GWT IN ACTION

Easy Ajax with the Google Web Toolkit

SAMPLE CHAPTER

Robert Hanson
Adam Tacy

MANNING

GWT in Action

by Robert Hanson
and Adam Tacy

Chapter 10

brief contents

Part 4

Advanced techniques

Part 2 explored the user-interface components of GWT, explaining how to create custom widgets and bundle them as a reusable library. Part 3 takes you to the next step by looking at GWT's advanced toolset for making remote procedure calls, code generators, application configuration, and internationalization tools.

10 Communicating with GWT-RPC

This chapter covers

- Asynchronous communication
- Overview of the GWT-RPC mechanism
- Step-by-step instructions for using GWT-RPC
- Building an example widget using GWT-RPC

When you're building a rich Internet application, it's likely you won't get too far before you need to contact the server. The reasons for doing so are numerous and can range from updating the contents of a shopping cart to sending a chat message. In this chapter, we'll explore the primary remote procedure call (RPC) mechanism that ships with the GWT toolkit. Throughout this chapter, and the chapters that follow, we'll refer to this mechanism as GWT-RPC to distinguish it from other general RPC flavors.

If you're unfamiliar with RPC, it's a term used to describe a mechanism that allows a program to execute a program on another computer and return the results of the calculation. This is a simplistic view of RPC, but this is essentially what you'll be doing in this chapter.

Throughout this chapter, as well as the next, you'll learn by building and extending an example component. This component, once completed, periodically requests performance data from the server and displays the values to the user. We call this example component the "Server Status" component.

To make the task of writing RPC code as intuitive as possible, this chapter follows a strict organization. In the first section, we'll define the component you're going to build, including a basic UI layout and defining the data that will be displayed in the component. In that context, we'll discuss asynchronous communication and some security restrictions of browser-based RPC.

In the second section, we'll examine all the nuts and bolts of GWT-RPC. We'll define the data object that will be passed on demand between the client and server, and the serialization of data objects. We'll then get down to business and write the code for the component from beginning to end.

At the end of the chapter, we'll wrap up the discussion with a detailed overview of the project and a review of the core participants of the GWT-RPC mechanism. But it doesn't end there; chapter 11 extends the example component by applying software patterns and polling techniques, providing for reusable and maintainable code.

Without further delay, let's begin the journey by defining the Server Status example project and examining the fundamental concepts behind the GWT-RPC mechanism.

10.1 Underlying RPC concepts

In this section, we'll explain how the GWT-RPC mechanism works by building a sample component. We wanted the component to be of interest to the widest audience possible, so we chose to create what we call the Server Status component. The

purpose of the component is to provide up-to-date memory and thread usage for the Java Virtual Machine (JVM). As you go through the process of building the component, think about what other information you may want to add, like perhaps the number of logged-in users or maybe disk usage. Once it's completed, you'll be able to use this component in the GWT Dashboard project introduced in chapter 3 or in any of your own GWT projects.

You need to assemble three pieces of the puzzle to have a working RPC application: the service that runs on the server, the client in the browser that calls the service, and the data objects that are transported between the client and the server. Both the server and the client have the ability to serialize and deserialize data so the data objects can be passed between the two as ordinary text.

Figure 10.1 provides a visual representation of what the completed Server Status component will look like, along with an example service request and response. The connection between client and server in figure 10.1 is initiated by the client and passed through a proxy object provided by GWT. The proxy object then serializes the request as a text stream and sends it to the server. On the server, the request is received by a special Java servlet provided by GWT. The servlet then deserializes the request and delegates the request to your service. Once your service returns a value to the GWT servlet, the resulting object is serialized and sent back to the client. On the client side, the response is received by the GWT proxy, which deserializes the data back into a Java object and returns the object to the calling code.

Figure 10.1 The completed GWT-RPC based Server Status component receiving serialized data from the server. The three pieces of the RPC application include the service that runs on the server, the client in the browser that calls the service, and the data objects that are transported between the client and the server.

The important part to remember about the round trip of the request is that the code deals with ordinary Java objects, and GWT handles serialization and deserialization of the data automatically. The example transaction in figure 10.1 shows the request and response data that is passed between the client and server; in practice you'll never deal with this data directly, although it may sometimes be helpful to view the data when debugging a problem. In this book, we won't explain the serialization scheme; it isn't a documented part of GWT and will likely change in later GWT versions as performance enhancements are introduced.

The idea of GWT doing most of the work for you sounds great, but the devil is in the details. You must understand a few things before you start coding, the first of which is the asynchronous nature of browser-based communication.

10.1.1 Understanding asynchronous communication

Calling a remote service with GWT-RPC is just like calling a local method with a couple additional lines of code. There is one caveat: The call is made in an asynchronous manner. This means that once you call the method on the remote service, your code continues to execute without waiting for a return value. Figure 10.2 shows this graphically; a gap of time exists between the call to the server and the response.

