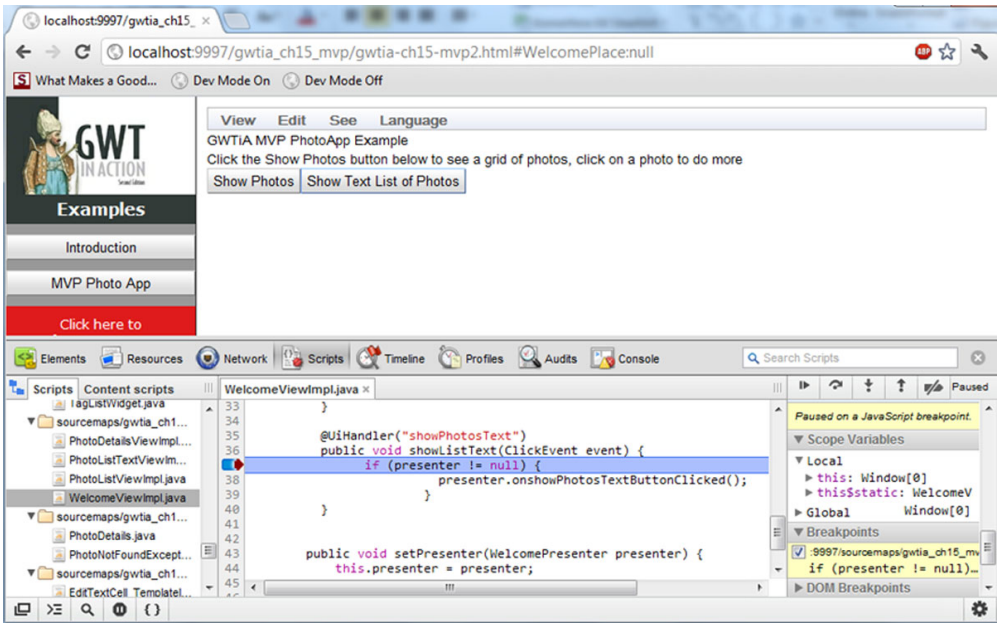

Figure 2.5 Looking ahead to the future and super dev mode. Starting in GWT 2.5 you'll be able to debug your application directly in the browser. At the moment it's experimental, but you should expect this to become stable over time, expand to all browsers, and replace the Development Mode Browser Plugin.

[1] Overview of source maps: http://mng.bz/JENs.

WEB (PRODUCTION) MODE

You're in web mode when you've compiled your application and have deployed it to a web server of your choice. In this mode, you don't need the browser plug-in or a development server, and your code is pure HTML, JavaScript, and CSS (plus any server-side technology you've chosen to use, which would include any GWT-RPC you've developed).

> **DEFINITION** *Web mode*—Running a compiled GWT application that consists of only HTML, JavaScript, and CSS, and server code on a server of your choice.

It usually takes more effort to test your application in web mode because you have to wait for the compilation process to finish before running. Then if you find an error, you have to start the whole cycle again. That's why we have development mode.

You're ready to start building an application. The first step is to decide how to do it. Remember, you have to get a structure similar to that shown in figure 2.3. Let's look at the options.

2.2 Examining the options for building an application

The structure of a GWT project is slightly more complicated than that of a normal application. It's not massively complicated, but enough subtleties exist that trying to build it by hand is likely to lead to errors and frustration.

It's therefore quite useful to have an automated way of building a simple application that you then enhance. You can create an application, automatically or manually, in several ways:

- Using Google Plugin for Eclipse (GPE) wizard
- Using GWT command-line tools
- By hand

You already know that our preference is the GPE wizard. It's easy to use and creates the simplest structure possible, which you then enhance to implement your functionality. The GPE doesn't only provide wizards for creating applications; it also has a host of supporting functions to make your development smoother, as you'll see in the rest of this book. But it requires that you use Eclipse as your IDE for the GPE approach to work.

If you aren't using Eclipse as your IDE, or you don't like the idea of using the wizards (why ever not?), you could use the GWT command-line tools. GWT comes with two command-line tools:

- `webAppCreator`—For creating web application structure
- `i18nCreator`—For creating internationalization interfaces from existing property files

The problem with the `webAppCreator` command-line tool, and GPE if you forget to stop it, is that it produces a relatively complicated default client/server application. Although that's useful to show off what you can do with GWT, it's not so useful to you when it comes to examining what is a GWT application. Nor is it useful when you want to have only a framework application to enhance, because you have to remove a lot of

code and configuration first. This gets confusing when you get errors—is it your code, or parts of configuration from that unavoidable application that you forgot to remove, or did you remove the wrong thing?

On the upside, the `webAppCreator` command-line tool creates an Ant build file with various useful targets, such as for compiling, running development mode, and running any JUnit tests all outside an IDE, although we'd go with common best practice and say that using an IDE is going to help you. The application resulting from the command-line tool can also be imported into your choice of IDE. We won't cover the command-line tools in this book, but they're amply covered in the online GWT documentation.[2]

If the GPE and the command-line tools aren't your thing, you could try to create a GWT application by hand. The issue here is that you must remember to do everything, do it correctly, and make sure all interconnections and links are there. One mistake and you'll likely spend loads of time trying to fix issues that should never have popped up, instead of developing your application.

Because we assume that you're using Eclipse as your IDE and that you agree the GPE wizards will be helpful, you're ready to start creating `HelloWorld`. We'll introduce terminology and concepts as we go, and then after you've finished creating we'll recap some of the key points and troublesome areas.

2.3 Creating the HelloWorld application with the GPE

Figure 2.6 shows the Eclipse toolbar when you have the Google Plugin for Eclipse (GPE) installed (you may have additional buttons if you've installed other items, such as GWT Designer).

To avoid some potential confusion as we move forward in this chapter over what tool we're using where, we'll use the following:

- GPE to create a new project and add necessary aspects
- Eclipse Run/Debug buttons to launch development mode

The first thing to do is to get the GPE wizards to create the web application structure.

Figure 2.6 The Eclipse toolbar, showing the buttons relating to the Google Plugin for Eclipse (left) after it's installed, and the standard Eclipse Run/Debug functions (right)

[2] GWT command-line tools information: http://mng.bz/2hoi.

2.3.1 Creating a web application

To start creating your new application, click the arrow next to the Google Plugin for Eclipse button shown in figure 2.6 to reveal the drop-down menu shown in figure 2.7.

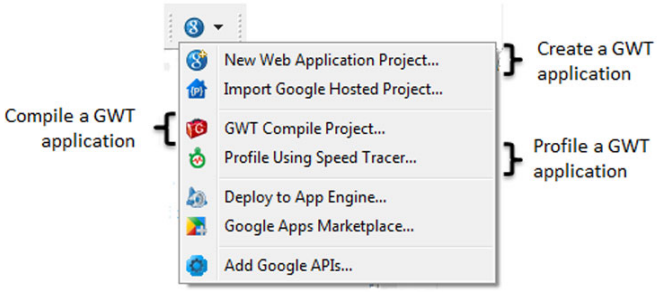

Figure 2.7 The tools available within the Google Plugin for Eclipse (GPE) drop-down menu

The drop-down menu contains a number of tools, some directly related to GWT and some indirectly. Three tools that are directly related to GWT are highlighted in figure 2.7:

- New Web Application Project
- GWT Compile Project
- Profile Using Speed Tracer

We'll concern ourselves with the first and second wizards in this chapter and leave the Profile Using Speed Tracer tool to our optimization chapter (chapter 19).

To create a new GWT application, click the New Web Application Project button, which will bring up the New Web Application Project wizard. You'll need to complete three pages of this wizard to create your application. You can see the completed first page of the wizard in figure 2.8. (Google may have made changes to the wizard since this writing, but the concepts should be the same.)

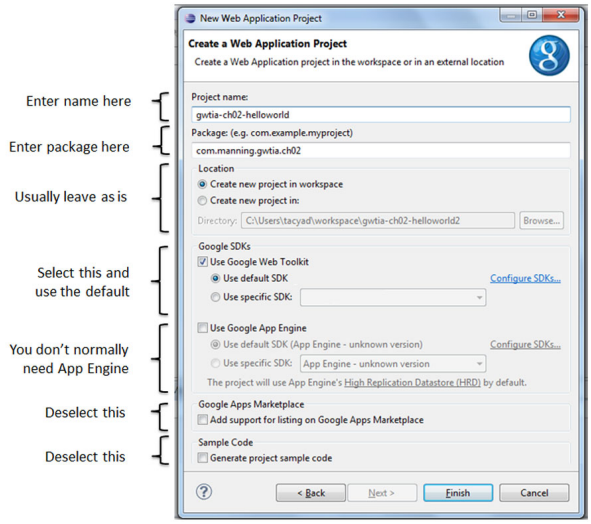

Figure 2.8 The GPE New Web Application Project Wizard—fill in the project and package name, and deselect Use Google App Engine and Generate Project Sample Code (your versions of GWT and Google App Engine are likely to be the same or later than those shown here).

All you need to do is enter a project and a root Java package name. We've chosen `gwtia-ch02-helloworld` as the project's name and the Java package name `com.manning.gwtia.ch02`. As with all Java applications, you need to store your classes within a package hierarchy. The package name can be anything you choose or restricted to the naming conventions of your development group; if you want some advice on package naming, you could check out the relevant Java tutorial.[3]

If you have several GWT SDK versions installed on your machine, now is the time to select the one you wish to work with—you'll leave it as is in this example. Also deselect Use Google App Engine because you don't want support for it in the application (the plug-in provides, among other things, support for real-time validation that code is compatible with Google App Engine and enhancement of JDO classes).

Also deselect Generate Project Sample Code because you'll create the application yourself. Leaving this selected means the wizard would produce the default client/server application code that we mentioned earlier and you'd end up tying yourself in knots trying to delete bits to get a basis to move forward. (The example is good if you want to see a whole prebuilt client/server application, and you may want to do that at some point, but for now leave the box unchecked.)

Once you click the Finish button, you've created the basic project structure, and you should see something like figure 2.9.

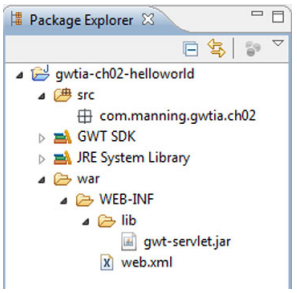

Figure 2.9 Project structure and contents after creation in Eclipse using the GPE's New Web Application Project Wizard

If you find that you now have more classes and packages in the src folder, you probably neglected to stop the wizard from creating its sample application. If this happens you should delete the project and start again.

So what do you have? The project contains two folders: an src folder, which is where your GWT Java code will sit, and the war folder, which is where the deployable application sits. At this point, the client side is the empty Java package you named in the wizard, and the deployable part contains only the gwt-servlet.jar servlet. (Don't worry about this for now. The servlet handles any GWT-RPC communication you might implement; you'll have no server side in our example, so you could delete this servlet, but leaving it causes no harm.)

