

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 7

brief contents

PART 1 BASICS 1

- 1** ■ GWT 3
- 2** ■ Building a GWT application: saying “Hello World!” 24
- 3** ■ Building a GWT application: enhancing HelloWorld 67

PART 2 NEXT STEPS 101

- 4** ■ Creating your own widgets 103
- 5** ■ Using client bundles 140
- 6** ■ Interface design with UiBinder 168
- 7** ■ Communicating with GWT-RPC 196
- 8** ■ Using RequestFactory 231
- 9** ■ The Editor framework 269
- 10** ■ Data-presentation (cell) widgets 309
- 11** ■ Using JSNI—JavaScript Native Interface 352
- 12** ■ Classic Ajax and HTML forms 387
- 13** ■ Internationalization, localization, and accessibility 417

PART 3 ADVANCED 457

- 14 ■ Advanced event handling and event busses 459
- 15 ■ Building MVP-based applications 483
- 16 ■ Dependency injection 516
- 17 ■ Deferred binding 538
- 18 ■ Generators 566
- 19 ■ Metrics and code splitting 591



Communicating with GWT-RPC

This chapter covers

- Using GWT-RPC to make remote calls
- Debugging communication between client and server
- Protecting against XSRF attacks

At this point in the book you've learned the basics of creating a GWT application, allowing you to do some great stuff in the browser. The next step will be learning to communicate with the outside world. GWT offers several tools for this, including HTML forms (chapter 12), `RequestBuilder` (chapter 12), `RequestFactory` (chapter 8), and GWT-RPC. HTML forms are exactly as the name implies, and `RequestBuilder` is your typical Ajax solution. But the next two, `RequestFactory` and GWT-RPC, are special in the sense that they allow you to send Java objects between the client and server, as opposed to JSON,¹ XML, or whatever other program you can dream up.

¹ JSON stands for JavaScript Object Notation. Learn more at <http://json.org>.

In this chapter we discuss GWT-RPC, or GWT Remote Procedure Call, which allows you to call Java methods on the server, passing and receiving back Java objects and primitives. You'll find this to be a convenient solution for general-purpose RPC calls when you're already running Java on the server.

Before we get into the specifics, we believe it's pertinent to explain how we structured this chapter, because we approach this topic differently than you may find in other books and tutorials. We start with a non-GWT implementation of a method and slowly transform it into a GWT-RPC call. This will allow us to start on familiar ground before we move into the specifics of GWT-RPC.

How to read this chapter

We devised this chapter to be used as both a tutorial and a reference. The chapter follows a single example from start to end, but GWT-RPC has lots of features, many of which won't apply to the development of the example application. We've grouped those features we didn't use in our example with related features.

If you're reading through this chapter as a tutorial, you'll end up with both a working application and an understanding of other features that will be applicable in certain situations. If you're instead looking for a reference, you can skip to any section in this chapter and expect it to be a complete reference.

With an understanding of the method we want to execute, we then look at how we can make the method's input and output types compatible with GWT serialization. Once this is complete, we'll implement the server side of the equation followed by the client and then follow up with how to add some cross-site request forgery (XSRF) protection to your RPC calls.

If you plan on developing the example project as you read through this chapter, we strongly recommend using the Google Plugin for Eclipse (GPE), which we introduced in chapter 2. It includes several features that will help you along the way and warn you of potential mistakes.

Before we dive into the tutorial, we begin with a broad overview of GWT-RPC and provide a few diagrams to give you an idea of the big picture.

7.1 Surveying GWT-RPC

As they say, the devil is in the details, and this holds true for GWT-RPC. But before we get to those, we want to provide a broad overview of how GWT-RPC works and the classes involved. This includes surveying the parts of GWT-RPC that are provided and the parts you'll need to write.

The parts that you'll need to write include your server-side business logic, the service interface, and potentially custom data objects that will hold the data that's passed between the client and server. The provided parts include everything else, namely the

code that glues the client and the server together. Much of this glue code is generated at compile time and is unseen by the developer.

At its core GWT-RPC is a way to have the GWT code on the client make an asynchronous call to Java code on the server. Although asynchronous calls have become more and more common, we still wanted to provide a few paragraphs to explain what they are and why we use them, so that's where we begin.

7.1.1 *Understanding asynchronous behavior*

If you've ever bought something online, you've participated in asynchronous behavior. The first step is to go to an online store like Amazon and purchase something, perhaps the latest edition of *GWT in Action*. The second step is to wait for it to arrive. But until it arrives, you don't sit out at the mailbox; you do other things. And when it does arrive, then you deal with it. That's asynchronous behavior.

That's how GWT-RPC and Ajax work. If you haven't used Ajax before, particularly if you've only done procedural programming, this feels a bit strange, but it's absolutely necessary. The unfortunate truth is that JavaScript is single-threaded. So if you performed a synchronous call, meaning you blocked execution until the RPC call returned, the browser wouldn't be able to handle other events, like mouse clicks. The browser will appear to have locked up, and that wouldn't be a user-friendly interface.

With this in mind, let's move on to looking at the classes involved when using GWT-RPC to see how this fits into the puzzle.

7.1.2 *Defining the GWT-RPC classes, interfaces, and annotations*

In order to help you understand all of the working parts of GWT, we present two tables, followed by a diagram to help you visualize how the parts fit together. The first table provides a list of GWT's classes, interfaces, and annotations that you'll use each time you write a new GWT-RPC service. The second table lists the classes and interfaces that you'll need to write yourself.

For each table we provide a short description of each item and point to the section in this chapter where you can get the details. The goal is to provide you with not only a brief overview but also a quick reference of where to get more information.

Let's begin with the GWT classes in table 7.1.

Table 7.1 Classes, interfaces, and annotations you'll use when creating a GWT-RPC service

GWT class	Explanation
<code>com.google.gwt.user.server.rpc.RemoteServiceServlet</code>	This is a specialized servlet that your implementation class on the server will extend. It provides serialization, deserialization, and auto-dispatching services. We cover the details in section 7.5.
<code>com.google.gwt.user.client.rpc.RemoteService</code>	This is a marker interface with no methods that will need to be implemented. Your implementation of this class will serve as the contract the client and the server will use for communication. We work through the details of this in section 7.5.

Table 7.1 Classes, interfaces, and annotations you'll use when creating a GWT-RPC service (*continued*)

GWT class	Explanation
<code>com.google.gwt.user.client.rpc.ServiceDefTarget</code>	This is an interface that will be implemented by your client-side service implementation. You won't implement this yourself; it will be implemented by the GWT-generated client-side service implementation. We discuss this more in section 7.6.
<code>com.google.gwt.user.client.rpc.RemoteServiceRelativePath</code>	This annotation is used when creating the service interface and specifies the URL to the server where the service is implemented. We discuss the details in section 7.5.
<code>com.google.gwt.user.client.rpc.IsSerializable</code>	This is a marker interface to indicate that the class is serializable. This marker would be placed on classes that you pass between the client and server. In section 7.4 we explain how this works and its compatibility with Java's <code>java.io.Serializable</code> interface.

If you read through the table, you'll notice that we hinted at the fact that GWT will generate some code for you. This is the key to how GWT-RPC works. At compile time GWT will inspect your service interface and generate the code required to make the calls to the server. This includes generating the serialization and deserialization code required to handle the Java objects passed between the client and server.

DEFINITION *Serialization* is the act of taking an object graph (for example, a Java object instance) and converting it into a binary or textual data, allowing it to be transported to another computer or stored on disk. *Deserialization* is the reverse, where you read the serialized data and convert it back into a Java object. The serialized Java objects are often binary data, but in some cases, like with GWT-RPC, the serialized data is (semi) readable text.

This is all made possible by GWT's deferred binding, which allows a code generator to inspect your Java code and generate additional Java code prior to all of the code being compiled to JavaScript. In order to use GWT-RPC you don't need to understand how deferred binding and generators work, but if you're curious, it's covered in chapter 19.

Let's look at the code you need to write, which table 7.2 summarizes.

Table 7.2 The classes and interfaces you need to create when using GWT-RPC

Construct	Explanation
Servlet implementation	You'll create a servlet that extends <code>RemoteServiceServlet</code> and implements your service interface. This will be the server-side implementation for your service. We cover the details in section 7.5.
Service interface	The service interface will define the remote methods that may be called from the client, and your servlet will implement this interface. See section 7.5 for more information.