Asynchronous communication works much like an event handler in that you provide a callback routine that is executed when the event occurs. In this case, the event is the return of the call to the service. If you haven't dealt with this sort of behavior before, it may feel foreign at first, and it can require some additional planning when building applications. GWT uses this type of communication due to the way the underlying `XMLHttpRequest` object works. The reason why the `XMLHttpRequest` object behaves this way is beyond the scope of the book, but in

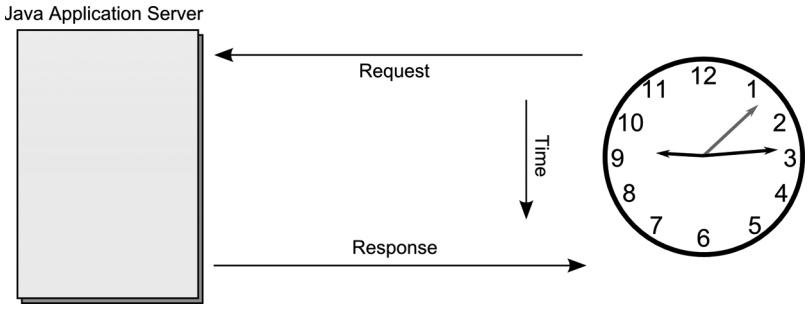

Figure 10.2 A visual representation of the time-delay of asynchronous communication used in browser-based communication

part it's due to the nature of JavaScript implementations. This asynchronous nature, though, has some advantages.

The first advantage is that network latency and long-running services can slow down communication. By allowing the call to happen asynchronously, the call won't hold up execution of the application waiting for a response; plus it feels less like a web application and more like a desktop application. The other benefit is that you can use some tricks with this to emulate a *server-push*.

DEFINITION *Server-push* is a mechanism where the server pushes data out to the client without it being requested. This is used in applications like chat, where the server needs to push messages out to its clients. We'll look at how to emulate server-push along with some polling techniques in chapter 11.

Besides asynchronous communication, another RPC issue we need to deal with is security, which affects how you use any browser-based RPC mechanism.

10.1.2 Restrictions for communicating with remote servers

Security is always a concern on the Internet, especially when a call can be made to a server that potentially returns sensitive data to the client, or perhaps provides access to secure systems. We have no intention of conducting a complete examination of security for web servers, which we leave to the experts. We do, though, want to explain one feature of making remote calls from the browser, which may seem like more of an annoyance than a feature. When you make a remote call from the browser, it *must* make the call to the same server from where the JavaScript code originated.

This means your GWT application running in the browser, which was loaded from your web server, can't call services hosted by other sites. For some people, this is more than an annoyance; it's a problem that means they may not be able to deploy their code in the manner they intend. Before we explain why this is truly a feature, understand that there are ways around it by providing a proxy on your server that calls the remote server. In chapter 13, you'll use such a proxy to communicate with a third-party search service.

Note that some browsers may let you override this behavior. For example, with Internet Explorer, you can tell the browser to allow the remote connection to proceed, even though it's attempting to contact a foreign server. Requiring your users to bypass restrictions like this is never a good idea, and it can make your users vulnerable to attacks. This is a strong statement; but as you'll see shortly with cross-site request forgery, there is a good reason for this.

To help you better understand how an attack like this might play out, 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. The connection between the client and server is an encrypted connection provided by SSL. Using cookies as a way of handling user sessions and SSL are common tools used by most secure web applications.

While 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 information contained in a cookie to the top-secret extranet server, even though the JavaScript calling the server originated from the "news" site. Figure 10.3 shows the order of events in such an attack, allowing the malicious JavaScript access to the "protected" site.

This scenario is plausible and has been proven to work when the web browser allows JavaScript to call foreign servers. At the Black Hat convention in 2006, the security firm iSEC Partners provided details about how they used this technique to remove $5,000 from a stock account. The user was logged in to a financial site and then viewed a foreign site, which contained the malicious JavaScript code. The JavaScript code in question was contained in five separate hidden iframes. Each script ran in turn, making one call to the financial service. The scripts changed the user's email notification settings, added a new checking account for transfer-

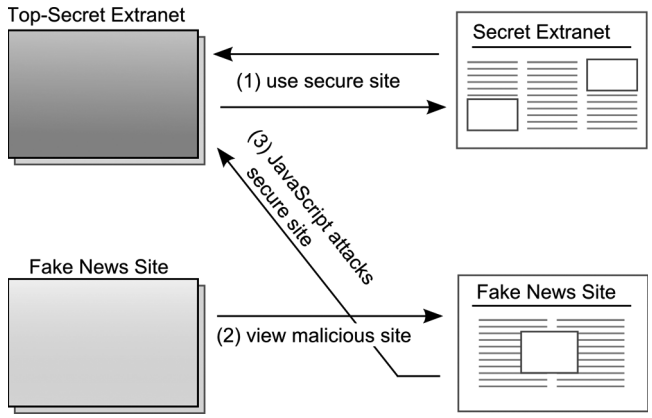

Figure 10.3
Cross-site scripting attack, using JavaScript to break in to a "secure" application

ring funds, transferred $5,000 out of the account, deleted the checking account, and restored the email notification settings. All of this occurred while the user was viewing the malicious site. This is a scary scenario, and it's why browsers don't typically allow JavaScript code to contact foreign hosts.