To take what you have now and make it useful, you need to create the following:

- *A GWT module*—A set of instructions to the GWT compiler
- *A Java* `EntryPoint` *class*—The *main* class of your application
- *An HTML file the user will request*—Which will, among other things, request the bootstrap code for your soon-to-be compiled application

[3] Java package-naming tutorial: http://mng.bz/maoh.

You can create all three using GPE wizards by opening the File menu and selecting New and then Other. You'll get a list of relevant wizards under the Google Web Toolkit folder (see figure 2.10).

The ClientBundle and UiBinder wizards are topics for later chapters, but the middle three are of interest right now. You don't have to run these three wizards in any strict order, but we find it easier to start with the Module wizard, because it's the unit of a GWT application.

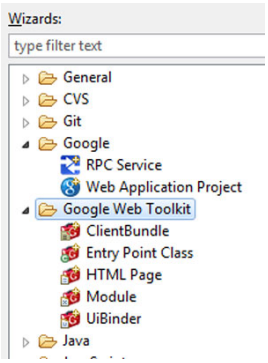

Figure 2.10 Under the Google Web Toolkit folder in Eclipse are the other GPE wizards that are available to help build GWT applications (more wizards are likely to become available over time).

2.3.2 Defining a GWT module

Highlight the `com.manning.gwtia.ch02` package in the Eclipse package/Project Explorer, and start the Module wizard through the File > New > Other menu option. You'll get a dialog box similar to that shown in figure 2.11.

Because we highlighted the package before starting the wizard, it automatically filled in the Source Folder and Package fields. If you highlighted the wrong package name before launching the widget or didn't select one, you can change it now through the Browse button.

A GWT module gives instructions to the GWT compiler. These instructions are fairly varied, but you must at least once tell the GWT compiler that you're inheriting all of the instructions in the `com.google.gwt.user.User` module. All GWT applications need to inherit this `User` module because it contains the basic GWT compiler rules for any GWT application. We'll come back to these rules later.

All you need to do in the wizard is enter the module name you want (`HelloWorld`) and click Finish. The wizard creates the module file in the package you indicated, and it's an XML file that will be called HelloWorld.gwt.xml. It looks like the following:

```
<module>
    <inherits name="com.google.gwt.user.User" />
    <source path="client"/>
</module>
```

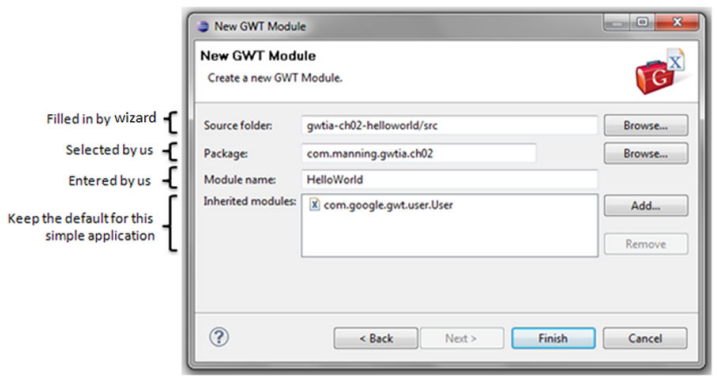

Figure 2.11 The GPE Module wizard. You should type in the module name and make sure the source folder and package are what you want.

This is a simple application with only one module; a more complicated application could have more modules. In section 2.5.2 we'll discuss this in a little more detail together with module naming and how to tell the difference and the relationship between GWT modules and Java packages.

The created module file, for now, tells GWT to inherit the `User` module and that the GWT compiler should compile code found in the `client` package and its subpackages. We'll discuss the specific reasons for needing to tell the GWT compiler what to compile (or rather, what not to compile) in section 2.6.3.

With the module in place you've told the GWT compiler how to handle your application, but you haven't provided any code for the application yet. You need to create an entry point.

2.3.3 Adding an entry point

An application's entry point is a Java class that contains the `onModuleLoad` method. You can think of this method as equivalent to the `main` method of a normal Java program and in a class that extends the `EntryPoint` interface. You need to tell the GWT compiler the name of that class through an entry in a module.

> **How many entry points?**
>
> Typically, a GWT application will have only one `EntryPoint`—you only have one way to start your application.
>
> If you define multiple `EntryPoint`s, then they *all* execute, in the order in which they're defined in the module file(s), when your application starts.
>
> Sometimes developers forget that a GWT application isn't a set of pages and they create a module/class per "page" and create multiple `EntryPoint`s, hoping only the relevant one will start—but GWT doesn't work that way.

Luckily, you can do all that in one go using the Entry Point Class wizard. It's contextually aware, so if you run it after highlighting the src folder in the project, you'll see some data filled in; if you run it when the project is selected, other data is filled in. The end point you want to get to is shown in figure 2.12.

The wizard does the following:

- It adds the necessary entry-point item to the `HelloWorld` module, telling the GWT compiler where to find the entry point. It looks like this:

```
<entry-point class="com.manning.gwtia.ch02.client.HelloWorld"/>
```

- It creates a `HelloWorld` Java class under the `client` package.

At the moment, the `onModuleLoad` method in the new `HelloWorld` class is an empty implementation; you need to fix that.

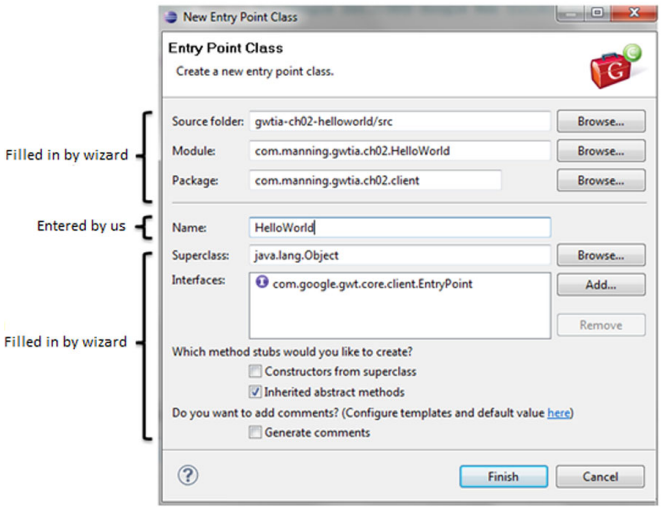

Figure 2.12 The Entry Point Class GPE wizard. Enter the class name that will be the entry point, and check that the rest of the data is what you want.

IMPLEMENTING SOME FUNCTIONALITY

As we've mentioned, the `onModuleLoad` method is the first thing executed when your application starts. It's in here that you'd normally create the application's user interface and start reacting to users' interactions. A simple application will probably do that directly in the method; more complicated applications will, like any sensible application, have functionality divided across many class files, encapsulating functionality in a well-designed manner.

Open the recently created HelloWorld.java file in Eclipse, and replace the `onModuleLoad` method with the code shown in ❷ in listing 2.1 (note that because you use the `RootPanel` and `Label` GWT user interface components in this new version of `onModuleLoad`, you need to import them into the class using the `import` statements at ❶ in listing 2.1. You can do that by either right-clicking the errors shown against the `onModuleLoad` method and selecting Fix Imports or directly typing them at the top of the `HelloWorld` class. Or you can copy all of the code in listing 2.1 into HelloWorld.java.

Listing 2.1 The entry point of the basic `HelloWorld` framework application

```
package com.manning.gwtia.ch02.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;

public class HelloWorld implements EntryPoint {

   @Override
   public void onModuleLoad(){
      Label theGreeting = new Label("Hello World!");
```

❶ **Added import statements**

❷ **Creating new widget**

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

❸ Adding widget to the page

Now the entry point will do something when you later execute the application—without wanting to jump too far ahead, we'll explain what it does. At ❷ you create a new widget—a `Label`—that holds the text "Hello World!" The `RootPanel`, in ❸, is a GWT panel that can be seen as representing the web page (or to be more exact, the body of the application's HTML). By getting the panel (`RootPanel.get()`) and adding (`add(…)`) the `Label` created in ❷, the entry point will display "Hello World!" on the application's web page when executed. (We'll talk more about panels and widgets in chapter 3.)

We've talked about the application's web page, but you still need to create the page.

2.3.4 Providing the web page

The final wizard you use for the application is the HTML Page wizard, as shown in figure 2.13. You need to fill in the filename of the web page. You don't have to follow any naming conventions here, and you'll call it a fairly obvious HelloWorld.html. You should leave Support for Browser History checked in the wizard (which results in an iFrame being inserted into the resulting HTML page) because most applications are likely to need history support. We'll cover history in the next chapter, but it's always good to include support for it from the start.

Clicking Finish means the wizard creates the HelloWorld.html file in the war directory, which should be the same as what you see in listing 2.2 (be careful to check that the wizard has created the link to the nocache.js properly, and if for some reason you can't get this wizard to work, you can manually create the file yourself with the contents of listing 2.2).

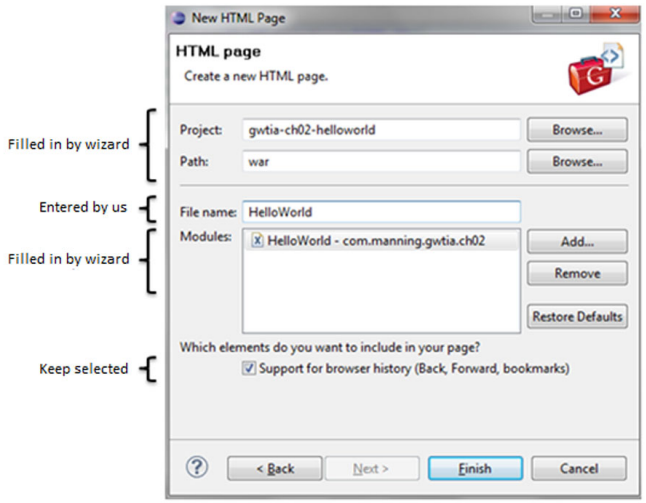

Figure 2.13 The HTML Page GPE wizard. You should enter the filename, and we recommend you keep the Support for Browser History check box selected.

Listing 2.2 Simple HTML page for the GWT application

```
<!doctype html>                                              <-- 1 Indicates standards mode
<html>
   <head>
      <meta http-equiv="content-type" content="text/html; charset=UTF-8">
      <script language="javascript"                          <-- 2 Bootstrap application
         src="com.manning.gwtia.ch02.HelloWorld/com.manning.gwtia.ch02.
          HelloWorld.nocache.js">
      </script>
   </head>
  <body>                                                     <-- 3 The RootPanel
    <iframe src="javascript:''" id="__gwt_historyFrame"
            tabIndex='-1'
            style="position:absolute;width:0;height:0;border:0">
    </iframe>                                                <-- 4 Supports history in IE
  </body>
</html>
```

BEWARE Some versions of Google Plugin for Eclipse do not create the HTML file correctly. Make sure the `<script>` tag is as shown in 2 in listing 2.2.