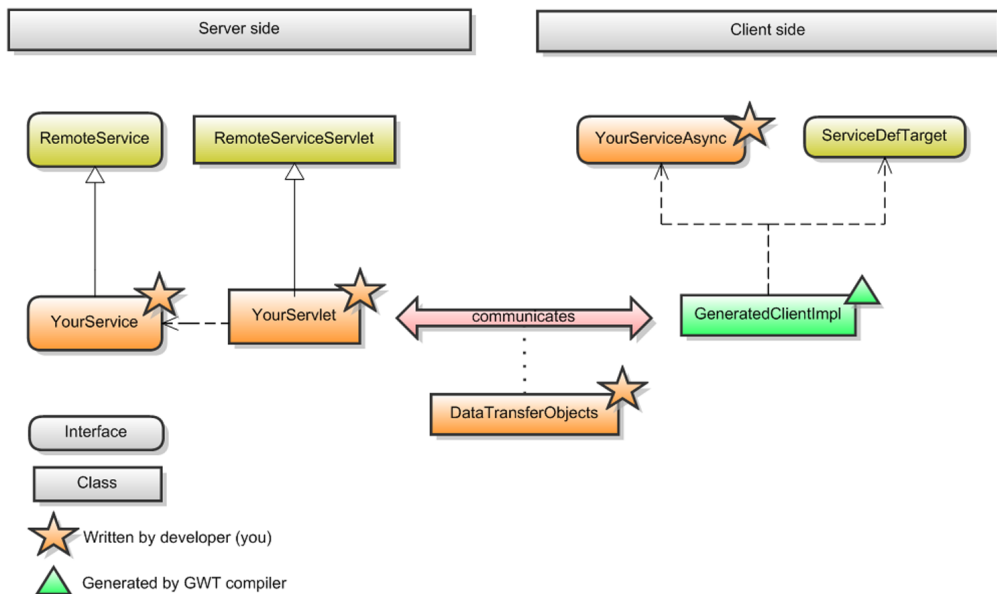
Table 7.2 The classes and interfaces you need to create when using GWT-RPC (*continued*)

Construct	Explanation
Asynchronous interface	This interface is a copy of your service interface but has modified method signatures in order to allow for asynchronous communication. This is covered in section 7.6.
Data transfer objects	Any data objects that you pass between the client and server will need to be created and will need to be tagged with the <code>IsSerializable</code> interface. This is covered in detail in section 7.4.

This list of code you need to write isn't daunting, but it's perhaps more than you expected. In particular, the asynchronous interface is an added piece of code you need to create because of the asynchronous requirement when communicating from the web browser.

The last thing we want to look at is a diagram of how all of these parts fit together to help you visualize how the system works, shown in figure 7.1. You can see how the client-side code uses the asynchronous interface, whereas the server-side code uses the regular service interface. If there's anything tricky about GWT-RPC it's the asynchronous behavior, because it not only adds one additional interface to create but also changes the way you'd normally code the client portion of the application.

As you can see in figure 7.1, you'll be writing code for both the client and server, so you need to understand how to lay out your packages properly in order for this to work.

**Figure 7.1** An overview of the GWT-RPC landscape

7.1.3 Understanding GWT-RPC package structure

As you learned earlier, you can't use any library in your GWT project because of the restrictions of the GWT compiler (compiles from source, limited JRE, and so on). So the question is, how can we mix server-side and client-side code in the same project, where the client-side code meets the GWT compiler requirements and the server-side code doesn't? And in the case of GWT-RPC we also have data objects that are used by both the client and the server.

Ultimately, the answer is simple. When you create your GWT project, using whatever means, you'll have created a client package. You've seen this throughout the book. This package should be used for both client-side and shared code. This works because although the GWT compiler compiles only the client package code into JavaScript, the Java compiler compiles all of the code, both client and server code, into Java bytecode. Therefore, the server-side code can make use of every class in your project.

So where does the server-side code go? The answer is, anywhere your client-side code isn't. Typically, if you have a package named `org.foo.project.client` for the client-side code, then you'd create `org.foo.project.server` for the server-side code. But again, the only real requirement is that the client-side code is segregated into its own package structure.

If you're using GPE, you'll see that it creates three packages for you: one for client, one for server, and one for shared. You can see this in figure 7.2.

Looking at the figure, you may have noticed that this seems to break the rule in that the shared code isn't inside the client package. The way this works is that the plug-in alters the default source package to include both client and shared. This is achieved by adding the shared package as a source code package in the module configuration that the GWT compiler should compile. For example, here's a snippet of module configuration that was generated by the Google plug-in:

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

As you read through this chapter we use the package hierarchy created by GPE, so throughout the chapter we'll use the client package for client-only code and the shared package for code used by both the client and server.

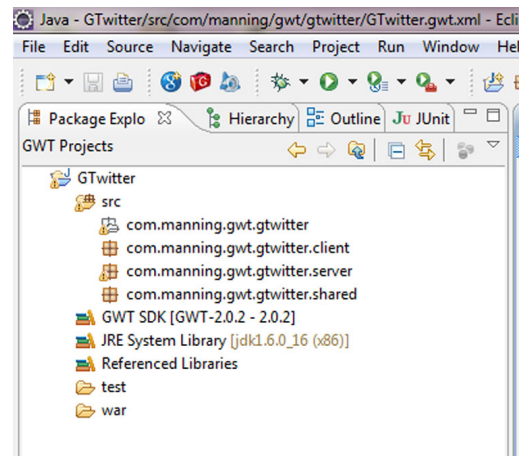


Figure 7.2 The default package structure that's created when you use GPE includes client, server, and shared packages.

Now that you have a broad overview of GWT-RPC, let's write some code. In the sections that follow we show you how to create a simple Twitter client that has a lot of the same challenges you can expect when using GWT-RPC.

7.2 *Learning GWT-RPC with Twitter*

In selecting an example to demonstrate GWT-RPC, we settled on the idea of a simple Twitter client. Twitter, once the secret of techies, has gone mainstream, with celebrities and companies using it to keep their fans informed. So Twitter seemed like a good choice because GWT is a tool for the modern web application, and social media is a cornerstone of that class of application.

NOTE If you want to follow along and run the example project, you should create a new GWT project with the name `GTwitter` and a base package of `com.gwtia.ch07`. Or you can download the project source code and follow the provided instructions.

The Twitter client that we'll build in sections 7.3 through 7.6 is named `GTwitter`. It's a simple client that will only display the latest tweets for a specific Twitter user. The GWT application will send the request to Twitter via our web server, so the client won't be communicating with Twitter directly. This offers some advantages over a client-only solution in that we can extend the server code over time to also return content from Facebook, Orkut, and RSS feeds, to name a few. On top of that we can also add caching if we desire to improve performance, something that you can't do in a client-only solution.

`GTwitter` will make use of `Twitter4J`,² an open source Java library for interacting with the Twitter API. You'll need to download `Twitter4J` if you wish to follow along with the example. The examples in this chapter use version 2.1.1 of `Twitter4J` and only require the use of `twitter4j-core-2.1.1.jar`. If you wish to use a newer version of the `Twitter4J` API, you may be required to make some changes to the examples in order to match any changes made to the API.

Including the Twitter4J library in your project

In order to develop and run the `GTwitter` example, you'll need to download the `Twitter4J` jar file and add it to the `/war/WEB-INF/lib/` directory of your project, as well as add it to the classpath of your IDE. You'd also do this for any other jar file that you need to include on the server side of your application. This is different from libraries used to create the client side, which don't need to be placed in `/war/WEB-INF/lib/` or even be deployed with the application to the server.

We'll develop the `GTwitter` client over the next few sections. This includes looking at model considerations in section 7.4, developing the server component in section 7.5,

² `Twitter4J` can be downloaded from <http://twitter4j.org>.

and writing the client portion in section 7.6. But first we begin by looking at an example of a non-GWT Twitter4J call from a Java application. This will be a good way to see what issues and limitations come to play when using GWT-RPC.

7.3 Fetching data from Twitter the non-GWT way

With the Twitter4J library in your classpath, you need a few lines of code to fetch a feed and display the results. Take a look at the following listing, and then we'll explain it.

Listing 7.1 A non-GWT version of fetching a Twitter feed using Twitter4J

```
package com.gwtia.ch07.server;

import twitter4j.*;

public class TwitterServiceImpl
{
    public static void main(String[] args) throws Exception
    {
        1 | TwitterServiceImpl impl = new TwitterServiceImpl();
          | ResponseList<Status> resList = impl.getUserTimeline("ianchesnut");

        for (Status status : resList) {
            System.out.println(status.getCreatedAt()
                               + ": " + status.getText());
        }
    }

    public ResponseList<Status> getUserTimeline (String screenName)
        throws TwitterException
    {
        Twitter twitter = new TwitterFactory().getInstance();
        return twitter.getUserTimeline(screenName);
    }
}
```

Fetch a user's feed

2 Print feed contents

3 Helper method

As you can see, fetching data from Twitter doesn't require a lot of code when you use the Twitter4J library. Still, we'll explain what's going on before we point out the issues you'll face when trying to port this to GWT-RPC.

1 In the `main` method you create a new instance of the class and then call the `getUserTimeline()` method, passing the Twitter screen name of the user you want tweets for. This is essentially what the GWT-RPC client-side code will look like.

The part that isn't GWT-compatible is the fact that this is a synchronous call, meaning that your code waits until the method returns the `ResponseList`. As you saw in section 7.1, GWT-RPC solves this by using asynchronous calls, which will alter the way you call the service.

2 Next, you iterate over the results returned from the call. Note that the `ResponseList` object that you're iterating over and the `Status` objects in the list are classes from the Twitter4J library. These classes aren't GWT-compatible either, because GWT requires the Java source code in order to compile it to JavaScript.

This is one of the more common issues you'll run into with GWT-RPC, and the solution is to make use of data transfer objects (DTO).³ What this means is that you need to have your own versions of `ResponseList` and `Status` that are GWT-RPC-compatible (the DTOs), and copy the data into these.

DEFINITION A *data transfer object* is a design pattern used for transferring data between different applications. A DTO is characterized by not having any behavior, meaning it doesn't "do" anything; it stores data. DTOs are often used when it's not possible to transfer your business or data access objects. In the case of GTwitter, the `Twitter4J` objects can't be serialized by GWT, so you'd want to create DTO objects that are serializable in order to transfer the data to the client browser.