We've gotten far off track and need to get back to where we started: an overview of RPC architecture of GWT. So far, we've discussed its asynchronous nature and automatic data serialization, and addressed why the service must be provided by the same host that served the GWT application (for critical security reasons). To get back on track, you'll create a new GWT project that will be used to house the Server Status component.

10.1.3 Creating the Server Status project

By now, you know how to set up a new GWT project, but this one will be a little different. When you create a GWT project that will perform RPC, you need to account for the fact that it will contain both server-side and client-side code. The GWT compiler should compile only the client-side code to JavaScript without including the server-side portion of the code. You'll be including code for both the client and server in this project, so you need to do a few extra things to inform the GWT compiler which source files it needs to compile.

The first step is to use the `projectCreator` and/or `applicationCreator` command-line tool to create a new project. If you need a refresher on how to do this, consult chapter 2. For our purposes, this chapter assumes you already know how to do this. The following command and subsequent output creates the project:

```
applicationCreator -out ServerStatus org.gwtbook.client.ServerStatus

Created directory ServerStatus\src
Created directory ServerStatus\src\org\gwtbook
Created directory ServerStatus\src\org\gwtbook\client
Created directory ServerStatus\src\org\gwtbook\public
Created file ServerStatus\src\org\gwtbook\ServerStatus.gwt.xml
Created file ServerStatus\src\org\gwtbook\public\ServerStatus.html
Created file ServerStatus\src\org\gwtbook\client\ServerStatus.java
Created file ServerStatus\ServerStatus-shell.cmd
Created file ServerStatus\ServerStatus-compile.cmd
```

Next, you need to remove the sample Java code provided by the `applicationCreator` tool. The following code is ServerStatus.java after removing the sample application:

```
package org.gwtbook.client;

import com.google.gwt.core.client.EntryPoint;

public class ServerStatus implements EntryPoint
{
  public void onModuleLoad ()
  {
    // code
  }
}
```

You also want to start with a new HTML page. Listing 10.1 shows the ServerStatus.html page in the project. You can remove the various comments, provide a better page title, and add an empty style block. Once you finish the component, we'll provide some style code you can use to make the component look like figure 10.1.

Listing 10.1 The minimal HTML page you'll use to host the Server Status project

```
<html>
  <head>
    <title>Server Status</title>

    <!-- used to load module in GWT versions through 1.3 -->
    <meta name='gwt:module' content='org.gwtbook.ServerStatus'>

    <!-- used to load module in GWT versions 1.4+ -->
     <script language='javascript'
       src='org.gwtbook.ServerStatus.nocache.js'></script>

    <style type="text/css">
    </style>
  </head>
  <body>

    <!-- used to load module in GWT versions through 1.3 -->
    <script language="javascript" src="gwt.js"></script>
  </body>
</html>
```

NOTE Because GWT is thriving, it's subject to regular improvements. In listing 10.1, we've inserted HTML comments to identify the lines that are required to load the Server Status module in the current 1.3 release of GWT as well as the proposed loading method that will be used in GWT version 1.4. The older module-loading method, using `gwt.js`, will still work in GWT version 1.4, but it has been deprecated.

The CSS styles in listing 10.2 style the Server Status component to look like the example look and feel provided in figure 10.1. Feel free to adjust the styles to your liking or use your own. You can place the following CSS code into the `<style>` element in the HTML page or put the CSS code into an external file and reference it with the HTML `<link>` element.

Listing 10.2 A CSS file for styling the Server Status component

```
.server-status {
  width: 200px;
  height: 200px;
  border: 1px solid black;
}

.server-status td {
  font-family: Arial;
  font-size: 12px;
}

.server-status .title-bar {
  text-align: center;
  background: #666;
  padding: 2px 0;
  color: white;
  font-weight: bold;
}

.server-status .stats-grid {
  width: 200px;
}

.server-status .stats-grid td {
  border-bottom: 1px solid #ccc;
}

.server-status .stat-name {
  font-weight: bold;
}

.server-status .stat-value {
  text-align: right;
}

.server-status .last-updated,
 .server-status .update-button {
  font-size: 10px;
  margin: 0 5px;
}
```

Set component width and height

Set font style

Set title-bar styles

Set inner data-grid width

Add row lines to data-grid

Set statistic title styles

Right justify data values

Set styles of status label and update button

Now that you have a new project to work with, you can get down to writing some code. In the next section, you begin by creating a data object that will be passed from the server to the client and then get into creating the service and calling it from the client.