The web page `doctype` indicates the application will run in standards mode, which means that the browser will adhere to W3C standards 1. It also contains the bootstrap code for the application 2 and has an iFrame 4 that allows GWT to support browser Back/Forward buttons in IE (we'll look at this, known as *history*, in section 3.7). Your GWT application will see the HTML body element as the `RootPanel` that you used in the entry point to add a `Label` with some text.

Please note

You aren't restricted to HTML for the application's web page. You could use a JSP, a PHP, or whatever new technology comes out in the future. Doing so is useful if you need to create dynamic content in the page on the server side before sending it to the user.

As long as the link to the bootstrap code is included in the page, your GWT application will work (don't forget to include the iFrame as well if your application is managing history and you think your user is possibly using IE).

OK, that was a lot of information; let's take a moment to recap.

2.3.5 Recapping the magic

If you take a moment to think through what you've done in the wizards, you'll notice you created

- An HTML page for the application. Importantly, this page had four key aspects:
  - A body element that can be accessed in your Java code through a `RootPanel.get()` call
  - An iFrame that will be used to support browser history management

  - A script tag that's used to download the bootstrap code, which will download the compiled-to-JavaScript version of the GWT application and start it
  - A doctype indicating the browser should treat the application in standards mode
- A Java class file that
  - Implemented the `EntryPoint` interface
  - Implemented the `onModuleLoad` method—the compiled-to-JavaScript version that is called to start the application
  - Created a `Label` widget with some text and added it to the `RootPanel`
- A GWT module that tells the GWT compiler
  - To inherit all of the rules in the `User` module
  - To compile all the code in, and under, the `client` Java package and not other packages
  - Which class implements the `EntryPoint` interface, so it knows where to find the start of the application to insert into the bootstrapping code
- A GWT application/project that holds all of these in the appropriate structure

The most interesting thing you can do now is run the application, and you can do that in either development mode or in web mode. We'll start with development mode.

2.4 Running HelloWorld in development mode

Development mode is where you'll spend most of your time. Once you've used development mode for a while, it's easy to forget how magical it is. In this mode your application is running in a browser linked back to your Java code in your IDE by the Development Mode Browser Plugin we discussed in chapter 1. The benefit, beyond using the browser of your choice, is that you don't need to keep going through compilation cycles in order to check your application.

Even better, if you're using an IDE with a debugger, you can link your running application to it. You can add break points in Eclipse, and when you reach that part of the code while using the browser, Eclipse's debugger kicks in. Then you can inspect variables and step over or into code—all the normal things you can do in a debugger, for a web application running in a browser. That's real development power.

The downside to development mode is that it can sometimes be a little slow (though this is a minor issue, in our experience, compared to the benefits). Also, when you're working with a new version of a browser, there's often a need for a new version of the browser plug-in—those two aren't always in sync, so you have to be careful about when you update your browser (it's safest to not have automatic updates enabled). These two issues are some of the problems that the future super dev mode will remove. But, as we said earlier, as of GWT 2.5 super dev mode is still highly experimental, so we limit ourselves to talking about the usual development mode (and only mention super dev mode in passing).

Figure 2.14 shows the results of running the newly created GWT application in development mode. OK, we didn't say it was going to be the most exciting application.

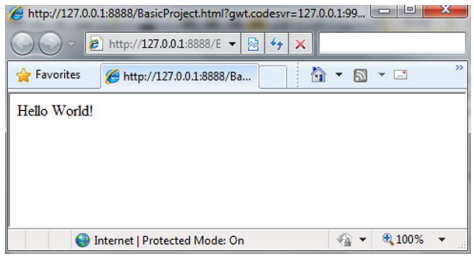

Figure 2.14 The result of running the `HelloWorld` application created using the GPE wizards

But it has enough for us to explore some key concepts of GWT before you add in some complexity.

2.4.1 Starting development mode in Eclipse

To start development mode, all you do is select the project in Eclipse's Package Explorer, click the Run button in the Eclipse toolbar, and navigate down to Run As > Web Application (see figure 2.15).

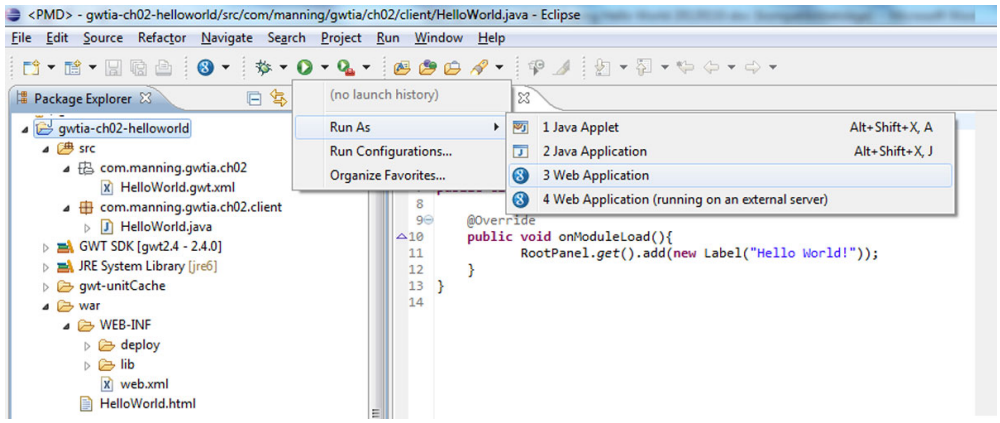

Figure 2.15 Running the project in development mode through the Eclipse Run button (highlight the project in Package Explorer, click the Play button, and choose Run As > Web Application)

If you have only one HTML page in your project, as we do, then GWT will assume that's the page you want to load when the user arrives at your application (projects with several HTML pages will trigger GWT to ask which page is the startup URL). After a while, you'll notice a new view window open inside Eclipse, with a URL in it (usually at the bottom of the screen). You can see part of this view in figure 2.16.

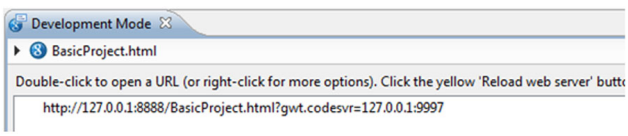

Figure 2.16 In development mode the Eclipse view window shows the project's URL. You can then enter the URL into a browser of your choice as long as it supports the Development Mode Browser Plugin.

Development mode is now running, but you won't see your application yet. To see your application, you need to fire up your favorite supported browser: IE, Firefox, Safari, or Chrome. When your browser is running, go back to Eclipse, copy the URL you see in the development mode view window, paste it into your browser, and press Enter.

If you're running development mode for the first time in the browser you chose, you may be missing the necessary Development Mode Browser Plugin. See section 1.3 for how to install it, or click the Download button shown in the browser window.

Assuming the browser plug-in is in place, it will connect back to the development mode server and your application will start. Figure 2.17 shows the development mode window in Eclipse, and that the Safari GWT plug-in has connected and the module has been loaded.

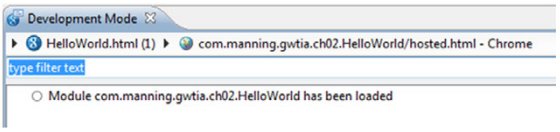

Figure 2.17 Development mode view window in Eclipse once the Safari browser has connected. You can connect more browsers, and each gets its own entry in the development mode Eclipse view window.

It might take a few moments from the time you enter the URL into the browser to when your application starts. This is normal as GWT goes through the process of preparing your application for use. Once the preparation is over, you'll see the application running in your browser. For now, it shows the "Hello World!" text, as shown in figure 2.18.

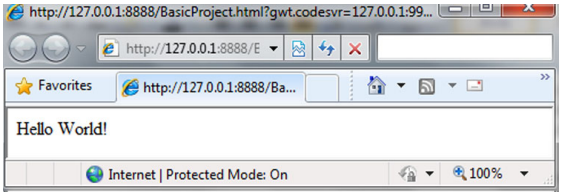

Figure 2.18 Running the `HelloWorld` application in a browser

If startup produces any errors, they'll appear in the console view window of Eclipse, though you shouldn't have any at this point.

When you started development mode at the beginning of this section, you started it using its default set of functionality. In the next section we'll show how you can change some of that by passing in some parameters.

2.4.2 Passing parameters to development mode

Starting development mode through Eclipse's Run As button fires it up using a default set of parameters. If, instead of selecting Run As, you select Run Configuration, you'll see the dialog box shown in figure 2.19, where we've already selected the Arguments tab (in some versions of Eclipse these options are at the same level in the menu, as in figure 2.15; in others the configuration choice is under the Run As menu).

In the Arguments tab you'll see two sections: Program Arguments and VM Arguments. You can use the VM Arguments section to pass any normal Java VM arguments.

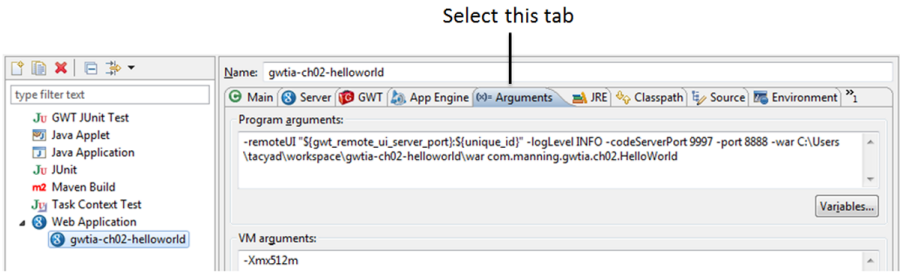

Figure 2.19 Using the Run Configuration option to set both development mode and Java VM configurations for your application

The example we have sets stack size to 512 MB as well as a couple of Mac-specific parameters.

The Program Arguments area allows you to set GWT-specific parameters. Assuming you've already run the project once, the following is what's in the standard setup:

```
-remoteUI "${gwt_remote_ui_server_port}:${unique_id}"
-startupUrl HelloWorld.html
-logLevel INFO
-codeServerPort 9997
-port 8888
-war /Users/adam/workspace/HelloWorld/war
com.manning.gwtia.ch02.HelloWorld
```

If you haven't run the project yet, then the `-startupUrl` parameter won't be there.

Our example tells development mode that it will be using a remote user interface. Next, it informs development mode what HTML page to serve when the browser plug-in connects. Development mode has logging capabilities, and the next parameter sets logging to the INFO level.