③ The last part of the code example is the part that will run on the server. Here you use the `Twitter4J` API to fetch the status data and return it to the caller. The best thing about this is that this runs on your server, so you don't need to make any changes.

You'll have to do a little extra work here, like copying the `ResponseList` into the GWT-RPC-compatible DTO that we mentioned. And this is exactly where we'll start. In this next section we'll define the requirements of the objects returned from a GWT-RPC call, as well as discuss some common issues like trying to use JPA entities as DTOs.

7.4 Defining a GWT-RPC-compatible model

When deciding on the data types to pass between the client and server, the most important concern is to make sure that the data types can be serialized by GWT. Table 7.3 provides a list of these types.

Table 7.3 Data types that can be serialized for GWT, not including types with custom serializers

Java type	Explanation
Primitive	Includes byte, char, short, int, long, float, double, and boolean.
Java enum	Enumeration constants are serialized by name only; any other field values won't be carried over.
Array	Must be an array of other serializable types.
Serializable user defined	A class that implements the <code>IsSerializable</code> or <code>Serializable</code> marker interface, along with a no-arg constructor and serializable fields.

You may have noticed that this list is missing quite a few basic types, like `java.util.Date` and `java.lang.Integer`. In GWT these are handled by creating custom serializers, and GWT ships with a bunch of them. We'll discuss how to create your own custom serializer soon, but for now table 7.4 provides a list of Java types for

³ For more information on data transfer objects, refer to its Wikipedia page at http://en.wikipedia.org/wiki/Value_object.

which GWT provides custom serializers. For reference, these serializers are found under the subpackages of `com.google.gwt.user.client.rpc.core` in the `gwt-user.jar` file.

Table 7.4 Java classes for which that GWT provides custom serializers out of the box

Package	Classes
<code>java.lang</code>	<code>Boolean</code> , <code>Byte</code> , <code>Character</code> , <code>Double</code> , <code>Float</code> , <code>Integer</code> , <code>Long</code> , <code>Short</code> , <code>String</code>
<code>java.sql</code>	<code>Date</code> , <code>Time</code> , <code>Timestamp</code>
<code>java.util</code>	<code>ArrayList</code> , <code>Collection</code> , <code>Date</code> , <code>HashMap</code> , <code>HashSet</code> , <code>IdentityHashMap</code> , <code>LinkedHashMap</code> , <code>LinkedList</code> , <code>Map</code> , <code>TreeMap</code> , <code>TreeSet</code> , <code>Vector</code>

As you can see, GWT either provides built-in serialization for your data or gives you a way to build your own serializer. There's a good amount of information to cover here, so let's start by looking at using `Serializable` and `IsSerializable`, which directly relate to our example.

7.4.1 Using the `Serializable` and `IsSerializable` interfaces

GWT provides two marker interfaces for identifying classes that can be serialized. The first is Java's own `java.io.Serializable`, and the second is `com.google.gwt.user.client.rpc.IsSerializable`. A marker interface has no methods that need to be implemented and acts as a marker to let other classes know that certain semantics apply. The semantics in this case are those required for the class to be serializable by GWT. By implementing either of these interfaces you're agreeing that the class meets the following requirements:

- All nonfinal and nontransient fields must in turn be serializable by GWT.
- The class must have a zero-arg constructor.

If you've used `java.io.Serializable` before, you might have noticed that this isn't at all compatible with Java's definition of a serializable class. It's provided merely as a convenience for developers who wish to reuse their database entities that already implement this interface. In GWT the semantics of the two interfaces are exactly the same. In general you should use `IsSerializable` because it's more correct.

IsSerializable vs. Serializable

When GWT was first released, the designers felt that because GWT couldn't comply with the semantics of Java's own `Serializable` interface, they should create a GWT-specific `IsSerializable` interface. This caused some pain for developers who wanted to pass their database entities directly to the browser via GWT. After quite a bit of discussion, the GWT community opted to allow the use of `Serializable` as the equivalent of `IsSerializable` because the benefit was deemed greater than the possible semantic confusion that it might cause.

Let's take this information and apply it to the GTwitter client. As you might recall, the Twitter4J library returned a `ResponseList<Status>`, essentially a list of `Status` objects. To make the return objects more generic so that they can be used for other types of values, our model will consist of a list of `FeedData`. The `FeedData` class is presented in the next listing.

Listing 7.2 GTwitter serializable model object

```
package com.gwtia.ch07.shared;

import java.util.Date;
import com.google.gwt.user.client.rpc.IsSerializable;

public class FeedData implements IsSerializable
{
    private Date createdAt;
    private String text;

    public FeedData() {}

    public FeedData(Date createdAt, String text) {
        this.createdAt = createdAt;
        this.text = text;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public String getText() {
        return text;
    }
}
```

In listing 7.2 the only GWT-specific remnant is that the bean implements `IsSerializable` and the package in which the class resides. Specifically, the class is in the shared package, which will be one of the packages compiled by the GWT compiler, assuming you're using the default-generated module configuration for the project. Strictly speaking, you can place your DTO classes in any package that your client-side application uses. See chapter 2 for how to configure your project and alter the `<source>` packages.

Implementing `IsSerializable` will cover many use cases but not all. In some cases you may want to use the same classes as your JPA or JDO entities, and we'll look at some special rules regarding their use.

7.4.2 Special considerations when using JPA/JDO model objects as DTOs

If you plan on using the Java Persistence API (JPA) or Java Data Objects (JDO) and want to use the same entity objects as DTOs that will transfer data between server and client, you have two issues to consider. First is that GWT's `IsSerializable` interface is no longer desirable, and `java.io.Serializable` should be used instead. As mentioned previously there are no semantic differences from GWT's point of view, but your persistence mechanism will require it.

The second and more important issue is how GWT handles enhanced persisted classes. Some persistence mechanisms work by enhancing either the source code or bytecode of the class prior to deploying it to a server. A good example of this is the JDO support for Google App Engine (GAE), which will use the Data Nucleus to enhance your bytecode prior to deploying it to the server. We'll provide an example of this in chapter 12 when we discuss using GWT with GAE.

When a class is enhanced, additional static or instance fields are added to the class. These won't be present in the source code that GWT compiles, which means that compiled client-side code and the server-side code for these same classes will differ. One of the enhancements in GWT 2.0 is the ability to handle this situation, but be warned that it may not work with all persistence tools.

GWT will consider a class as being enhanced if one or more of the following are true:

- For JPA, the class is annotated with `javax.persistence.Entity`.
- For JDO, the class is annotated with `java.jdo.annotations.PersistenceCapable` with the attribute `detachable=true`.
- The GWT module file includes the fully qualified class name in the `rpc.enhancedClasses` configuration property.

GWT handles these classes differently. When sending these classes from the server to the client, it will use Java serialization to serialize the added fields but won't deserialize them on the client side. This means the client won't be able to access this persistence engine-specific data. If you send the same object back to the server, this data will be deserialized on receipt so that it can be used by the persistence engine.

In order to do this work, GWT makes some assumptions about the data entity. It assumes that the object is in a detached state. This means that changes to the fields of the object don't affect persistent storage. Furthermore, GWT will also assume that all nonstatic and nontransient fields are serializable.

In general, if you fall into this use case, you'll need to experiment a little to see if GWT-RPC works with your persistence tool or if some tweaking is required. If some research is necessary, we suggest starting with the archives of the GWT General Discussion mailing list⁴ and perhaps even using it to get help from others who've already trod the same ground.

But what if your serialization needs are more complex, and you need to customize how the serializer works? GWT provides a mechanism for that as well.

7.4.3 Developing custom serializers

In some cases GWT developers have had a need to customize the process of serializing their data objects. This is usually a last resort, used when GWT can't automatically handle this for you. Some of the common reasons for this include the following:

- The default serialization causes performance issues for a complex object.

⁴ The GWT general discussion list is found at <http://groups.google.com/group/google-web-toolkit>.

- The class that needs to be serialized doesn't implement `IsSerializable` or `Serializable`.
- The class that needs to be serialized doesn't have a zero-argument constructor.

When you do need to create your own custom serializer, as you will for the Twitter example, GWT makes this a relatively easy task. For the purposes of example we'll use the DTO that we created for the GTwitter client, which would be serializable by GWT, but then make some changes so it isn't—and so you need to create a custom serializer, as shown in the following listing.