10.2 Implementing GWT-RPC

As we stated at the beginning of this chapter, there are three parts to the GWT-RPC mechanism, as you can see in figure 10.4. The first is the service that runs on the server as a servlet, the second is the web browser that acts as a client and calls the service, and last are the data objects that pass between the client and server.

We'll start with the last of these, the data objects, and explain what types of objects GWT can serialize for you. Following this, we'll look at the server side of the threesome and how you implement the service on the server. Finally, you'll call the service from your browser.

During the course of the discussion, we'll reference the Server Status project that you started in the previous section. We'll also provide code examples not related to the Server Status project when doing so helps explain details of the GWT-RPC mechanism that aren't explicitly used by the Server Status component. By the end of this section, you'll have completed the Server Status component, and you'll be able to reuse it with any of your own GWT projects. Now, let's look at the serializable data objects.

10.2.1 Understanding serializable data objects

We need to begin our discussion of GWT-RPC with data objects because the data is what gives GWT-RPC life. Describing the functionality of GWT-RPC without data would be akin to describing the function of the human heart without first understanding the purpose of life-giving blood.

At the beginning of section 10.1, we mentioned that with the GWT-RPC mechanism, you can call methods that are executed on the server, and a resulting value

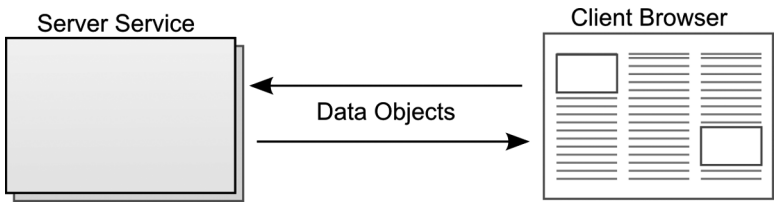

Figure 10.4 The three parts of GWT-RPC: client, server, and data objects

is passed back to the client. Just like any Java method, you may pass arguments to the method, which may be a primitive like an `int`, an object like a `String`, or an array of values. The list of value types that GWT can serialize, though, is finite:

- *Java primitive types*—`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`
- *Java primitive wrapper types*—`Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`
- *Subset of JRE objects*—Only `ArrayList`, `Date`, `HashMap`, `HashSet`, `String`, `Vector` (future versions of GWT may add to this)
- *User-defined classes*—Any class that implements `IsSerializable`
- *Arrays*—An array of any of the serializable types

CHANGES IN GWT 1.4

Added in GWT 1.4 is the ability to have your GWT serializable classes implement the java.io.Serializable interface instead of the GWT-specific IsSerializable interface. The change is being introduced to make it easier to share data objects with server-side persistence frameworks like Hibernate, which require data objects to implement java.io.Serializable. It's important to note that this change in GWT 1.4 is only intended to make it easier to integrate with persistence frameworks; it doesn't imply that GWT serialization follows any of the semantics of java.io.Serializable.

The first two groups of value types—primitives and their wrapper counterparts—are self explanatory, and all are supported. Only a limited number of Java data types are supported. In chapter 1, we discussed the JRE Emulation Library provided by GWT, and how GWT allows only certain Java classes to be used in client-side code. The list of supported Java classes here includes all of the value object types from the emulation library.

The first three groups consist of types that are part of standard Java, but what about user-defined types? The `IsSerializable` interface answers this question.

Implementing the IsSerializable Interface

The second from the last of the list of serializable types is any class that implements the `IsSerializable` interface. This interface is part of the GWT library, and it's used to signify that a class may be serialized. This serves a similar purpose as Java's own `java.io.Serializable` interface but is specific to GWT applications. Most of this section will be about using the `IsSerializable` interface.

The `IsSerializable` interface has no methods. It's only used to let the GWT compiler know that this object may be serialized, and it implies that you created the class with serialization rules in mind:

- The class implements `com.google.gwt.user.client.rpc.IsSerializable`.
- All non-transient fields in the class are serializable.
- The class has a zero-argument constructor.

By *non-transient field* we mean any field not using the `transient` modifier. GWT also won't serialize fields that have been marked as final; but don't rely on this, and be sure to mark all final fields as transient. That isn't to say that bad things will happen if you don't mark the final fields as transient, but you want the intention of the code to be clear if you or someone else ever needs to revisit the code for maintenance:

```
private transient String doNotCopy = "some value";
```

GWT serialization also traverses relationships between parent and child classes. You could have a superclass that implements `IsSerializable` and a subclass that doesn't; but, because the superclass is serializable, so are all of its subclasses:

```
public class Person implements IsSerializable {
  String name;
  Date birthday;
}

public class Programmer extends Person {
  String favoriteLanguage;
}
```

Another thing to consider when working with serializable data objects is that they will be used by both the client and server code, and, therefore, must adhere to the rules for client-side code. This includes being compliant with the Java 1.4 language syntax and may only reference classes that are part of the JRE Emulation Library or are user created classes.