The next two parameters are associated with the development mode server. Parameter `codeServerPort` sets the port with which the browser plug-in will communicate. You see it in play in the URL you used to launch development mode, highlighted in bold as follows:

```
http://127.0.0.1:8888/HelloWorld.html?gwt.codesvr=127.0.0.1:9997
```

The `port` parameter sets the port number on which the development mode server will listen (access to any server-side GWT code you'll have would be found here). Typically you'd only change these ports if your development machine is already using them for something else. If you're running server-side code on a different server, then use the `server` parameter.

The `war` parameter is the location of the web archive (`WAR`) in which your application sits. Again, we'll cover that in more detail in the next section. Finally, you need to tell development mode the class file where the `EntryPoint` can be found.

The GPE wizards added all this information for you as you used them. If you're doing it by hand, you'll need to get this correct in order for things to work.

Table 2.1 defines the parameters you can use.

Table 2.1 Arguments that can be passed to GWT development mode

Argument	Description
`-logLevel levelVal`	The level of logging detail to provide. The `levelVal` can be one of the following: ERROR, WARN, INFO, TRACE, DEBUG, SPAM, or ALL. It works similar to Log4J in that there's a hierarchy (in the order shown previously). By setting the level to WARN, you won't see INFO or TRACE messages, for example.
`-logDir dir`	Sends the logging to the directory dir as well as to the normal screen location.
`-workDir path`	The directory that the compiler uses for storing its internal working files. It must be writable, and it defaults to the system's temporary directory. Usually, you won't have a case for changing this, unless the system's temporary directory isn't writable in your setup.
`-gen path`	The directory where the compiler will store all code created by generators. You'll use this in chapter 17 to see the code your generators are creating for debug purposes.
`-noserver`	Prevents the embedded development mode web server from starting. You might do this if you have no server-side code (note that this isn't the code server). If you're providing the server-side code on another server, then see the `-server` parameter.
`-port num`	Gives the port number for the embedded development mode server. You could change it if you need to deal with firewalls in development environments.
`-whitelist`	A comma-separated list of regular expression URLs that the user is allowed to browse to (within development mode).
`-blacklist`	A comma-separated list of regular expression URLs that the user is blocked from browsing to (within development mode).
`-bindAddress addr`	Changes the address for the code and embedded web server. It defaults to 127.0.0.1.
`-codeServerPort num`	The TCP port number, `num`, for the code server. It defaults to 9997.
`-server serv`	By providing a value here, you specify that a different embedded server is to run. The server must implement `ServletContainerLauncher`. You also need to take account of any cross-site issues you may run into.
`-startupUrl url`	Automatically launches the URL defined. This is the start page of your application.
`-war dir`	The directory, dir, to which the web archive of compilation will be sent. By default, this is the war directory in your project.
`-extra dir`	The directory into which extra files, not intended for compilation, will be placed. You'll use this in chapter 13 on internationalization, where you want to get hold of a file the compiler produces.

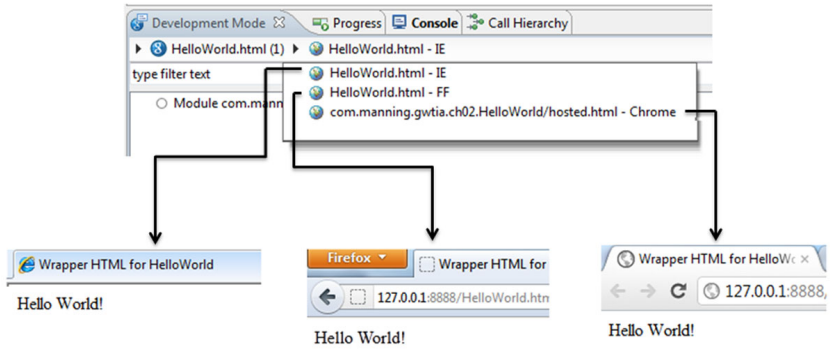

Figure 2.20 Development mode running in three different browsers at the same time, through Eclipse

In development mode you can quickly test to see if your application is doing what you want. You can set debug points in your Java code, and when the application gets there, you're taken back to your IDE to step through your Java code line by line. You can run the application in several browsers at once to quickly check that you have no browser issues; see figure 2.20. Development mode is truly powerful in cutting down your development and debugging time.

Although unlikely, it could be the case that something has gone wrong and you don't see the text on the screen. Let's see what you can do if the application isn't working as you expect.

2.5 Finding out where it went wrong

It's unlikely that `HelloWorld` has gone wrong, because it's such a simple application, but once you step beyond that simple world, things can happen. We think it's useful to check out briefly where you can look if something is going wrong, and the first place is usually the Java code in your IDE.

2.5.1 Checking the code in the IDE for errors

It should be quickly obvious if there's a syntactical problem with your Java code, it will be flagged in the IDE, usually with a red marking. In Eclipse you get a red X next to the lines in error, and in the explorer view a red X appears next to classes and packages where those errors are, as you can see in figure 2.21.

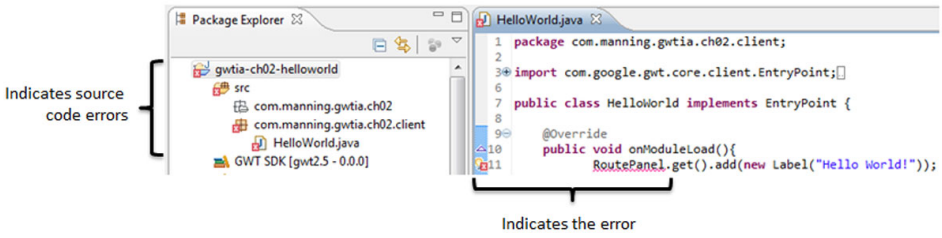

Figure 2.21 Java syntax errors highlighted by the IDE, waiting for us to fix them

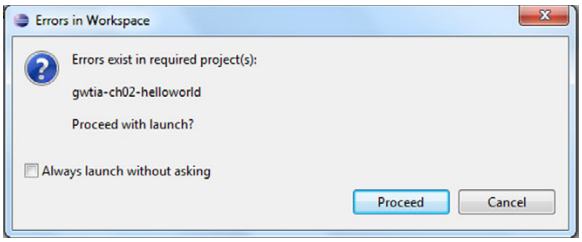

Figure 2.22 You'll see this error if you try to launch development mode when you have Java syntax errors in your application. It's not usually sensible to proceed until they're fixed.

Additionally, if you try to run dev mode without fixing these types of errors or noticing that you have them, you'll get the dialog box shown in figure 2.22.

In both these cases it's a matter of going back to the source code and fixing the underlying Java error, which is most likely a spelling issue, or you've forgotten to import a class that you're using, or there's a missing semicolon at the end of a line, and so on.

If you manage to launch dev mode without a problem, then the next place to look for errors is in the output of dev mode.

2.5.2 Looking at development mode output

The next place to look for issues is in the dev mode output. If you're using Eclipse with the Google plug-in, then you'll see that at the bottom of the IDE, as shown in figure 2.23.

The error you can see in figure 2.23 is because we forgot to include the `User` module in the application's module file (or rather, in this case, we deleted it in order to get the error). You might see in figure 2.23 that dev mode is complaining about the `Core` module being missing. We could fix this by adding it, but then we'll get a message saying that another module is missing and then another. The solution is to inherit the `User` module in our application because that inherits several other modules necessary for a GWT application to run smoothly.

Assuming there are no errors in dev mode output, you could check the console output to see if anything has gone wrong there.

2.5.3 Reading the console output

GWT dev mode runs its own web server, unless you've told it not to by using the `-noserver` flag or told it another system is running the server using the `-server` flag

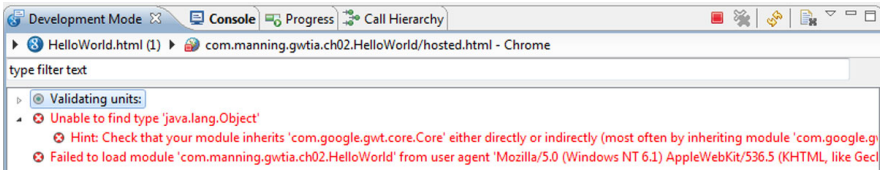

Figure 2.23 Development mode errors are usually shown on the development mode output, which is in the Eclipse window if you're using Eclipse with the GPE, or it could be in your standard output if you launched development mode from the console.

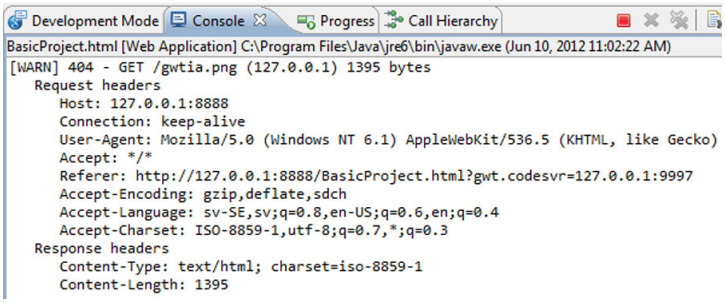

Figure 2.24 Missing resource issue, shown on the console output as a 404 warning in the `GET` call

(remember from table 2.1?). Output from the built-in web server goes to the standard console. This is different from the development mode output, but in the Eclipse interface it usually sits right next door.

Typical errors you'll see here relate to missing resources. For example, let's say we expect to show an image called gwtia.png, and it's missing; then we'd see output similar to figure 2.24. To resolve these types of errors you need to make sure resources are in the expected place and named correctly.

But it's not only syntax and missing resources that can mess up your application. Sometimes you have the logic incorrect or have misunderstood how the DOM manipulation is happening. In that situation, we need to dig a little deeper into debugging.

2.5.4 Debugging in Eclipse

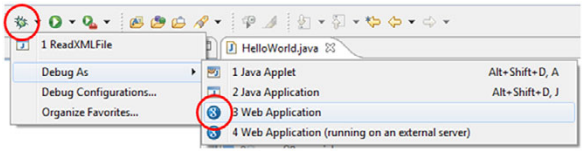

Figure 2.25 Firing up a GWT application in Eclipse to debug it

The great benefit of dev mode is the ability to debug your Java code while it's running as interpreted JavaScript in the browser. It's simple to do in Eclipse. Instead of using the Run As menu that you saw in figure 2.15, you'd use the Debug As menu item located right next to it, as shown in figure 2.25.

You launch the application in exactly the same way as running it; that is, a URL appears in the launch window, you open that, and your application starts the same as before. Back in Eclipse you can set breakpoints on lines of code where you want to dig deeper into what's happening. You do that by double-clicking next to the line of code you want to stop at, and a breakpoint symbol will appear (see figure 2.26).