Listing 7.3 An unserializable GTwitter DTO

```
package com.gwtia.ch07.shared;

import java.util.Date;

public class BadFeedData
{
    private Date createdAt;
    private String text;

    public BadFeedData(Date createdAt) {
        this.createdAt = createdAt;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

In listing 7.3 you take the GTwitter DTO and remove the `IsSerializable` interface and the no-arg constructor, making it GWT-incompatible. In addition you add one argument to the constructor so that you can fully exercise the capabilities of a custom serializer. Now let's look at how to create a custom serializer for the modified DTO.

A custom serializer is a class that you'll place in the same package as the DTO, with specific requirements for naming the class. You don't have any classes to extend or any interfaces to implement. Before we codify the rules, let's see what the custom serializer for this class looks like.

Listing 7.4 A custom serializer for the BadDataFeed DTO

```
package com.gwtia.ch07.shared;

import java.util.Date;
import com.google.gwt.user.client.rpc.*;
```

```

public class BadFeedData_CustomFieldSerializer
{
    public static void serialize(SerializationStreamWriter ssw,
        BadFeedData instance) throws SerializationException {
        ssw.writeObject(instance.getCreatedAt());
        ssw.writeString(instance.getText());
    }

    public static BadFeedData instantiate(SerializationStreamReader ssr)
        throws SerializationException {
        return new BadFeedData((Date) ssr.readObject());
    }

    public static void deserialize(SerializationStreamReader ssr,
        BadFeedData instance) throws SerializationException {
        instance.setText(ssr.readString());
    }
}

```

1 Special class name

2 Send instance properties to stream

3 Instantiate new instance from stream

4 Set properties from stream

The first thing to note is that the class name is the name of the DTO plus the postfix `_CustomFieldSerializer` ①. This scheme is the unfortunate side effect of Java 1.4. GWT was initially released without support for Java 5 language features like generics. Because of this, custom serializers make use of this naming rule.

Getting into the body of the class, you'll need to write three methods. The first is the `serialize()` method ②. The method takes a writer and an instance and writes the properties of the instance to the writer. The writer allows you to write Java primitives, `Strings`, and `Object` types. `Object` types are then serialized by their own serializer. For example, in our serializer we write a `java.util.Date` instance to the stream, and because GWT ships with its own customer serializer for `Date`, that serializer will be called in order to convert the object to a primitive value.

The order in which you write the values to the stream is important, because when you deserialize you need to read the values in the same order. In this case the DTO doesn't have a zero-arg constructor, so you need to provide an `instantiate()` method ③. This method takes a reader and uses it to create and return a new object instance. If you had a zero-arg constructor you wouldn't need to provide this method at all, and GWT would handle creating a new instance for you. But you don't have one, so you need to handle this. When you serialize the object, you write the value of the `createdAt` field first, and that's because you need it for the constructor. Note that you call `readObject()` on the reader. Because the writer would in turn use the `Date` custom serializer to write the `Date` value, that same serializer is used to read the `Date` value.

Finally, you need to provide a `deserialize()` method ④. The `instantiate()` method has already created the object you'll return and set the `Date` value; now you need to read in any remaining values and use them in the setters of the instance. In this case, that means you need to read the text and set the value in the instance.

As with the DTO itself, the custom serializer is shared code, used by both the server and the client. Because of this you're limited to the parts of the JRE that can be compiled to JavaScript, as you are with any GWT client code.

So now that we've explained it, let's boil this down to a simple list that you can use as a reference when you need to create your own customer serializer:

- The serializer must be in the same package as the object it serializes/deserializes.
- The name is the name of the DTO plus `_CustomFieldSerializer`.
- The required `serialize` method has a signature of `public static void serialize(SerializationStreamWriter ssw, T instance)`.
- The required `deserialize` method has a signature of `public static void deserialize(SerializationStreamReader ssr, T instance)`.
- The optional `instantiate` method has a signature of `public static T instantiate(SerializationStreamReader ssr)`.
- All methods may rethrow `SerializationException`.

As you've seen, depending on your project needs, your GWT-compatible model can be either extremely easy or quite complex. Fortunately, this is likely the most complex of the GWT-RPC topics, and things will fall into place from here on out.

So with that we move to the server side of the equation, which is perhaps the easiest because it's familiar territory for Java web developers.

7.5 *Building and deploying the server side*

In section 7.4 we defined a GWT-compatible model for our GTwitter client. Now we want to look at coding the server side of the equation and deploying it to a servlet container.

GWT makes developing the server-side code easy. The first rule is that this is the server, so anything goes. You're not limited to what the GWT compiler can accept, because your server-side code isn't processed by the GWT compiler. You can use all of your favorite Java libraries and aren't limited to specific classes in the JRE as you are with your client-side code. If you felt a bit overwhelmed by all of the rules of GWT-compatible serialized objects, this section will be a nice break.

This section assumes some familiarity with servlet containers, what a servlet is, and how to deploy an application to a servlet container. If you're unfamiliar with servlet containers, we suggest going through the process of downloading Apache Tomcat⁵ and walking through some of the introductory material before continuing.

Three topics need to be addressed when writing and deploying server-side code: writing the servlet, deploying the servlet, and handling exceptions. Because the GTwitter example throws an exception, we start with handling exceptions and show how to make an exception GWT-compatible.

⁵ Apache Tomcat downloads and documentation are available at <http://tomcat.apache.org/>.

7.5.1 Handling exceptions

Besides sending Java objects to and from the client, GWT supports having the server throw exceptions that are handled on the client. This is handled by enforcing the same methods we covered in section 7.4 when discussing how to make a GWT-compatible DTO. And an exception instance, from GWT's point of view, is merely another object that needs to be transported between the client and server.

To make this easier, you may use any of the exceptions provided by GWT's JRE Emulation Library without having to do anything else. These include `Throwable`, `Exception`, `IllegalArgumentException`, `NullPointerException`, and `NumberFormatException`, to name a few. In addition, because these already implement `Serializable`, you can create a subclass of these without having to do any additional work. But as you saw with DTO handling, there may be cases where you need to create a custom serializer. Again, custom serialization for exceptions is no different than handling DTOs; all the same rules apply.

Getting back to the GTwitter example, when we look at the non-GWT-compatible version we see that the Twitter4J library will throw `TwitterException` if it should fail to load the feed data for the specified user. This exception isn't part of the GWT Emulation Library and is therefore incompatible with GWT. To handle this, you can create your own `GTwitterException` and use it instead.

Listing 7.5 An application-specific exception for the GTwitter service

```
package com.gwtia.ch07.shared;

public class GTwitterException extends Exception {
    public GTwitterException() { }

    public GTwitterException(String reason) {
        super(reason);
    }
}
```

The `GTwitterException` is about as basic as they come. You extend `Exception` and provide some constructors. Because you extend `Exception`, which in turn implements `java.io.Serializable`, you meet that requirement. In addition to that, you provide a no-arg constructor to meet the remaining requirement.

As you saw with the DTO, this class lives in a package that will be compiled to JavaScript by the GWT compiler. This is because you'll be using this class on both the client and server side of the application.

Now that you have a GWT-compatible DTO and exception, let's define the service interface that will need to be implemented on the server.

7.5.2 Defining the service interface

Although creating the service interface isn't as complicated as dealing with GWT-compatible DTOs, you still have a few GWT specifics you need to deal with.

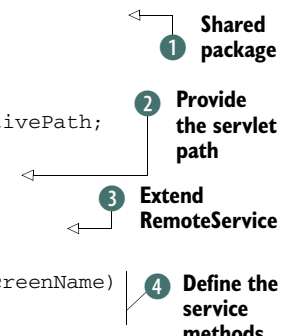
The easiest way to explain how this works is to start with a code example and then provide an explanation. The following listing shows the service for the GTwitter application.

Listing 7.6 The server-side interface for the GTwitter application

```
package com.gwtia.ch07.shared;

import java.util.ArrayList;
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

@RemoteServiceRelativePath("service")
public interface TwitterService
    extends RemoteService
{
    public ArrayList<FeedData> getUserTimeline(String screenName)
        throws GTwitterException;
}
```



1 Shared package

2 Provide the servlet path

3 Extend RemoteService

4 Define the service methods

This code is devilishly simple, but there's quite a bit to it. Let's explore each of part of listing 7.6.

❶ Like the DTO, the service is used by both the client side and the server side. So this class must reside in a directory that will be compiled to JavaScript by the GWT compiler.

❷ When GWT compiles the code to JavaScript and turns the service into an Ajax call, it needs to know the location of the servlet on the server. The `@RemoteServiceRelativePath` annotation allows you to do this. The value, "service" in this case, is appended to the result of `GWT.getModuleBaseURL()`. The URL that this translates to depends on where you deploy the compiled GWT code on your server. Most of the time this will make sense, but there are always exceptions. Because of this, this annotation is optional and can be specified when you write the client-side code. We'll look at that more in section 7.6, but for now it suits our purposes to use the annotation.

❸ Your service needs to extend `RemoteService`. This is another one of those marker interfaces, as you saw earlier when we looked at `IsSerializable`, and it doesn't require you to implement any additional methods. The purpose of this interface is to trigger the generation of the code by the GWT compiler that will do the real work of allowing you to (somewhat) transparently call your server-side service. If you want to know more about how compile-time code generation is triggered in GWT, you should read about deferred binding, covered in chapter 19.

Why does Eclipse show "Missing asynchronous interface"?

If you're developing the application using GPE, Eclipse will report the error "Missing asynchronous interface TwitterServiceAsync." The reason is that the plug-in is anticipating that you also need to create a client-side asynchronous interface. We create this interface in section 7.6. If you're coding along with the example, you can ignore this error for now.

④ You need to define the server-side service that you'll implement. No additional rules apply here other than what we already defined. Specifically, all of the parameters, return types, and exceptions need to be compatible with GWT serialization. The GWT compiler will inspect this method during code-generation time in order to know what type serializers must be included in the resulting JavaScript output.

For this same reason it's important to be as specific as possible when specifying the types, because this will result in code bloat. For example, specifying that this method throws `Exception`, as opposed to `GTwitterException`, requires the generated code to include every available `Exception` type. So instead of only `GTwitterException`, the compiled code would include closer to 30 exception types and the code to serialize all of them.

NOTE Be as specific as possible when specifying the types. Using `java.util.List` instead of `java.util.ArrayList` will force the compiler to include every `List` implementation in the JavaScript output, which results in larger files that the browser will need to download. So be specific with your types.

The flip side of that coin is that if your code were to throw a `NullPointerException`, it wouldn't be serialized and sent to the client as such. Instead it would be returned as a generic failure. So be as specific as possible, but be sure to include every exception that you'd want sent to the client.

NOTE When compiling GWT code to JavaScript, you can use the `-gen <DIR>` switch to tell the compiler to output the generated Java code to the specified directory. This will include generated field serializers, type serializers, and RPC proxy code. Besides this being a way to get a handle on what's being generated, it's also a great way to learn about the inner workings of GWT-RPC.

Now that we've defined the service, we can write the server-side servlet that will act as an implementation of our service.

7.5.3 Writing the servlet

When using GWT-RPC, the server-side code takes the form of a servlet, although it doesn't look much like one. For many of us, using Spring, Struts, or other servlets may be a relic of the past, because our favorite framework provides an abstraction above the basic servlet. GWT-RPC is the same, and although it's a servlet, it doesn't resemble one.

Let's start the conversation as we've done before, presenting the code up front followed by a detailed explanation. The next listing presents our GTwitter servlet implementation.

Listing 7.7 The server-side implementation of the GTwitter client