In terms of optimizations for the GWT compiler, it's good to be as specific as possible when specifying the types of your fields in the data object. For example, it's common practice to specify `java.util.List` as a type instead of either `ArrayList` or `Vector`. The benefit of using a generalized type is that it allows you to change the underlying implementation without changing the type declaration. The problem is that when you generalize the type, it's harder for the GWT compiler to optimize the code, and you often end up with larger JavaScript files. The rule of thumb it to try to be as specific as possible in your typing.

You may have noticed the catch-22 situation we've run into. You're limited to the Java 1.4 syntax, which rules out using generics. If you're to be as specific as possible, you need a way to let the GWT compiler know what types of objects are contained in an `ArrayList` or `Vector`. This leads us to the `typeArgs` annotation.

Using the typeArgs annotation

Specific typing isn't possible when your data object has a member that is a collection of objects. Collections in Java 1.4 hold values of type `java.lang.Object`, and this is as generic of a type you can get. This would be an issue if, for example, your data class had a member type of `ArrayList`, `Vector`, `HashSet`, or any other type that implements `java.util.Collection`.

In the following example, the GWT compiler has no way of knowing what types of objects are held by either `listOfNames` or `listOfDates`, so it won't be able to properly optimize the client-side JavaScript:

```
private ArrayList listOfNames;
private Vector listOfDates;
```

GWT provides an annotation that allows you to let the compiler know what types of objects are in a collection. This isn't a Java 5 annotation, so it isn't part of the Java language; instead, you provide the annotation inside a Java comment. Figure 10.5 shows the syntax of the `typeArgs` annotation.

Figure 10.5 The `typeArgs` annotation is specified in a Java comment preceding a field in the class to provide a hint to the GWT compiler about the contents of a `java.util.Collection`.

There are two variations of this annotation, the second of which will be described later in this chapter when you define the service interface. In this first variation, the only parameter is the contained object type inside angled brackets. The contained type is specified using its full package and class name. When you apply this to the two fields `listOfNames` and `listOfDates`, it looks like this:

```
/**
 * @gwt.typeArgs <java.lang.String>
 */
private ArrayList listOfNames;

/**
 * @gwt.typeArgs <java.util.Date>
 */
private Vector listOfDates;
```

Now that you understand the basics, let's apply that knowledge to the example component.

Implementing the Server Status data object

To get back to the Server Status component we introduced at the beginning of this chapter, you need a data object that will be used to hold server statistics data that will be requested by and sent to the client browser. Here is the complete data object:

```
package org.gwtbook.client;

import com.google.gwt.user.client.rpc.IsSerializable;

public class ServerStatusData implements IsSerializable
{
    public String serverName;
    public long totalMemory;
    public long freeMemory;
    public long maxMemory;
    public int threadCount;
}
```

The class complies with both rules of serializable objects: It implements `IsSerializable`, and all the fields are also serializable. We built this class with only public fields, but it would be just as appropriate to provide private or protected fields with associated getter and setter methods, which is common in Java programming.

Also note that this class is in the `org.gwtbook.client` package, and it will be compiled into JavaScript to allow it to be used in the browser. On the server, you'll also use this class, but the server will use a compiled Java version of the class. As part of the data serialization handled by GWT, it will handle the mapping for the fields of the client-side JavaScript version to the server-side Java version of this same class.

Now that you have your data object, the next step is to define and implement your service. You need to define a service by using a Java interface and then implement a servlet that adheres to that interface.

10.2.2 Defining the GWT-RPC service

The next step in using the GWT-RPC mechanism is to define and implement the service that will live and be executed on the server. This consists of one Java interface which describes the service, and the service implementation. When you write a server-side service, you'll probably want to integrate it with other backend systems like databases, mail servers, and other services. In addition to the basics of using GWT-RPC, we'll touch on how you might do some of these things. First, we'll look at the GWT `RemoteService` interface.

Extending the RemoteService interface

To define your service, you need to create a Java interface and extend the GWT `RemoteService` interface. This is as easy as it sounds. If you recall, the Server Status project calls an RPC service method that returns a `ServerStatusData` object, which contains various server metrics. The interface looks like the following:

```
package org.gwtbook.client;

import com.google.gwt.user.client.rpc.RemoteService;

public interface ServerStatusService extends RemoteService
{
  ServerStatusData getStatusData ();
}
```

That is all there is to it: Provide a name for the interface, in this case `ServerStatusService`, and have it extend GWT's `RemoteService` interface. The `RemoteService` interface doesn't define any methods, so you won't need to implement anything special to get the service running. There are some additional fairly subtle requirements when defining this interface:

- The interface must extend `com.google.gwt.user.client.rpc.RemoteService`.
- All method parameters and return values must be serializable.
- The interface must live in the client package.