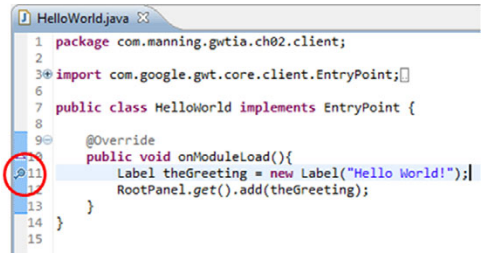

Figure 2.26 A breakpoint inserted into the `HelloWorld` application. This will trigger the debugger to kick in if you run dev mode through the Debug As menu option rather than the Run As option.

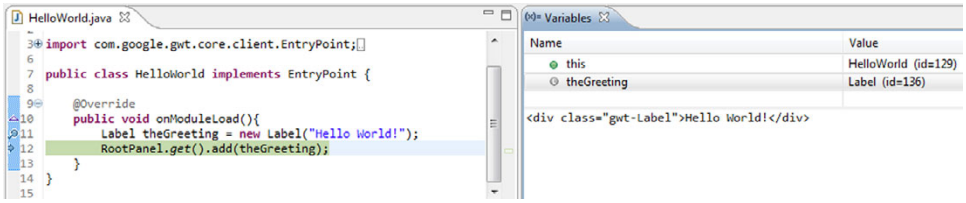

Figure 2.27 Debugging in action. We've stepped over line 11 and the variable `theGreeting` has been created. We can inspect it and see that it's a `div` element with our text in it.

With a breakpoint set, if you refresh the application in the web browser, the IDE debugger kicks in, allowing you to step into code execution and watch variable values and so on. It's a great way to check that things are happening in your code and in the order you expect.

Looking at figure 2.27 you can see that the breakpoint was hit at line 11; we ran that line, which created the `theGreeting` variable.

In the variable inspection part of the debugger, you can select that variable and see what value it has. You can see in figure 2.27 that it has the value `<div class="gwt-Label">Hello World!</div>`. This is a hint that GWT Java widgets are DOM elements (we'll come back to this more in chapter 4).

You can couple debugging in your IDE to using the browser's development tools, and you'll get visibility as to what's happening.

2.5.5 *Inspecting using browser development/inspection tools*

Some browsers come with a comprehensive set of development tools built in or that you can add in. Chrome has a great built-in set (so does Safari, but it doesn't have a development mode plug-in, so that's more applicable to debugging in web mode). For Firefox there's Firebug (http://getfirebug.com/), and IE has, for example, the F12 Developer Tools[4] (or Firebug Lite).

If you fire up your favorite inspector, you can see what has happened in the DOM, check that the right scripts have been downloaded, and so on.

Figure 2.28 shows the Chrome development web tools (accessed through the wrench at the top right of Chrome) in action. We've selected the Elements tab in the tool, and you can see the whole active web page on the left side.

Inspecting the DOM elements, you can see our label inserted (the `div` element you saw in figure 2.27). Highlighting the label in the tool shows on the right side what styles it has.

Coupling the Java debugger with a web development/inspection tool is a powerful way to debug those awkward issues you can't find otherwise. You get to step through your code line by line and see the impact on the DOM in real time.

---

[4] IE F12 Developer Tools: http://mng.bz/6Fe4.

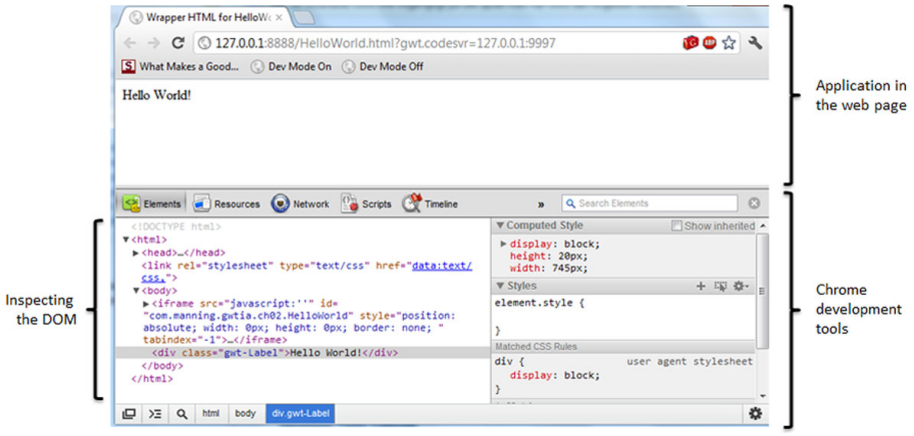

Figure 2.28 The Chrome development tools in action on the simple `HelloWorld` application

Once you're happy that your application is working as you want it—usually when you're ready to deploy it for real web testing and production—it's time to turn the client side from Java into JavaScript. With the GPE it's as simple as clicking a button.

2.6 Compiling HelloWorld for web mode

In this short section, we'll look at how to compile an application for real-world use: that is, make it into a complete package that can be deployed to a web server for users to access. We'll do that in Eclipse as well as look at what parameters can be passed to the GWT compiler.

> **TIP** Want to get efficient JavaScript? From GWT 2.5 you can get GWT to use Google's Closure[5] Compiler to extract as much efficiency as possible in the compiled JavaScript of your application. Pass the `-XenableClosureCompiler` flag to the compiler.

Ready to compile? Then let's do it.

2.6.1 Running the GWT compiler from Eclipse

To run the GWT compiler from Eclipse you select the GWT Compile Project menu option from the Google Plugin for Eclipse's drop-down menu (as shown in figure 2.29).

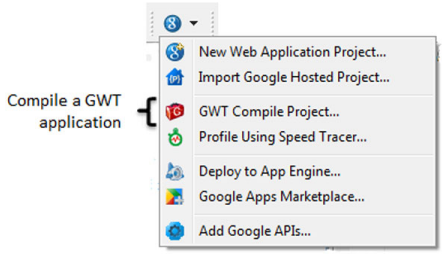

Figure 2.29 Starting the compile process for a GWT project in Eclipse using the GPE

[5] Closure compiler: https://developers.google.com/closure/compiler/.

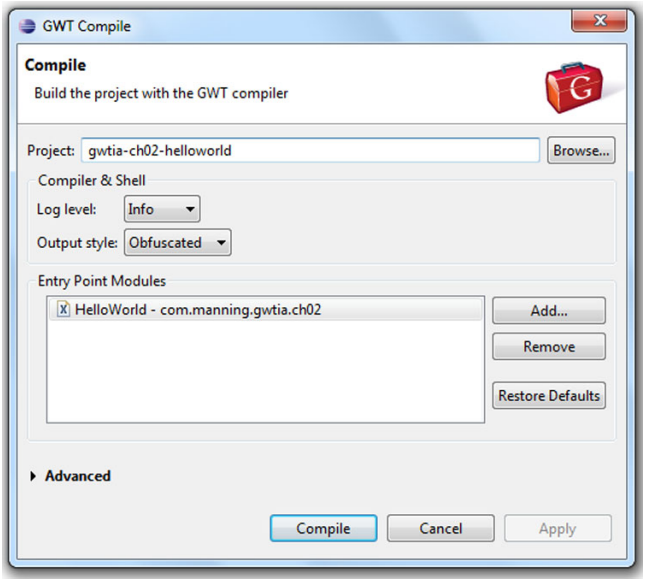

Figure 2.30 Compiling `HelloWorld` using Google Plugin for Eclipse. Check that the information is correct, and click the Compile button to start compilation.

This opens the dialog box shown in figure 2.30.

The GWT Compile dialog shows the project it will compile, the level of logging it will output, and the output style. By default, the GWT compiler will output JavaScript code in Obfuscated style; following is a quick sample from compiling this chapter's project as Obfuscated:

```
function Pb(a,b){a.length>=b&&a.splice(0,b);return a}
function hi(a,b){var c;c=di(a,b);if(c==-1){throw new Yl}gi(a,c)}
function Tk(a){if(a.b>=a.c.c){throw new Yl}return ml(a.c,a.b++)}
function ni(a){if(a.b>=a.c.c){throw new Yl}return a.c.b[++a.b]}
```

This is next to impossible for humans to read and get a sense of what's happening, but it's highly compacted into the smallest size possible. That's good, because the smaller the download size, the quicker your application will start, and 99.9999% of the time you should have no interest in reading the outputted JavaScript. But it's not so great that 0.0001% of the time when all your other error-resolution and debugging techniques still haven't found the cause of an issue and you want to read the compiler output. If you change the output style to Pretty, then the output code is much more readable:

```
function gwtOnLoad(errFn, modName, modBase, softPermutationId){
  $moduleName = modName;
  $moduleBase = modBase;
  if (errFn)
    try {
      $entry(init)();
    }
     catch (e) {
```

```
      errFn(modName);
    }
   else {
    $entry(init)();
  }
}
```

Setting the output style to Detailed provides even more information in the JavaScript file, such as fully qualified function names that include the package and class they came from. We'll leave that to you to experiment with if you want.

You can also add/remove entry points to/from your application and indicate which one should be used. For now, leave this as it is, because the example application has only one entry point.

If you click the Compile button, then your client-side code will be compiled into the war folder inside the HelloWorld folder, along with any Java server-side classes being moved to the WEB-INF classes folder (you don't have any in this example). Assuming you haven't found any compilation errors, your complete application is now almost ready for deployment to your test or production servers and should look similar to figure 2.31.

We say "almost ready" because there's one final step you might like to do: tell the web server what file to serve up as the welcome page to the application.

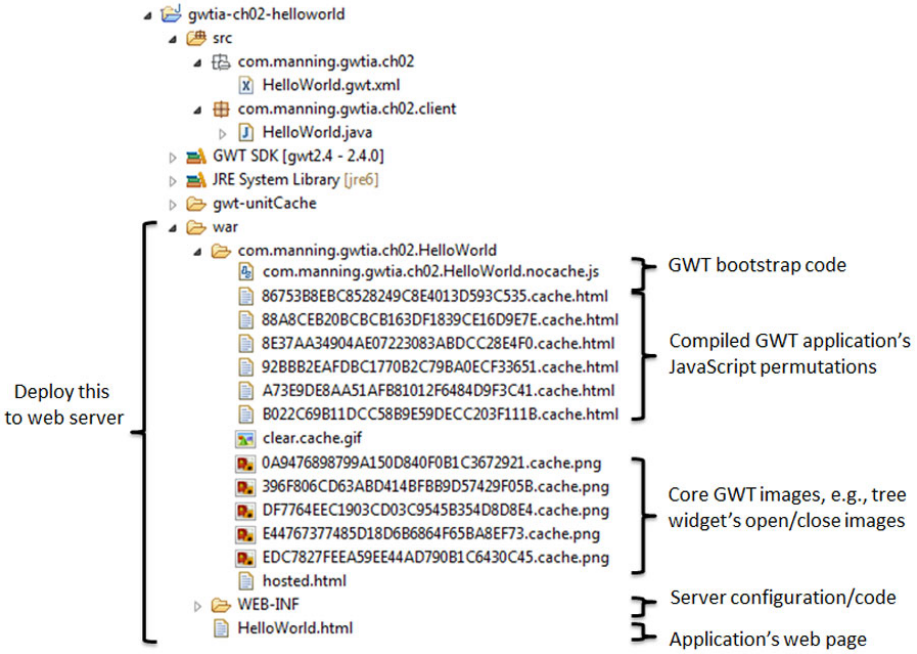

Figure 2.31 The result of compiling this chapter's example application into JavaScript held within a web archive format that's almost ready for deployment to your server (you might optionally want to indicate a welcome page for the application before deploying)

2.6.2 Welcoming the user

Because you're dealing with a web archive here, you can tell your server which page to serve up as the default. You need to add an entry into the deployment descriptor[6] (web.xml) file stored under the WEB-INF folder. The GPE has already created that file for you, and the following listing shows it together with an entry for the welcome.