```
package com.gwtia.ch07.server;

import java.util.ArrayList;
import twitter4j.*;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
```

← 1 Server package

```

import com.gwtia.ch07.shared.*;

public class TwitterServiceImpl
    extends RemoteServiceServlet implements TwitterService
{

    public ArrayList<FeedData> getUserTimeline (String screenName)
        throws GTwitterException
    {

        ArrayList<FeedData> result = new ArrayList<FeedData>();
        Twitter twitter = new TwitterFactory().getInstance();

        try {
            ResponseList<Status> responses
                = twitter.getUserTimeline(screenName);

            for (Status status : responses) {
                result.add(
                    new FeedData(status.getCreatedAt(), status.getText()));
            }
        } catch (TwitterException e) {
            throw new GTwitterException(e.getMessage());
        }

        return result;
    }
}

```

2 Defines superclass and service interface

3 Create Twitter instance

4 Fetch timeline

5 Copies data into a GWT-RPC-compatible DTO

6 Rethrows as GWT-RPC-compatible exception

Listing 7.7 is a little longer than the original non-GWT version of this code that we provided in section 7.3, so let's look at what's going on.

1 This class lives in the server package of your project. Specifically, the rule is that this class can live in any package *not* compiled to JavaScript by the GWT compiler. The reason is that server-side GWT-RPC code uses classes outside of the JRE Emulation Library and therefore is unable to be compiled to JavaScript.

2 When you define the class, you need to extend `RemoteServiceServlet`. This is the GWT servlet that handles the incoming request from the client, deserializes the incoming data, and executes your method. In addition, you implement the `TwitterService` interface that you created in the previous section. The `RemoteServiceServlet` uses reflection to read your interface definition so that it knows what methods may be called remotely, meaning that it won't allow calls to methods not defined in your interface.

The code you use to get the tweets from Twitter is the same as it was in the non-GWT version of the code. You use the Twitter4J library's `TwitterFactory` to create a `Twitter` session **3** and then call `getUserTimeline()` to fetch the tweets for that user **4**.

5 Once you have the list of `Status` objects, you need to convert them to a GWT-compatible DTO. In this case you copy them into the `FeedData` objects defined in section 7.4. The need to copy data, particularly complex data, is one of the major

complaints with GWT-RPC. One popular remedy for this is to use Dozer,⁶ a library that provides the tools to copy data between objects of different types. If you find a need to copy complex data structures, you may want to consider using Dozer.

⑥ When you're using the Twitter4J library, it will throw a `TwitterException` if it's unable to fetch the requested data. This could occur in your application if someone requested tweets for a screen name that doesn't exist. Here you convert the `TwitterException` into your `GTwitterException`. Recalling the earlier discussion on serialization, you need to do this because `TwitterException` isn't serializable by GWT, and you therefore can't pass this exception back to the client. Your exception, on the other hand, is serializable by GWT and is defined in your service interface, so exceptions of this type will be sent to the client.

And there you have it, a GWT-RPC servlet. As you can see, the fact that you're using GWT or a servlet is hidden rather well. You can even test our method without a servlet container at all. Listing 7.8 provides some test code that you can add to your GWT servlet so that you can also run it as an application.

Listing 7.8 Test code to verify the functionality of the GTwitter server

```
public static void main(String[] args) throws Exception
{
    TwitterService impl = new TwitterServiceImpl();
    ArrayList<FeedData> resList = impl.getUserTimeline("ianchesnut");

    for (FeedData status : resList) {
        System.out.println(status.getCreatedAt() + " " + status.getText());
    }
}
```

Executing this code will result in output similar to what you see here, which shows several tweets displayed along with the date and time they were posted:

```
Sat Mar 21 21:39:43 EDT 2009 Super test
Sat Mar 21 21:13:22 EDT 2009 test already
Sat Mar 21 21:12:48 EDT 2009 Test again
```

This is basic output, but it shows that our code works. That's good enough for us right now until we develop the client side of our application. So now that we have our servlet, we'll want to deploy it to a servlet container.

7.5.4 Deploying the servlet

As noted throughout this book, when you create a new GWT project, it creates a war directory at the top level of the project. This is your web root, where the files destined to be deployed are stored. Within this directory is a WEB-INF folder that contains the servlet deployment descriptor file `web.xml`.

⁶ Dozer is available at <http://dozer.sourceforge.net/>.

To configure the servlet, open this file, `war/WEB-INF/web.xml`, and add the servlet configuration. The following listing shows the full `web.xml` after making our changes.

Listing 7.9 Servlet deployment descriptor for the GTwitter application

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

<web-app>
  <servlet>
    <servlet-name>twitterServlet</servlet-name>
    <servlet-class>
      com.gwtia.ch07.server.TwitterServiceImpl
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>twitterServlet</servlet-name>
    <url-pattern>/GTwitter/service</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>GTwitter.html</welcome-file>
  </welcome-file-list>
</web-app>
```

1 Define servlet

2 Define mapping

3 Define the welcome file

If you're following along, and you used the Eclipse plug-in or command-line tool to create the basic app structure, you'll see that you stripped out the example servlet configuration and are left with only what you need for this project. What's left are three important sections, so we'll explain each.

1 The `<servlet>` element is used to register a servlet with the servlet container. Here you give the servlet an arbitrary name with the `<servlet-name>` element, in this case `twitterServlet`, and provide the full package and class name of the servlet in the `<servlet-class>` element.

2 Next, you need to map the registered servlet to a URL, and that's where the `<servlet-mapping>` comes into play. Here you specify the same `<servlet-name>` that you used in the `<servlet>` element. This is what ties this mapping to that servlet. In addition, you use `<url-pattern>` to specify the path to the servlet. The URL that you specify is relative to the servlet context, or the root of the project. The path is the directory where the project is deployed plus the name you specified in the `@RemoteServiceRelativePath` annotation in the service interface. In this case the GWT application is deployed to `/gtwitter`, and the path used in the annotation was `service`, which gives you the resulting path shown in listing 7.9.

3 The last item in your `web.xml` is a pointer to the welcome file. If configuring servlets is new to you, this is the file that's served if the user navigates to `/` on the application. It's the default page when none is defined.

TIP The most common cause for a non-working GWT-RPC service is when the URL pattern defined in the mapping doesn't match the URL used by the client application. In section 7.7 we show you how to verify the URL used by the client when we discuss how to debug remote calls.

With the servlet written and registered in the servlet descriptor, we can move on to coding the client to call the service.

7.6 Writing the client

We're almost at the end—only two steps left. As you might expect, we need to write the code that calls the server and receives the result. As we explained earlier, a GWT-RPC call is made asynchronously. And you may have noticed that the service interface that we created for the server side won't work for an asynchronous call. The reason is that it doesn't allow for a callback, meaning that there's no way to handle the result that's returned at some time after the call is made. We need to start by creating one last interface, which will allow for a callback.

7.6.1 Defining the asynchronous interface

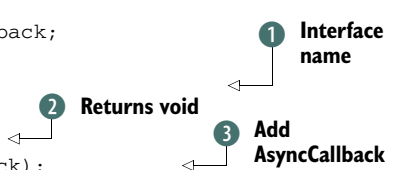
The asynchronous interface is the interface that the client side is coded to. It's a mirror image of the service interface you already defined, assuming that the mirror you use is one of those funhouse mirrors. Specifically, it's an altered version of the service interface, altered in a specific way. The next listing contains the code for this interface.

Listing 7.10 Asynchronous interface used by the client side of the GTwitter application

```
package com.gwtia.ch07.shared;

import java.util.ArrayList;
import com.google.gwt.user.client.rpc.AsyncCallback;

public interface TwitterServiceAsync
{
    void getUserTimeline(String screenName,
        AsyncCallback<ArrayList<FeedData>> callback);
}
```



1 Interface name

2 Returns void

3 Add AsyncCallback

This looks a lot like the service interface, but with two differences. The first is the name of the interface. The asynchronous interface uses the same name as the service interface plus `Async` ❶. So the `TwitterService` service interface will have a `TwitterServiceAsync` asynchronous interface. This interface must also reside in the same package as the service interface.

Second is the method definition. All methods in the asynchronous interface must return `void` ❷ and have an additional `AsyncCallback` parameter added ❸. The `AsyncCallback` uses generics, so you'll need to specify the return type here. If the return type of the method is `void`, you'll specify `java.lang.Void`.