We have covered the first of these, but we want to explain it a little further. GWT automatically generates proxy classes for the client-side application and forwards to methods using reflection on the server. To put it another way, GWT goes out of its way to make RPC easy by minimizing the amount of code you need to write. The `RemoteService` interface is used to signal which interfaces define the remote service. This is important because this may not be the only interface the server implementation implements.

The second requirement is that all parameters and return values must be serializable. As we mentioned in the previous section, this includes all primitive Java types, certain objects that are part of the standard Java library, and classes that implement the `IsSerializable` interface. For the `ServiceStatusService` interface, the only method, `getStatusData()`, returns a `ServerStatusData` object that you created in the last section, and it implements `IsSerializable`. Here are some more examples, all of which include parameters and return values that can be serialized:

```
boolean serviceOne (String s, int i, Vector v);
String[] serviceTwo (float f, Integer o);
```

The last requirement is that the interface must be in the client package—the interface code must be in your project where it's compiled by GWT into JavaScript. Typically, this is in a package name ending in `.client` unless you have configured your project differently. This interface needs to be compiled to JavaScript because it's used by both the client and the server. On the server, the service implementation will implement this interface. We'll look at how you reference this interface from the client side when we get to the next section and call the service.

Using the typeArgs annotation

We introduced you to the `typeArgs` annotation in the previous section when you used it to define the contents of collections in a serializable object. Here, we'll introduce an alternate syntax, but with the same purpose—to provide a hint to the GWT compiler about what type of object a collection is holding. The specific collection types that GWT supports are `ArrayList`, `Vector`, `HashSet`, and their respective interfaces `List` and `Set`.

You should provide the `typeArgs` annotation for each parameter and return value that is a collection. Figure 10.6 shows the syntax, which differs in what you saw in the last section, because it also allows you to specify the parameter name. When you add an annotation for the return value, you don't specify the parameter name.

Theh following code defines a method that takes a `List` of `Integer` values and a `Vector` of `Date` values, and returns an `ArrayList` of `String` values:

```
/**
 * @gwt.typeArgs arg1 <java.lang.Integer>
 * @gwt.typeArgs arg2 <java.util.Date>
 * @gwt.typeArgs <java.lang.String>
 */
ArrayList operationThree (List arg1, Vector arg2);
```

In practice, you can probably leave off the `typeArgs` and everything will run fine. If you decide to leave it off, you may end up with additional JavaScript code being generated for the client because the GWT compiler won't be able to optimize the serialization code for handling `Collection` parameters and return values.

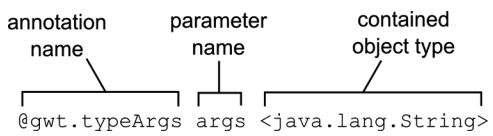

Figure 10.6 The second version of the `typeArgs` syntax, which is used to provide hints to the GWT compiler about the underlying data types contained by Collection arguments to the service

This covers serialization. But if you want remote calls to be able to pass Java objects, you also want them to have the ability to pass exceptions.

Throwing exceptions

Often, it's desirable to let a method throw an exception that will be handled by the calling code. For example, you may have a login function that returns a user-data object on success but, on failure, throws an appropriate exception:

```
UserData loginUser (String username, String password)
  throws FailedAuthenticationException;
```

When GWT calls the service method and throws an exception, it serializes the exception and returns it to the client browser. The only requirement is that the exception be serializable just like any data object. In other words, it must implement `IsSerializable`, all of its fields must be serializable, and the class must be in the client package. See the previous section, "Understanding serializable data objects," for a complete discussion of creating serializable objects.

As an alternative to writing your own serializable exception class from scratch, GWT supplies an exception class `SerializableException`, in the package `com.google.gwt.user.client.rpc`. You can use this exception class instead of writing your own, or you can use this as the base class for your exceptions.

Next, let's look at how you can implement the service interface.

Implementing the service

With the service interface defined for your service, you need to implement its methods. You do this by creating a servlet that extends GWT's `RemoteServiceServlet` and implements the service interface. Listing 10.3 shows the complete `ServerService` implementation.

Listing 10.3 Server Status server-side implementation

```
package org.gwtbook.server;

import org.gwtbook.client.ServerStatusData;
import org.gwtbook.client.ServerStatusService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

public class ServerServiceImpl
       extends RemoteServiceServlet
       implements ServerStatusService
{
  public ServerStatusData getStatusData () {
```

```
    ThreadGroup parentThread =
      Thread.currentThread().getThreadGroup();
    while (parentThread.getParent() != null) {
      parentThread = parentThread.getParent();
    }

    ServerStatusData result = new ServerStatusData();

    result.serverName = getThreadLocalRequest().getServerName();
    result.totalMemory = Runtime.getRuntime().totalMemory();
    result.freeMemory = Runtime.getRuntime().freeMemory();
    result.maxMemory = Runtime.getRuntime().maxMemory();
    result.threadCount = parentThread.activeCount();

    return result;
  }

}
```

Find root Java server thread

Create result object

Populate result object

return result

The benefit of using GWT-RPC, which listing 10.3 shows, is that there is nothing special you need to do to make your class a service other than extend the `RemoteServiceServlet` and implement an interface that extends `RemoteService`. When the service is called, the underlying `RemoteServiceServlet` parses the request, converting the serialized data back into Java objects, and calls the service method. When the method returns a value, it is returned to the `RemoteServiceServlet` that called the method, and it in turn serializes the result and returns it to the client browser.