Listing 2.3 Default web.xml file with an entry added for your welcome page

```
<?xml version="1.0" encoding="utf-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
   <!-- TODO: Add <servlet> tags for each servlet here. -->
   <!-- TODO: Add <servlet-mapping> tags for each <servlet> here. -->
   <welcome-file-list>
      <welcome-file>HelloWorld.html</welcome-file>
   </welcome-file-list>
</web-app>
```

Servlet definitions (1)

(2) Welcome page

You'll see in the chapters on server-side code what you'd put into listing 2.3 at (1) if you had server code, but because you don't, you can either leave in the `TODO` comments or delete them. (2) is the interesting part for us. You've added the application's HTML page to the descriptor, so that the user can write the prettier http://www.myapp.com rather than http://www.myapp.com/HelloWorld.html. Because you're still most likely running in dev mode, adding (1) to web.xml means you can write http://127.0.0.1:8888/?gwt.codesvr=127.0.0.1:9997 in the browser to start the application (note that you still need the ? before the `url` parameter) instead of the longer http://127.0.0.1:8888/HelloWorld.html?gwt.codesvr=127.0.0.1:9997.

How you deploy the WAR to the server is dependent on which server you're using, so we won't cover that.

As you could for development mode, you can pass a number of parameters pass to the compiler.

2.6.3 Passing parameters to the GWT compiler

Table 2.2 shows some of the parameters that can be passed to the compiler (the others we defined in table 2.1: `-logLevel`, `-workDir`, `-gen`, `-war`, `-extra`). You pass most of these parameters through the Advanced section of the dialog shown in figure 2.30 by typing them in with any associated value. Two parameters can be set directly in the dialog box itself: the log level and the output style.

---

[6] web.xml: http://en.wikipedia.org/wiki/Web.xml.

Table 2.2 Arguments that can be passed to the GWT compiler

Argument	Description
`-style styleVal`	The style that the created JavaScript will have. The `styleVal` can be one of OBF[USCATED], PRETTY, or DETAILED. OBF is used for production because it produces the smallest code; but it's not easily readable, so for debugging you may prefer PRETTY, or to get the compiler to show all information in the output, try DETAILED.
`-ea`	Makes the compiled code include asserts from the source.
`-validateOnly`	Validates all source code but doesn't compile it.
`-draftCompile`	Compiles the code in a faster but not so optimized manner.
`-compileReport`	Use this argument, and the compiler will produce a Story of Your Compile report. This is useful to understand what decisions the compiler made; you'll see this again in chapter 18.
`-localWorkers num`	The number of local worker threads to use in compilation.
`-soyc`	Enables the production of the Story of Your Code report (see chapter 19 on optimization).
`-strict`	Succeeds with compilation only if all input files have no errors.

Let's also review a number of other options that are somewhat experimental at the time of writing. Table 2.3 lists them.

Table 2.3 Experimental arguments that can be passed to the GWT compiler

Argument	Description
`-XdisableAggressiveOptimization`	Disables the aggressive optimization the compiler performs.
`-XdisableClassMetadata`	Disables some `java.lang.Class` methods. This can reduce compiled file sizes.
`-XdisableCastChecking`	Disables runtime checking of cast operations. This can reduce compiled file sizes.
`-XenableClosureCompiler`	From GWT 2.5 onward, there's the option Use the Closure Compiler on GWT Output—this should create smaller JavaScript output.
`-XfragmentMerge`	Enables the GWT 2.5 fragment merge code splitter (as opposed to the pre-GWT 2.5 standard one—see the optimization chapter for details).
`-XdisableRunAsync`	Disables the ability to code split.
`-XdisableSoycHtml`	Disables the production of HTML from the Story of Your Code report, leaving only the XML output.

Table 2.3 Experimental arguments that can be passed to the GWT compiler *(continued)*

Argument	Description
`-XdisableUpdateCheck`	Disables the update check for a new version of GWT at compilation time.
`-XsoycDetailed`	Enables extra detailed information to be produced for the Story of Your Code report.
`-XcompilerMetrics`	From GWT 2.5, compiler metrics aren't emitted by default for the Compile report; this flag turns them on.

That's it; you now have a basic application that you can use as the basis for any application. In the next chapter we'll present the results of doing that in the form of the `BasicProject` application, which gives the functionality of figure 2.1.

You could take the output you now have and play with different widgets and panels (see the `com.google.gwt.user.client.ui` package of GWT for different types if you want to get hacking, or look through chapter 4 for some more guidance).

Looking to generate source maps?

GWT doesn't have a compiler flag to generate source maps yet. Instead, you need to add the following to a GWT module that your application uses:

```
<set-property name="compiler.useSourceMaps" value="true">
```

You may have noticed that we introduced a lot of terminology when we went through the wizards, such as `EntryPoint` and module. These might be new to you, or you might be wondering what the difference is between concepts that seem close, such as Java packages and GWT modules. Let's spend a few minutes trying to clarify these areas of potential confusion, starting with perhaps the most easily confused: GWT modules and Java packages.

2.7 Understanding modules vs. packages

Every GWT application/project will contain Java packages and GWT modules. You're probably fairly familiar with Java packages; they're a way of encapsulating related Java classes and/or functionality together under a single namespace.

> **DEFINITION** *Java package*—Encapsulates related Java classes and functionality under a single namespace. They're useful, among other reasons, for the separation of logical units of code.

The application you'll create shortly will have a package called `com.manning.gwtia.ch02.client`, with a single Java class file.

GWT overlays a parallel module structure in the source code, which is invisible and inconsequential to the Java compiler but is visible and important to the GWT compiler.

> **DEFINITION** *GWT module*—Encapsulates units of GWT configurations (paths, properties, deferred binding, and so on) on the source code of your application. They're defined in an XML module file and stored in the Java package hierarchy. They're useful, among other reasons, for the separation of logical chunks of functional instructions to the GWT compiler.

A module is defined by an XML format file that gives instructions to the GWT compiler. These instructions tell the compiler, among other things, where to find source Java files to compile into JavaScript, where the application's entry point is, and how to handle generators and deferred bindings (these are techniques GWT relies on to minimize code you need to write and handle such things as browser differences; we discuss them in detail in chapters 16 and 17), and they can define a number of properties (such as what locales are used in internationalization; see chapter 11) or what other modules need to be inherited.

Modules are defined in a *.gwt.xml files, which we'll call module files. Let's take a moment to look a little more at a module file.

2.7.1 What's in a GWT module?

Listing 2.4 shows a relatively complicated example of a module file (more complicated than the simple one in `HelloWorld`).

Listing 2.4 A more complicated GWT module file

```
<module>
  <inherits name='com.google.gwt.user.User'/>
  <inherits name='com.google.gwt.i18n.I18N'/>
  <inherits name='com.google.gwt.user.theme.standard.Standard' />
  <inherits name="com.google.gwt.logging.Logging" />
  <inherits name="com.google.web.bindery.requestfactory.RequestFactory" />

  <entry-point
         class='com.manning.gwtia.ch02.client.HelloWorld'/>
  <source path='client'/>

  <define-property name="dev.mode" values="debug,prod"/>
  <property-provider name="dev.mode"
                     generator="com.gwtia.ch17.DMPropProvGenerator"/>


  <extend-property name='locale' values='en_US, en_GB, fi, is, sv, ar'/>
  <set-property-fallback name='locale' value='en_US'/>
  <set-property name="gwt.logging.logLevel" value="ALL" />
  <set-property name="gwt.logging.enabled" value="FALSE"/>
```

```
  <extend-configuration-property
        name="compiler.splitpoint.initial.sequence"
        value="com.manning.gwtia.ch19.client.car.CarGateway" />
  <set-configuration-property name="CssResource.style"
                              value="stable-notype"/>

  <replace-with class="com.manning.gwtia.ch17.client.devmode.Type_Debug">
    <when-type-is class="com.manning.gwtia.ch17.client.devmode.Type"/>
      <when-property-is name="dev.mode" value="debug"/>
  </replace-with>

  <generate-with
         class="com.manning.gwtia.ch18.rebind.WidgetDebugGenerator">
    <all>
      <when-type-assignable class="com.google.gwt.user.client.ui.Widget"/>
        <none>
          <when-type-assignable
                       class="com.google.gwt.uibinder.client.UiBinder"/>
          <when-type-assignable
                    class="com.google.gwt.logging.client.LoggingPopup"/>

        </none>
    </all>
  </generate-with>
</module>
```

Listing 2.4 contains a number of `inherit` instructions telling the compiler to find, and use, the rules in those inherited modules in another module in the first line. Inheritance in this case means that the GWT compiler will apply all the instructions it finds in the inherited module as well as those written in this module. For some entries GWT treats inherited rules as additions; you can have multiple inherits and more inherits in the inherited module. But other entries, particularly when setting properties, are hierarchical; a later entry will replace an earlier one. We'll note these in the relevant chapters.

The rest of this module file plays around with a number of properties and configuration properties, extending and setting them. These will all drive different behaviors in how the compiler operates. For example, the extending of the `locale` property tells the GWT compiler it needs to create a number of outputs for different locales (UK English, American English, Swedish, and so on).

Instructions such as `replace-with` and `generate-with` are powerful. The first tells the compiler to replace a certain type with another type when certain conditions are met; this is how GWT handles differences with browser implementations (more on this in chapter 17). Giving the GWT compiler a `generate-with` tag tells it that when it sees a particular type, it needs to automatically start creating code based on a generator class. Think of that—the GWT compiler can automatically create parts of your program for you.

We'll introduce the items that can be found in a module file when we need them as we go through the book.

The module file is always named after the related module. When you run the GWT tools in a moment, you'll create the `HelloWorld` module, and that will be defined in the HelloWorld.gwt.xml module file. It's stored within the Java package structure, and although a module can be placed anywhere, it's strongly recommended to place it in the root package, where it's most applicable.