Of interest here is that this method doesn't declare any exceptions. That's because the handling of the server side is taken care of by the `AsyncCallback`. We'll look at the

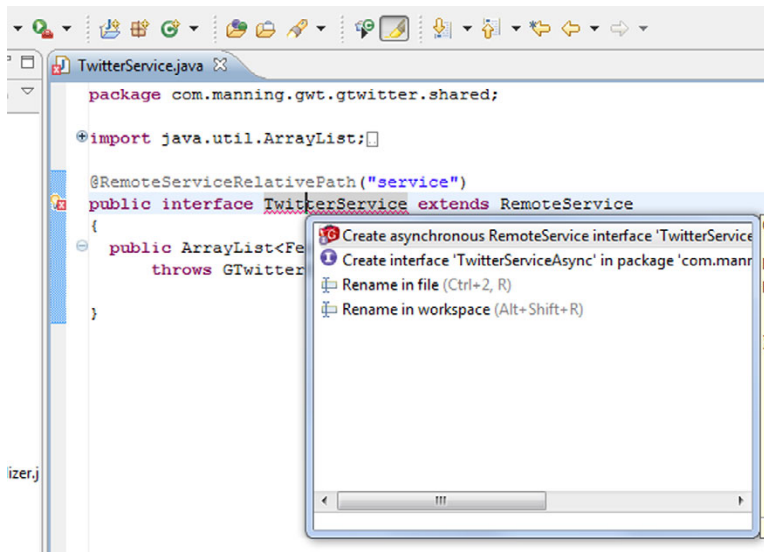


Figure 7.3 Using the quick-fix menu in Eclipse can save you typing by creating the asynchronous interface for you.

callback code shortly, but for now let's recap the set of steps to create the asynchronous interface:

- 1 Copy the service to a new file with the same name plus Async (for example, Foo to FooAsync).
- 2 Add an `AsyncCallback` parameter to each method, as the last parameter, using the method's return type as the generic's type (for example, `AsyncCallback<Foo>`).
- 3 Change the return type of all methods to `void`.
- 4 Remove any exceptions declared on methods in the service interface.

If you're using GPE, you can use an autofix function to create this interface for you and save yourself some typing as well as reduce the possibility of introducing a bug. Eclipse provides several ways to do this. One way is to open the service interface `TwitterService`, click the interface name, and press Ctrl+I (one, not L). This will open the quick-fix menu. Select the first option, `Create Asynchronous RemoteService Interface 'TwitterServiceAsync'`, as shown in figure 7.3.

At this point we've created everything except the client-side code that will make the call. Let's look at that now.

7.6.2 *Making the call to the server*

For simplicity's sake, we'll build an overly simple client in order to show off how to call a GWT-RPC service. Figure 7.4 shows what the final application looks like in the browser.

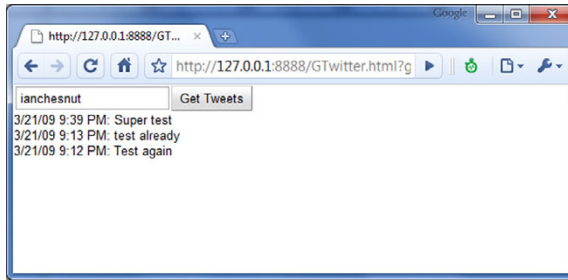


Figure 7.4 The finished GTwitter application running in the Google Chrome browser

As you can see, it's basic. It includes a `TextBox` for input, a `Button` for activation, and a `VerticalPanel` for output. The next listing shows the start of the application.

Listing 7.11 Basic structure for the GTwitter application's `EntryPoint`

```
package com.gwtia.ch07.client;

import java.util.ArrayList;

... imports omitted for brevity ...

public class GTwitter implements EntryPoint {
    private TextBox txtScreenName = new TextBox();
    private Button btnGetTweets = new Button("Get Tweets");
    private VerticalPanel tweetPanel = new VerticalPanel();

    public void onModuleLoad() {
        RootPanel.get().add(txtScreenName);
        RootPanel.get().add(btnGetTweets);
        RootPanel.get().add(tweetPanel);

        // final AsyncCallback<ArrayList<FeedData>>
        // updateTweetPanelCallback = ...

        // btnGetTweets.addClickHandler(new ClickHandler() { ... }
    }
}
```

This is a basic application, and the purpose is to explore GWT-RPC, so we won't provide any further explanation here. You need to add two things to complete the application, as noted by the commented lines in listing 7.11. First, you need to create an implementation of `AsyncCallback` that will receive the result from the server, and second, you need to add a `ClickHandler` to the `Button` to trigger the server call.

We'll tackle the `AsyncCallback` first. The following listing shows the code for this; it's added to the body of the `onModuleLoad` on the `EntryPoint`.

Listing 7.12 The `AsyncCallback` that will receive the GTwitter data from the server

```
final AsyncCallback<ArrayList<FeedData>> updateTweetPanelCallback
    = new AsyncCallback<ArrayList<FeedData>>() {
```

Anonymous
AsyncCallback
declaration

1

```

public void onFailure(Throwable e) {
    Window.alert("Error: " + e.getMessage());
}

public void onSuccess(ArrayList<FeedData> results) {
    tweetPanel.clear();
    for (FeedData status : results) {
        PredefinedFormat fmt = PredefinedFormat.TIME_SHORT;
        String dateStr =
            DateTimeFormat.getFormat(fmt).format(status.getCreatedAt());

        tweetPanel.add(new Label(dateStr + ": " + status.getText()));
    }
}
};

```

2 Handles exceptions

3 Handles receipt of result data

As you saw when we developed the asynchronous interface, all calls to the server require an `AsyncCallback` parameter, typed to the data type being returned from the server. Let's walk through listing 7.12 and examine the details.

① In this example you create the callback as an anonymous class. You could create a separate class for this, but an anonymous class works well for the purposes of this example. You also declare the variable as `final`, which isn't a requirement of GWT; it's a requirement of Java because of how you use this variable in the `ClickHandler` code that we'll look at soon. Notice that the type of your `AsyncCallback` matches the last parameter of the `getUserTimeline()` of the asynchronous interface.

② You need to implement two methods for the callback, the first of which is `onFailure`. This method will be called if the server throws a checked exception (in this case the `GTwitterException`), or if any number of other errors occur, for example, if the target service is unavailable. In our code we show the error in an alert box.

③ The second method you need to implement is `onSuccess`, which will receive the results from the server on a successful call. Notice that the parameter type is keyed on the type you used in the `AsyncCallback` declaration, in this case `ArrayList<FeedData>`. When you receive the callback, you clear the `VerticalPanel` used to display the results and fill it with the results from the server.

Now that you've defined the handler for the response, you can make the call to the server in the `ClickHandler` of your application's action button, as shown in the next listing.

Listing 7.13 Calling the GTwitter service from within a button `ClickHandler`

```

btnGetTweets.addClickHandler(new ClickHandler() {

    public void onClick(ClickEvent event) {
        TwitterServiceAsync service = GWT.create(TwitterService.class);

        service.getUserTimeline(txtScreenName.getText(),
            updateTweetPanelCallback);
    }
});

```

Create the async implementation 1

2 Call the remote method

In your handler, which is attached to your action button, you trigger the call to the server. Some explanation of what's going on here is in order.

❶ First, you create an instance of the `TwitterServiceAsync` interface by calling `GWT.create()`, passing the service interface class object as the parameter. The implementation returned is the class that will handle the serialization of the parameters, the call to the server, and the deserialization of the result.

Under the hood this is all driven by GWT's deferred-binding feature. Specifically there's a deferred binding that triggers code generation when the class passed to `GWT.create()` extends the `RemoteService` interface. When this is encountered by the GWT compiler, it kicks off a code generator that generates the asynchronous interface implementation as well as the serialization and deserialization classes to handle the calls defined in the interface. Although you don't need to understand deferred binding to use GWT-RPC, it does help with understanding how this works. Deferred binding is covered in chapter 19.

❷ You then make the call to `getUserTimeline`, passing your callback object as the second parameter, which starts a chain of events that will result in a call to the `onError` or `onSuccess` method of the `AsyncCallback`.

One thing to note is that the service returned by `GWT.create()` also implements `ServiceDefTarget`. This interface is used to alter how the generated service works, including setting the URL of the service. For example, here is an altered version of the server call that we already presented.

Listing 7.14 Calling the server service with an explicit server URL

```
import com.google.gwt.user.client.rpc.ServiceDefTarget;
...
TwitterServiceAsync service = GWT.create(TwitterService.class);
((ServiceDefTarget) service).setServiceEntryPoint(GWT.getModuleBaseURL()
    + "service");

service.getUserTimeline(txtScreenName.getText(),
    updateTweetPanelCallback);
}
```

The only thing new here is that you cast the service to `ServiceDefTarget` and call the `setServiceEntryPoint` method, passing the URL to the servlet that will handle the request. This can be used instead of, or to override, the `@RemoteServiceRelativePath` annotation that we put in our service in section 7.5.

In addition, you can also use this interface to specify a custom `RequestBuilder`, a lower-level API used to make the call to the server. This allows you to make changes to how requests are made, including setting custom header values and credentials. We'll discuss `RequestBuilder` in chapter 12.

If you've been following along, you now have a finished application. It's time to launch it and take it for a test drive. We expect that everything went well and it works out of the box, but if not, this next section is for you.