Unlike what you've seen with the rest of GWT, there are no restrictions on the server-side code. You can use any Java class, you can use Java 5 syntax, and there are no special annotations you need to use. There is one restriction, however: the package name. Because the server-side code contains Java code that can't be compiled into JavaScript, and because there is no need for this to be served to the client, you need to make sure this class lives outside the client package. To refresh your memory, figure 10.7 shows what the current project directory layout looks like.

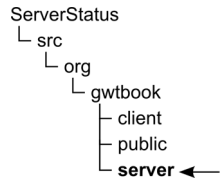

Figure 10.7 Reviewing the current project directory layout

The root of the project is the Java package `org.gwtbook`, and that package contains the `client` package and the public folder. The public folder contains any non-Java assets for the project, like the HTML file, and the `client` package contains Java classes that will be compiled to JavaScript. For the server-side code, it's

the standard practice to create a `server` package just beneath the root package, which for this project is org.gwtbook.server. Don't get too hung up on this; depending on what you're building, it may be more appropriate to use a different package name, and this is fine as long as it isn't in the client package.

Next, we'll look at how to configure the development environment to use the new servlet.

Setting up your service for hosted-mode testing

There is one last step: registering your service with GWT by adding it to the project configuration file. The project configuration file is located in the root package, and it's named the same as the entry-point class with a .gwt.xml extension; in this case, it's found under org.gwtbook and named ServerStatus.gwt.xml. By default, it contains both an `<inherits>` element to provide access to the core GWT libraries and an `<entry-point>` element indicating the runnable class for this project. To this, you need to add a `<servlet>` element to register your new service. The completed ServerStatus.gwt.xml file looks like this:

```
<module>
  <inherits name='com.google.gwt.user.User'/>
  <entry-point class='org.gwtbook.client.ServerStatus'/>

  <servlet path="/server-status"
           class="org.gwtbook.server.ServerServiceImpl"/>
</module>
```

We've stripped out the comments that the GWT application creator added, to shorten the example, so it may not look exactly like yours. The `<servlet>` element has two attributes: `path` to indicate the URL of your service, and `class` to indicate the servlet that should be run when this URL is requested. Note that just like a Java application server, the path to the servlet is relative to the web project root and not the root of the server. With this project, the default project URL in hosted mode is the following:

```
http://localhost:8888/org.gwtbook.ServerStatus/ServerStatus.html
```

When you specify /server-status as the path to the servlet, it's accessible with the following URL:

```
http://localhost:8888/org.gwtbook.ServerStatus/server-status
```

If you've worked with the servlet path setting before, this will be the expected behavior; but for some, this is a common cause of frustration when configuring a servlet.

With your service defined, implemented, and configured on the server, the next step is to address the client-side issues and finally call the service from the client.

10.2.3 Preparing the client side of a GWT-RPC call

When you call the remote service from the client, GWT does most of the work for you; however, you need to create one last interface. The GWT compiler uses this interface when it generates the service proxy object. A *proxy object* is an object instance that forwards the request to another target. In this case, you'll call a local method, and the proxy object is responsible for serializing the parameters, calling the remote service, and handling the deserialization of the return value. You don't write the code for the proxy class; the GWT compiler handles this for you. In the client-side code, you create the proxy object by writing the following:

```
GWT.create (ServerStatusService.class)
```

Here you call the static method `create()` of the `com.google.gwt.core.client.GWT` class, passing it the class object of the remote service interface. This returns a proxy object that you can use to set the service URL and call the remote methods. The proxy object returned implements two interfaces: one that you need to create, and one supplied by GWT (as shown in figure 10.8).

In figure 10.8, the proxy object is an instance of `ServerStatusService_Proxy`, which implements two interfaces. The proxy class is created at compile time, so you can't reference this class directly in your code. Instead, you need to cast it to each interface separately to be able to call its methods.