For a simple application with only one module, that means the module file is found in the root Java package (`com.manning.gwtia.ch02` in our example). Other, more complicated applications may have more than one module. But before we discuss how many you should have, we'll have a little discussion on the benefits of a module.

2.7.2 What are the benefits of modules?

The benefit of a GWT module is that it allows you to group together necessary instructions for the GWT compiler at the lowest level possible in your code tree, enabling reuse.

Say you have a math library that you'll use in your application, so you add it to your project. At first glance there's little benefit in creating a GWT module for it, because there are unlikely to be any GWT compiler-specific instructions for it.

But maybe this is a legacy math library that's shared with other applications. Because of that, its namespace is `com.myco.math` rather than `com.myco.client.math`. You'll see later that the GWT compiler works, by default, on code under `client`, so your math library won't currently be visible to the GWT compiler. That can be fixed by putting a `<source path="math"/>` instruction in a GWT module.

It could also be the case that the library needs to implement a math function a different way for IE6 than for other browsers. You'll see later in chapter 17 that you can use deferred binding to manage this by using a `<replace-with>` instruction in a module file.

You could put both of these instructions into the application's main module file, but that would hinder you if you wanted to use the math library in another application (because you'd have to remember to copy both instructions into that new application's module file).

The simpler solution is to put a GWT module file in `com.myco.math`, say Math.gwt.xml, which contains only those two GWT compiler instructions necessary to use the math library. You'd then inherit that module in your existing application's module, via an `<inherits name="com.myco.math.Math"/>` instruction, to make it all visible.

> **TIP** Sharing GWT code? You must remember to include the source code in any .jar file you create because the GWT compiler works from Java source code.

When it comes to sharing or reusing this module, you export the source files from the `com.myco.math` package, together with subpackages and the module file, as a .jar file. Then you make sure the .jar file is on the new application's class path, to make the Java compiler happy, and inherit the `Math` module in the new application's module file, to make the GWT compiler happy.

> **Inheriting a module?**
>
> When sharing GWT code across applications, you need to remember that in addition to putting the new code on your classpath, you must inherit the appropriate module (through an `<inherit>` tag) in your new application's module file.
>
> Failure to do so will result in GWT throwing a "did you forget to inherit a module?" error.

The act of inheriting the module makes the instructions visible to the GWT compiler of the new application. Unfortunately, inheriting isn't automatic. The author of the module needs to tell others what module to inherit for things to work.

If you don't inherit the correct module, the GWT compiler will throw a "did you forget to inherit a module?" error. You can even get this error in standard GWT if you use functionality outside the standard aspects, for example, client bundles. An efficient way to manage resources, which is discussed in chapter 5, requires you to explicitly inherit the `Resources` module (we'll cover these as and when needed).

Now that we've looked at the benefits, you probably have the question "How many modules should I have?" in your mind.

2.7.3 How many modules should you have?

You could have one module for your whole application, or at the other end of the scale, you *could* have a module per Java class. In practice, the number of modules you'll have will be a design decision based on how decoupled you wish to make logical GWT units of the code and how much reuse you expect in the future.

Our chapter examples are, we like to think, a good example. Let's take chapter 11's example about three different ways of supporting languages in your user interface. Like

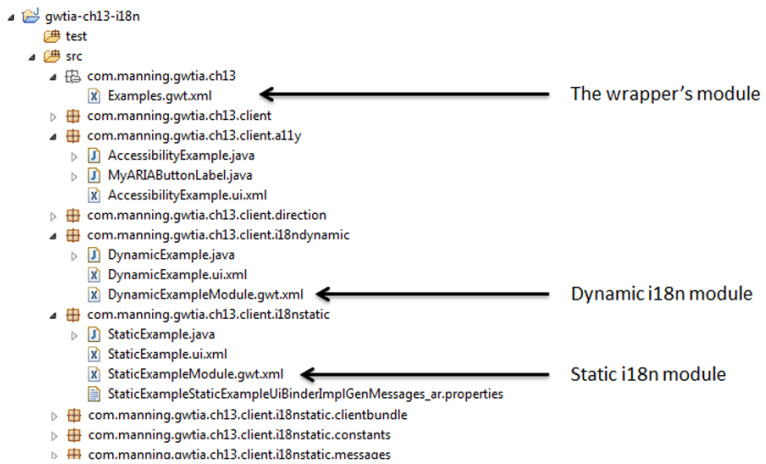

Figure 2.32 Chapter 11's module setup: one module for the wrapper application and one module for each of the distinct examples. This allows a nice separation of concerns, which means each example's module file shows only those module instructions necessary for it to work.

most of the chapter examples, it consists of a wrapper application that allows the user to select one of a number of examples. The wrapper is treated as one module and each example as another; see figure 2.32.

The benefit in such a setup is that each module describes only what it needs and no more. You can therefore quickly see what inherits and properties, for example, you need for a particular functionality.

The wrapper's module file, Examples.gwt.xml, is shown in the next listing.

Listing 2.5 An example of a module inheriting other modules

```
<module rename-to="gwtia_ch13_i18n">
   <inherits name='com.google.gwt.user.User' />
   <inherits name='com.google.gwt.user.theme.standard.Standard' />
   <inherits name="com.google.gwt.resources.Resources" />
   <inherits name=                                                    (1)
     "com.manning.gwtia.ch13.client.i18nstatic.StaticExampleModule" />
   <inherits name=
     "com.manning.gwtia.ch13.client.i18ndynamic.DynamicExampleModule" /> (2)
   <entry-point class='com.manning.gwtia.ch13.client.Examples' />
   <source path='client' />
</module>
```

(1) Inheriting the static example's module

(2) Inheriting the dynamic example's module

In listing 2.5 you can see that inheriting is merely a case of using the `<inherits>` tag and providing the fully qualified name of the module (1) and (2) (package structure, for example, `com.manning.gwtia.ch13.client.i18nstatic`, plus module name, for example, `StaticExampleModule`).

The module holds the entry point definition, so GWT knows where to find the "main" function, and it inherits GWT's browser history management functionality. It also inherits the three example module files; otherwise they'd be orphaned (meaning the GWT compiler may miss important information, and compilation will fail).

Each of the individual example modules contains only the GWT instructions it needs. For example, the StaticExampleModule.gwt.xml module file contains only the necessary instructions to tell the GWT compiler that we'll be using static internationalization. This means inheriting the `I18N` module, extending the `locale` property (defined in the `I18N` module) with the locales we're using, and setting a fallback locale.

As an interesting exercise, you can investigate the module structure GWT employs by looking at the `User` module defined in `com.google.gwt.user`, which is inherited by all GWT applications, and following all the inherits that are in there.

In this book we'll introduce the various instructions the module file can contain as and when they're needed (for example, in the chapters on internationalization, generators, deferred binding, and so on).

We hope modules now make a little more sense, and we hope that as you look at the example code in this book they become even clearer. With the knowledge of modules under your belt, we should dig into the *uncompiled* part of application a little more to see what structures and restrictions we need to consider.

2.8 Digging deeper into the uncompiled application

In the uncompiled part of the project sits the Java code for your application and any GWT module definitions, both client and server side (if you're using GWT-RPC or RequestFactory). You should stick to certain conventions for folder structure as well as Java package structure. Also noteworthy are some (sensible when you think about it) restrictions on the Java constructs you can use on the client side, as well as some Java classes GWT expects to see, as we'll cover next.

2.8.1 Folder structure convention

Typically, a GWT application structure has four folders: src, lib, bin, and test. The content of your test cases is usually found in the test folder. Because you aren't including any other Java/GWT libraries in the example here, you won't have a lib directory, but if you were, then you'd probably put them in here.

The src directory is where you'll put all of your Java source code, and sometimes Eclipse or another IDE will show the bin folder. The bin folder isn't relevant to GWT, but IDEs typically create it and store IDE-compiled versions of your Java source code there (the normal class files that aren't used by GWT).

Within the src folder, you'll find your project's Java package structure, along with its GWT modules.

2.8.2 Package structure convention

You're free to use any Java package structure you want, but there are some conventions GWT expects:

- Code you want the GWT compiler to compile to JavaScript should go under the `client` package.
- GWT RPC code for the server (see chapter 7) should go under the `server` package.
- Code that's shared between server and client should sit in a `shared` package.
- Code used for generators (see chapter 17) typically sits in a `rebind` package.

These are all "should" conventions, and GWT offers the flexibility to use different packages if you want through definitions in module files.

If you want GWT to compile code in a package other than the default `client` package, say `some_other_package`, use a `<sourcepath="some_other_package">` entry in one of your module files.

> **Have code to compile outside the client package?**
>
> Sometimes you may wish to store your code in a package other than `client`. That's possible with GWT by providing a `<source>` directive in your module file. For example,
>
> ```
> <source path="shared"/>
> ```

(continued)

would tell the GWT compiler that the package `com.manning.gwtia.ch02.shared` and its subpackages also contain code it should compile into JavaScript.

We'll come to this point here a few times with GWT. Because something is visible to the Java compiler (that is, you've imported it using an `import` statement) doesn't mean it's visible to the GWT compiler. The `source` directive may be required.

Similarly, if you're using code in another module, make sure it's inherited in your module definition; otherwise you'll get a "did you forget to inherit a module?" error in development mode/compilation.

The src directory holds your application's Java files, but there are some restrictions on what parts of Java you can use, which we'll explore next.

2.8.3 What parts of Java can you use in GWT?

Everything that's in a source package for GWT will be compiled to JavaScript. By default that means the `client` package, but you can add other packages, as you've seen.

Any code that's compiled to JavaScript needs to be written in the subset of Java that GWT understands. This restriction is common sense if you take a moment to think it through: you can only use those parts of Java that make sense in a browser context (for example, currently most of the file aspects of `java.io` aren't available because browsers can't, pre-HTML5, access the filesystem).

It's easy to forget this restriction when you're working in a Java environment only, and you'll be reminded at compile time only if you've used something you can't.

What Java can you use on the client side?

Check out the GWT JRE emulation document, which tells you what Java packages and classes you can use in your client-side code:

http://mng.bz/V0xm

As a rule, if a browser can't perform some functionality, it won't be available in the GWT JRE.

Outside of code that's to be compiled to JavaScript, you can use any Java you want. One particularly important Java class that each application will have at least one of (and usually only one) is the `EntryPoint`, which we covered back in section 2.3.3.

Up until now, we've talked mainly about the client-side part of your application, the bit your user sees on the screen. In preparation for the section on deployed applications, let's take a sneak peek at the server side of a GWT application, which might reside in the uncompiled application part or the deployable part, depending on your choice of server code.