7.7 Debugging GWT-RPC

There are a lot of ways to debug your GWT application. Speed Tracer,⁷ a plug-in for Chrome, can provide you information on the headers sent between client and server as well as execution times. Wireshark,⁸ an open source packet sniffer, will show you all of the network traffic, allowing you to inspect interaction RPC calls over the network. Eclipse can execute your application in debug mode, allowing you to step through the application line by line.

All of them are great at what they do, but Firebug, a Firefox add-on, possibly provides the most benefit for debugging GWT-RPC calls. You can learn about and download Firebug from <http://getfirebug.com/>.

The Firebug website has a great video to introduce you to the tool, but this isn't a book about Firebug, so we'll only cover the bits of Firebug that apply to GWT-RPC. Figure 7.5 shows Firefox with the Firebug panel opened.

With regard to accessing network resources, Firebug will show you the URLs accessed by your browser. This is useful if you want to verify that your client-side code is hitting the correct URL on the server.

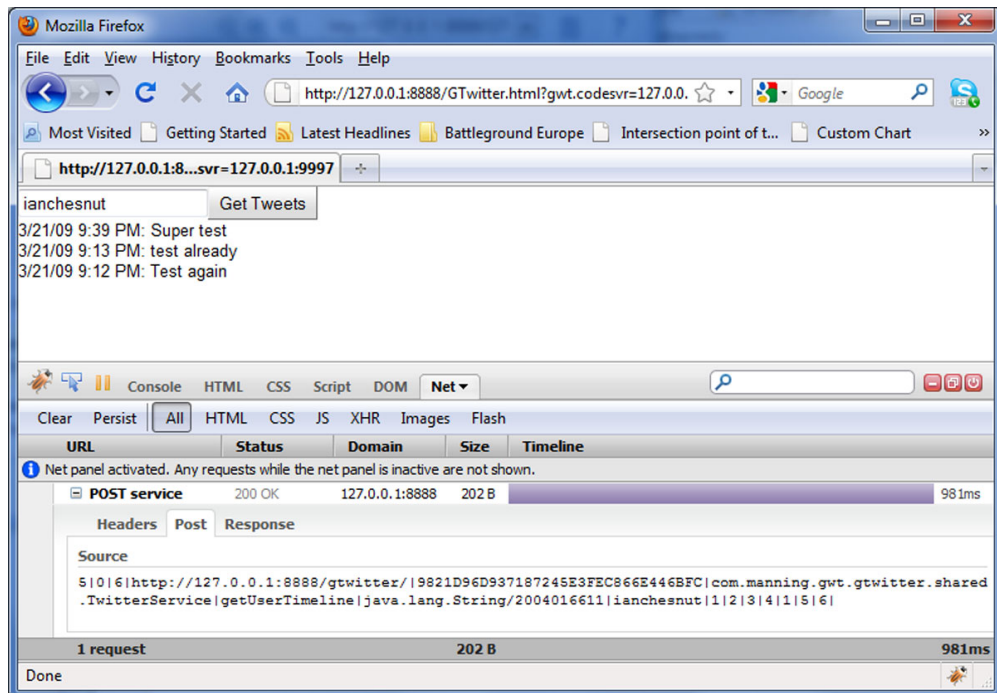


Figure 7.5 The finished GTwitter application as it runs in Firefox, with the Firebug panel opened

⁷ Speed Tracer can be found at <http://code.google.com/webtoolkit/speedtracer/>.

⁸ Wireshark is available at www.wireshark.org/.

When browsing a single request from the browser, you can then view the request/response headers and the content passed. Assuming the service URL is correct and hit the server, the most useful information is the contents of the request and reply. It's important to understand that the format of the serialization of GWT is unpublished and changes from time to time, but it's still fairly human readable. For example, it's easy to pick out the method name that was called by the client and the general contents of the reply.

So if you need to ask the question, "Am I hitting the right URL?," "Am I passing the data I think I am to the server?," or "Is the server sending me what I think it is?," then Firebug is a good tool for the job.

Next up is something that we don't talk too much about in this book, and that's security. We've opted to not make this a book about security because it's a deep topic, one that we hope you'll explore thoroughly, but in this case GWT provides some explicit support that's worth sharing.

7.8 Securing GWT-RPC against XSRF attacks

XSRF (or CSRF) is short for cross-site request forgery, and it's an attack that could allow an attacker access to your web mail, your social networking account, or even your bank account. If you haven't come across this term before, we suggest that you do some additional research, but let's see if we can describe it briefly.

7.8.1 Understanding XSRF attacks

To help you better understand how an XSRF attack works, let's examine a hypothetical situation. Pretend you're a high-ranking executive for company X-Ray Alpha Delta, and you're logged in to the top-secret extranet application doing some product research. Once you log in to the top-secret application, it keeps track of who you are by giving you a web cookie. Using cookies as a way of handling user sessions is a common tool used by most secured web applications.

As you're working on the top-secret extranet, you receive an email prompting you to review some competitor content on the internet. You click the link and start reading the page, which seems to be a legitimate news site. Unknown to you, the "news" site is running JavaScript in your browser and is making requests against your top-secret extranet.

This is possible because your browser automatically passes your session ID contained in a cookie to the top-secret extranet server, even though the JavaScript calling the server originated from the "news" site. Now understand that this isn't a bug in your browser; it's the way browsers work. The XSRF attack is taking advantage of how browsers work, which is why this type of attack is so problematic.

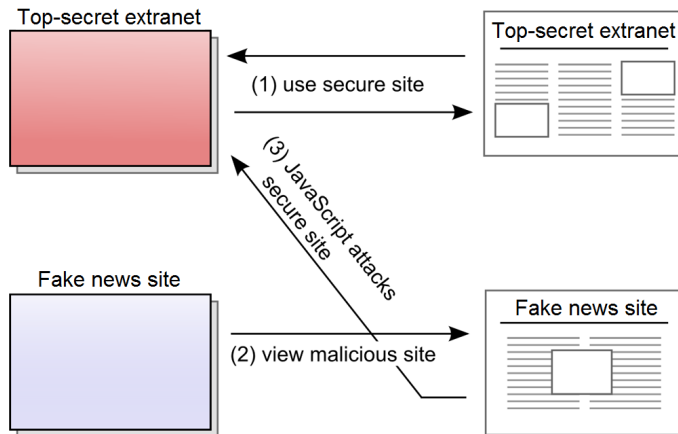


Figure 7.6
XSRF attack, using
JavaScript to break into a
“secure” application

Figure 7.6 shows the order of events in such an attack, allowing the malicious JavaScript access to the “protected” site.

The one advantage that you do have is that the attacker is making blind calls to the server. The attacker can trigger a call to a secured web page but won’t be able to read the result. So an attacker couldn’t use an XSRF attack to read an email on your web mail site, but it could call a method on the server that the attacker knows will change your password.

It’s the fact that these attacks are blind that allows us to prevent them.

7.8.2 Adding XSRF protection to your RPC calls

Because XSRF takes advantage of your browser automatically sending your session ID to the server, you can’t rely on that alone. What you need is a second key, but one that your browser won’t automatically send.

The mechanism that GWT-RPC can be enabled to use is to make an additional call to the server to get a second secret key and then have your normal server calls pass this extra key in the request. Now it’s true that someone could trigger a XSRF attack to call the server service that creates the secret key, but as we said earlier an attacker can’t read the result of the call, so they won’t be able to gain access to this key.

Warning: this code won’t work!

GWT-RPC’s XSRF protection requires that the browser have the ability to pass a cookie value to the server, and for real protection this needs to be tied somehow to the application authentication. In almost all Java deployment scenarios, this cookie name is JSESSIONID. In a typical application, the JSESSIONID cookie value allows the application server to tie the user to a Java `HttpSession` in memory, which in turn has the user’s authentication information. A problem, however, is that when you run the example code in Eclipse without any login mechanism, the Jetty server doesn’t create

(continued)

an `HttpSession` or pass a `JSESSIONID` to the server. To counter this, our example code in the source download includes a `ForceSessionCreationFilter` that will make sure that a `JSESSIONID` is issued to the client browser. Please review the source code download for details on how we did this.

To implement the protection, you have to make changes to three parts, and all of the changes are relatively minor. In the sections that follow we'll show you how to implement the change on the server, on the service interface, and in the call you make to the server.

ENABLING XSRF PROTECTION ON THE SERVER

Enabling XSRF protection on the server side is as simple as swapping out `RemoteServiceServlet` and replacing it with `XsrfProtectedServiceServlet`. Back in section 7.5.3 you might recall that we implemented the class `TwitterServiceImpl`, which is the server-side servlet that's the target of GWT-RPC calls from the client. In the example we had that class extend `RemoteServiceServlet`, which provides the glue that connects incoming GWT-RPC calls to our class. The class `XsrfProtectedServiceServlet` is a drop-in replacement for `RemoteServiceServlet`, so all you need to do is change the class declaration, as shown in the following listing.

Listing 7.15 The GTwitter server-side implementation, now supporting XSRF protection

```
package com.manning.gwtia.ch07.server;

import com.google.gwt.user.server.rpc.XsrfProtectedServiceServlet;
...other imports omitted...

public class TwitterServiceImpl extends XsrfProtectedServiceServlet
    implements TwitterService
{
    ...implementation omitted...
}
```

As you can see, it's that simple; have your server-side code extend `XsrfProtectedServiceServlet`, and you're finished with this class.

But you do need to make some changes to the deployment descriptor. The next listing provides an updated version of our project's `web.xml` file, annotated to show the new additions.