2.8.4 The server side

The `server` package is, by convention, where the Java code you wish to execute on the server is stored in the project. Because the package is normally not included in any `<source>` directives in module files, it will never be compiled to JavaScript, and therefore you have no restrictions on what Java constructs you use.

Java code in the `server` package of the undeployed application will be compiled into class files and moved by the GWT compiler into the classes directory in the deployable part of the project automatically—one less thing for you to worry about. In chapter 7 we'll look at introducing server-side code to applications using GWT's RPC approach, and in chapter 8 we'll look at the newer `RequestFactory`.

But if you have non-GWT-RPC/`RequestFactory` code, for example, other servlets or server-side code in another language, you can store it directly in the deployable application part of your code package, and that's what we'll look at next.

2.9 Reviewing the deployable application part of a GWT application

The web archive (war) is where the application you'll deploy to your production server resides. It's a standard Java Enterprise Edition[7] (JEE) web archive, which means your compiled GWT application is standards compliant (and if you're familiar with wars you can expect it to behave in line with your expectations).

Before you compile your application for the first time, this folder holds only the static resources of the project, for example, the application's HTML file and perhaps CSS files, images, and any non-GWT-RPC server-side code.

After compilation, the resulting JavaScript files for the client side are placed in the web archive and then any GWT-RPC server-side code is moved into the classes directory for you. You have to manually put any .jar files your server-side code relies on in the lib directory. GPE has already put the gwt-servlet.jar used by GWT-RPC code (see chapter 7) in this lib directory; you can safely delete this if you aren't using GWT-RPC.

What's with the funny filenames?

You'll have noticed in the deployable part of the application some strange filenames, such as 524C229849DD7888D1BBAB1A1C867FB5.cache.html. These are the JavaScript implementations of your application that the compiler creates and names. Within the bootstrap code of your application (also created by the compiler), the correct file will be requested.

By naming these files using an MD5 hashing algorithm, GWT ensures two things:

1. The browser can cache the file with no confusion. The next time you run the application, the startup is even faster.

[7] Java Enterprise Edition: http://mng.bz/815h.

(continued)

2. If you recompile, then the bootstrap code will force the browser to load the new version of your application (because it will be asking for a new filename). The next time you run the application, you'll get the latest version.

It's as simple as that.

Finally, in the war is a web.xml file. This file is also JEE standard and contains a set of directions for the web server. It should list the URL of any server-side code and in which package the server can find the Java code to run for that URL. This is a standard servlet-mapping definition, and it's something else we'll cover in chapter 7. As you've seen, you can also include a welcome file list in this file.

If you're thinking this sounds familiar, then yes, it should. GWT has taken many leaps and bounds to make the deployable application as standard as possible. If you have a modern application server, then deploying a GWT application should be as simple as deploying any other web application.

There's one more trick that can change the output of the GWT application for your needs: linkers.

2.9.1 Harnessing different linkers

The previous description of a GWT application's deployable part is the default view; it can be altered. Perhaps you want only a single script output or you want to use the application in a cross-site manner (in a different URL). In those and other cases, you can use GWT's linkers to help.

GWT's compilation is in two parts: Java to JavaScript and then linking and packaging. The default linker is the `com.google.gwt.core.linker.IFrameLinker` and is responsible for the default output you saw previously.

If you look in GWT's `Core` module and those it references, you'll see several linkers defined:

- `IFrameLinker`—This is the default linker and creates the bootstrapping process already described, loading the GWT application into an iFrame (it's named `std`).
- `SingleScriptLinker`—This produces a single script output where all code is in the nocache.js output (it's named `sso`).
- `XSLinker`—For use when you need a cross-site-compatible bootstrap sequence (named `xs`).
- `CrossSiteIframeLinker`—This uses an iFrame to hold the code, with a script tag being responsible to download it for use when you need a cross-site-compatible output.
- `DirectInstallLinker`—This is another cross-site-compatible approach. According to the documentation, this adds a script tag to the iFrame rather than downloading as a `String` and then inserting into the iFrame.

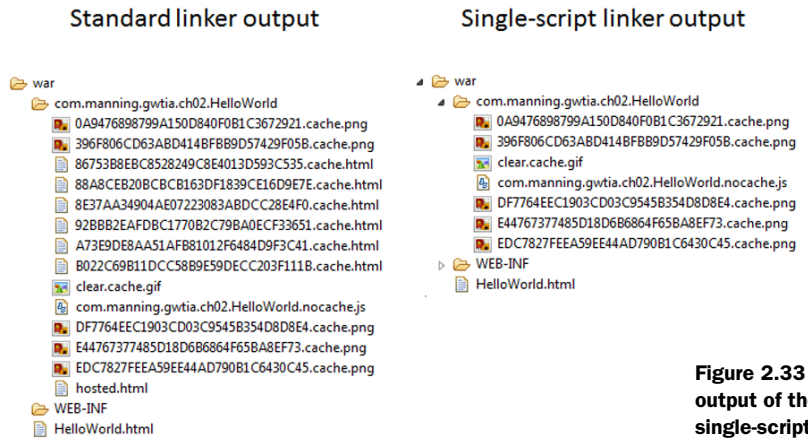

Figure 2.33 Comparison of output of the standard versus single-script linkers

We mention these so you know you have some options, and we won't explore them further. Most of the time the standard linker is what you need, so there's no need to fiddle. If you do find you need to use a different linker—maybe you're creating code that will be embedded into another person's site—then you can change the linker by adding it into your module definition. For example, to use the single-script linker you'd add

```
<add-linker name="sso" />
```

Doing so would change the output of the compiler, as shown in figure 2.33.

The difference shown in figure 2.33 is that the single-script linker has put all the JavaScript code into one file (com.manning.gwtia.ch02.HelloWorld.nocache.js—what a mouthful!), whereas in the standard output that file contains only the bootstrap code, which then selects the most appropriate cache.html file.

We won't cover linkers more in this book; as we said, most of the time you never need to think of them. But if you feel an urge to understand a little more, then you can find an interesting session[8] at Google IO in 2010 that talked about them (though it might be getting a little out of date).

We're at the end of the first part of building a GWT application. Before we summarize, we suggest that you play with the code and get a good feeling for all the concepts we've looked at. To that end, we'll introduce a brief section suggesting some things you might want to do to enhance your understanding.

2.10 Building on your understanding

Now that we've built a first application together, we suggest you spend a little time playing with what we've discussed before moving on. For example, you might try the following:

- Get comfortable with running development mode.

  Try running this chapter's simple application in several different browsers.

[8] Google IO session on linkers: http://mng.bz/5An6.

- Get comfortable with compiling.

  Try the different output styles and see the differences.
- If you're brave and eager to get ahead, try adding some different widgets and panels to the `RootPanel`.

  We'll discuss these more in the next chapter, but you can find a list of widgets online.[9] For example, you could write the following in the `onModuleLoad` method:

```
RootPanel.get().add(new Label("Hello World"));
RootPanel.get().add(new Button("Click Me!"));
SimplePanel testPanel = new SimplePanel();
testPanel.add(new Label("A label in a panel"));
RootPanel.get().add(testPanel);
```

- Try running the chapter's example in debug mode and set breakpoints in the `onModuleLoad` method.

  If you've changed that method to the code we showed, you'll have plenty of places to put the breakpoint.
- Use a `rename-to` attribute in the module file to make your code's URL more manageable.

  For example, change the first real line of the module file to the following and then see how to get your application running again:

```
<module rename-to="ch02">
```

  Hint: try clearing your browser cache, running your application in development mode, and then refreshing the code tree in Eclipse. Check the path shown for the bootstrap code in your HTML file (❷ in listing 2.2).

  Answer: see the sidebar in section 3.2.
- If you're eager to see a client/server application now, run the GPE New Web Application wizard you saw in figure 2.4.

  Use a different project name to avoid having problems in Eclipse, but don't turn off the generation of the sample application. You'll get a complete and more complicated GWT application to examine and play with.

2.11 Summary

You've reached the end of this chapter, and we expect you to be in possession of a simple GWT project that displays "Hello World!" on the screen. Not an earth-shattering application, but it has been useful. In this application we accomplished the following:

- Introduced some of the terminology used in GWT
- Showed how to use the GPE plug-in wizards to create a `HelloWorld` project
- Used Eclipse to run the project in development mode

[9] List of GWT widgets: http://mng.bz/C6p1.

- Examined the files produced and how they fit together to form an application
- Used the GPE to compile the application for deployment
- Looked at various ways you could find out where something has gone wrong in the application

Our hope is that this chapter has helped you become comfortable with how to create, run, and debug a GWT application, and you know what the structure of it looks like and how it's compiled. These are the steps you'll follow for every GWT application you create, and we can't overstate the importance of the benefits of development mode and the full debugging capabilities of an IDE. As your applications get bigger, you'll surely be happy this mode exists.

The next step is to take the structure you created and enhance some of the files to get to a more complicated application, and that leads us nicely into chapter 3.

JAVA/WEB DEVELOPMENT

GWT IN ACTION, Second Edition

Tacy • Hanson • Essington • Tökke

Google Web Toolkit works on a simple idea. Write your web application in Java, and GWT crosscompiles it into JavaScript. It is open source, supported by Google, and version 2.5 now includes a library of high-quality interface components and productivity tools that make using GWT a snap. The JavaScript it produces is really good.

GWT in Action, Second Edition is a revised edition of the best-selling GWT book. In it, you'll explore key concepts like managing events, interacting with the server, and creating UI components. As you move through its engaging examples, you'll absorb the latest thinking in application design and industry-grade best practices, such as implementing MVP, using dependency injection, and code optimization.

What's Inside

- Covers GWT 2.4 and up
- Efficient use of large data sets
- Optimizing with client bundles, deferred binding, and code splitting
- Using generators and dependency injection

Written for Java developers, the book requires no prior knowledge of GWT.

Adam Tacy and **Robert Hanson** coauthored the first edition of *GWT in Action*. **Jason Essington** is a Java developer and an active contributor to the GWT mailing list and the GWT IRC channel. **Anna Tökke** is a programmer and solutions architect working with GWT on a daily basis.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GWTinActionSecondEdition

Free eBook
SEE INSERT

"Covers all the newest features—a must-read for any GWT professional."
—Michael Moossen
Allesklar.com AG

"Clear, practical, efficient ... *in action*."
—Olivier Nouguier
Agilent Technologies, Inc.

"You will appreciate the abundance of tutorial material."
—Jeffrey Chimene
Systasis Computer Systems

"Up-to-date and detailed—a thorough guide."
—Olivier Turpin, IpsoSenso

"A quick and easy source for learning GWT."
—Levi Bracken, OPNET

ISBN 13: 978-1-935182-84-9
ISBN 10: 1-935182-84-6
54999
9 781935 182849

MANNING

$49.99 / Can $52.99 [INCLUDING eBOOK]