Listing 7.16 An updated web.xml with additions for supporting XSRF protection

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

```

<context-param>
  <param-name>gwt.xsrf.session_cookie_name</param-name>
  <param-value>JSESSIONID</param-value>
</context-param>

<servlet>
  <servlet-name>twitterServlet</servlet-name>
  <servlet-class>
    com.manning.gwtia.ch07.server.TwitterServiceImpl
  </servlet-class>
</servlet>

<servlet>
  <servlet-name>xsrf</servlet-name>
  <servlet-class>
    com.google.gwt.user.server.rpc.XsrfTokenServiceServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>twitterServlet</servlet-name>
  <url-pattern>/gwtia_ch07_gwtrpc/service</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>xsrf</servlet-name>
  <url-pattern>/gwtia_ch07_gwtrpc/xsrf</url-pattern>
</servlet-mapping>
</web-app>

```

1 Specify session cookie name

2 Add XSRF token service

3 Map XSRF service

The first thing you must add to the web descriptor is the name of the cookie that holds the key to the user's authentication information on the server **1**. In most cases this will be `JSESSIONID`, which is the standard for Java servlet containers.

To understand how this works, let's connect the dots. Let's assume that you use Spring Security to serve all of your services on the server. Spring Security does this by using a servlet filter that analyzes each URL that the browser is requesting, determining what the user does and doesn't have access to. Spring Security, like most Java applications, stores the authentication information in the in-memory `HttpSession` instance for that user. And the servlet container (such as Jetty or Tomcat) associates the `HttpSession` with the individual request.

Running that in the reverse order, that means the browser sends a `JSESSIONID` cookie value to the server, which then uses that as a key that allows it to associate the request with an `HttpSession`, which in turn is used by Spring Security to ensure the user is allowed to access the requested resource. Simple, right?

The next thing you need to do is configure the `XsrfTokenServiceServlet` **2** using the `<servlet>` tag and map it to some path **3** using a `<servlet-mapping>`. The `XsrfTokenServiceServlet` is a GWT-RPC service that will allow you to request a token from the server. More on that in a bit. All you need to know at this point is that you've installed the service and mapped it to the path `/gwtia_ch07_gwtrpc/xsrf`.

At this point you can run your application and it should work fine, but realize that you haven't protected anything yet. You need to do that on the service interface.

ENABLING XSRF PROTECTION ON THE SERVICE INTERFACE

In our GTwitter example we created an interface called `TwitterService`, which extended the `RemoteService` interface. You can force XSRF in two ways; the first is to secure individual methods by using annotations, and the second is to require XSRF protection for all methods by changing the interface that your service extends. The second way is the easiest and in many cases is the right thing to do, so we'll visit that one first.

The following listing presents an updated version of `TwitterService`, which now forces XSRF protection on all methods.

Listing 7.17 An updated `TwitterService` with XSRF protection

```
package com.manning.gwtia.ch07.shared;

import com.google.gwt.user.client.rpc.XsrfProtectedService;
...other imports omitted...

@RemoteServiceRelativePath("service")
public interface TwitterService extends XsrfProtectedService
{
    ...implementation omitted...
}
```

The change in the previous listing, as with the server implementation, is subtle. The only change made to the interface is to change the interface that you extend. Where the version without XSRF protection implements `RemoteService`, it now implements `XsrfProtectedService`. By having your service interface extend `XsrfProtectedService`, you're now requiring that all method calls include a token, which acts as a key. If the token passed along with the server call is valid, the server will accept it; if it's invalid, it's rejected.

Before we show you how the tokens work, let's first concentrate on what we have here. As we mentioned, all methods are now secured, but what if for some reason XSRF protection is enforced only for some methods while leaving others unsecured? For that you can use method-level annotations.

For those rare occasions when you need to protect only some of the methods in your interface, you can extend `XsrfProtectedService` and turn off protection on individual methods by using the annotation `@NoXsrfProtect`. Alternatively, you can go the opposite way and have your service interface extend `RemoteService`, which provides no protection, and then mark methods that should be protected with `@XsrfProtect`.

Now that the methods are protected, you'll find that the GTwitter application no longer works. Each GWT-RPC request will come back with the error message "Invalid RPC token (XSRF token missing)." If you get this, it means that your methods are now secured. Next, let's look at how to change the GWT-RPC call from the client side.

ADDING XSRF PROTECTION IN YOUR CLIENT-SIDE RPC CALLS

XSRF protection changes how you make calls from the server. Where you were making only a single call, you now need to make two. The first call to the server is to fetch a token. The default implementation takes the cookie value passed to the server and creates an MD5 hash from it, returning the result to the client. The client then uses this token as a key for calling the target GWT-RPC method.

The next listing shows the changes to the client-side call from our GTwitter application. Following the listing we discuss what's going on in the code.

Listing 7.18 Updated GTwitter client using XSRF token to unlock the service method

```
XsrfTokenServiceAsync xsrf = GWT.create(XsrfTokenService.class);

((ServiceDefTarget)xsrf).setServiceEntryPoint(
    GWT.getModuleBaseURL() + "xsrf");

xsrf.getXsrfToken(new AsyncCallback<XsrfToken>() {

    public void onSuccess (XsrfToken token)
    {
        TwitterServiceAsync service = GWT.create(TwitterService.class);
        ((HasRpcToken) service).setRpcToken(token);

        service.getUserTimeline(
            txtScreenName.getText(), updateTweetPanelCallback);
    }

    public void onFailure (Throwable caught)
    {
        try {
            throw caught;
        }
        catch (RpcTokenException e) {
            Window.alert("Error: " + e.getMessage());
        }
        catch (Throwable e) {
            Window.alert("Error: " + e.getMessage());
        }
    }
});
```

The diagram illustrates the sequence of operations in the code listing:

- 1 Create token service:** Points to the line `GWT.create(XsrfTokenService.class)`.
- 2 Call token service:** Points to the line `xsrf.getXsrfToken(new AsyncCallback<XsrfToken>() {`.
- 3 Set token:** Points to the line `((HasRpcToken) service).setRpcToken(token);`.

For brevity, listing 7.18 includes only the code that's changed. If you want to review the rest of the class, you should refer back to section 7.6.2. Listing 7.18 should look familiar, because it's a GWT-RPC call. What's different is that you nest the original call inside the `onSuccess()` method. That's useful to understand, but let's talk specifics.

In order to get a token, you use `GWT.create()` to get a service for `XsrfTokenService` ❶ and set the end point, which in this case is the module base URL plus `/xsrf`. This matches the mapping that you added to the `web.xml` for the `XsrfTokenServiceServlet`.

You then use the service to call the method `getNewXsrfToken()` ❷. The purpose of this method is to call the server, which will then return a token to the client.

In the `onSuccess()` method of the callback, you use the token as a parameter to `setRpcToken()` on your `TwitterService` method call ❸. The `setRpcToken()` method is made available by casting the `TwitterService` asynchronous implementation to `HasRpcToken`. You then call your service method as usual, but now it will include the token that will be used like a key to unlock the service.

At this point we want to point out that this code only works if your browser passes a `JSESSIONID` cookie to the server, which is then turned into the token. If you see errors on the server saying “Session cookie is not set or empty,” go back to the beginning of section 7.8.2 and review the sidebar titled “Warning: this code won’t work!”

As an alternative for the purposes of testing, you could also add this code snippet to your application, which generates a random value and sets the `JSESSIONID` cookie:

```
if (Cookies.getCookie("JSESSIONID") == null)
    Cookies.setCookie("JSESSIONID", Double.toString(Math.random()));
```

This isn’t ideal to have in production code, because it’s better to be able to tie the session ID to the user’s authentication, but it works great for testing this example. For more information on GWT and security, we recommend that you visit the security section of the online documents for GWT. It’s fairly comprehensive and goes beyond what we discuss in this book.

With that, we need to wrap up our tour of GWT-RPC. So let’s review what we covered in this chapter.

7.9 Summary

In the last few dozen pages we showed you everything you need to know about GWT-RPC, but it was a long journey. You might be asking how this helps you simplify your GWT applications over other remoting methods. We could take a mother’s approach and say that you should take your medicine because it’s good for you. But we’re all big boys and girls now and can make our own decisions.

The biggest benefit of using GWT is that you can use Java objects throughout, sharing data objects on both the client and server and reducing the code you need to write because all the serialization is handled for you. But you will pay a price for less code, and it’s in complexity. In order for GWT to generate the serialization code for you, it needs you to provide enough information so that it can do its job. This could be as simple as tacking on the `IsSerializable` interface to your DTOs, or it could require you to create custom serializers.

As programmers we know that the best way to deal with complexity is practice. The more you practice, the easier it gets, and over time it becomes habitual, requiring little thought to accomplish. We never said that GWT-RPC is the right solution for every problem. That’s an architectural decision that you’ll need to make on each and every project.

Getting beyond the reasons for using GWT-RPC, in this chapter we reviewed the four components of GWT-RPC that you need to build yourself: the model, the servlet, the service interface, and the asynchronous service interface. We provided tips on debugging and showed you how to handle serialization of JPA and JDO entities, how to deploy to your GWT-RPC servlet, how to lay out your project, and much more.

Next, we'll look at another tool in GWT's repertoire for managing entities: RequestFactory.

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

“Covers all the newest features—a must-read for any GWT professional.”

—Michael Moossen
Allesklar.com AG

“Clear, practical, efficient ... *in action.*”

—Olivier Nougier
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