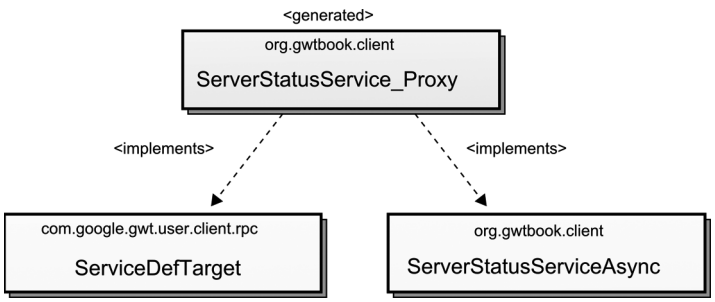

Figure 10.8 The client-side service proxy generated by the GWT compiler, and the interfaces it implements

Of the two interfaces, `ServiceDefTarget` is part of the GWT library and includes a method `setServiceEntryPoint()` for specifying the URL of the remote service. The other interface, `ServerStatusServiceAsync`, provides asynchronous methods for calling the remote service. You'll need to write this second asynchronous service interface yourself, as we'll discuss next.

This asynchronous service interface always has the same name as your service, with the name "Async" appended to it. The methods in the interface must match all the method names in your original service interface, but the signatures need to be changed. Specifically, for each method in the original interface, you must do the following:

- Set the return value to `void`.
- Add an extra `com.google.gwt.user.client.rpc.AsyncCallback` parameter.

Table 10.1 shows side-by-side examples of how a method in the service interface looks in the asynchronous service interface.

Table 10.1 Comparing methods in the service interface to how they look in the asynchronous interface of GWT-RPC

Service interface	Asynchronous service interface
`String methodOne (int i);`	`void methodOne (int i, AsynCallback cb);`
`List methodTwo ();`	`void methodTwo (AsynCallback cb);`
`boolean methodThree (int a, int x);`	`void methodThree (int a, int x, AsynCallback cb)`

This interface is used only by the client code and not the server. Because of this, you don't need to include this interface in any code deployed to the server, and you must place the interface in the client package. Here is what you get when you apply this to the `ServerStatusService` interface:

```
package org.gwtbook.client;

import com.google.gwt.user.client.rpc.AsyncCallback;
public interface ServerStatusServiceAsync {
  void getStatusData (AsyncCallback callback);
}
```

This sets you up for the last piece of the puzzle: calling the remote service from the client.

10.2.4 Calling the remote server service

Now that you have the remote service interface defined and implemented, you've created the remote asynchronous interface, and you've created a serializable object to pass between the server and client, all that remains is to call the service from the client browser. To do this, you need to do the following:

1. Instantiate a proxy object that will forward method calls to the server.
2. Specify the URL of the service.
3. Create a callback method to handle the result of the asynchronous method call.
4. Call the remote method.

The first time you do this, it doesn't feel natural, especially because you're calling the method asynchronously; but after a few tries, it becomes second nature. We'll examine each of these steps in turn and explain what needs to be done—and, perhaps more important, why it needs to be done. We'll also point out any areas of common mistakes.

Step 1: Creating the proxy object

Step 1 is to create your proxy object. You do this by calling `GWT.create()`, passing the remote service class as an argument. In return, the `create()` method returns a proxy object that you need to cast to the asynchronous interface. A common mistake is passing the wrong class as an argument to `GWT.create()`, so be sure to pass the remote service interface and not the asynchronous interface that will be implemented by the proxy object:

```
ServerStatusServiceAsync serviceProxy =
  (ServerStatusServiceAsync) GWT.create(ServerStatusService.class);
```

With the proxy object in hand, you can move on to the next step and use it to target the remote service.

Step 2: Casting the proxy object to ServiceDefTarget

The second step is to cast this same object to `ServiceDefTarget` so that you can specify the remote service URL. As you saw in the last section, the proxy object implements both the asynchronous service interface and `ServiceDefTarget`. All you need to do is cast this same object to the `ServiceDefTarget` interface. Once you do this, you can use `setServiceEntryPoint()` to set the URL for the service:

```
ServiceDefTarget target = (ServiceDefTarget) serviceProxy;
target.setServiceEntryPoint(GWT.getModuleBaseURL()
   + "server-status");
```

Or, alternatively, you can use a single-line syntax without creating a new variable:

```
((ServiceDefTarget)serviceProxy)
  .setServiceEntryPoint(GWT.getModuleBaseURL() + "server-status");
```

This sets the URL to /org.gwtbook.ServerStatus/server-status, which matches your servlet definition in the .gwt.xml file but may not match your production environment when you deploy the application. The `GWT.getModuleBaseURL()` method returns the location of the client-side code and appends a slash to the end, which works fine for hosted-mode development; but when you deploy your service, this may not be what you want.

Instead, you can detect whether the code is being executed in hosted mode, and use the appropriate service URL. You can do this by calling `GWT.isScript()`, which returns `true` when the application isn't running in hosted mode:

```
String serviceUrl = GWT.getModuleBaseURL() + "server-status";
if (GWT.isScript()) {
  serviceUrl = "/services/server-status.rpc";
}
((ServiceDefTarget)serviceProxy).setServiceEntryPoint(serviceUrl);
```

It may even be useful to take this a step further and define the web-mode service path in a `Constants` file or as a `Dictionary` object. You can get more information on how to use `Constants` and `Dictionary` in chapter 15.

You've set up the proxy object, so now you need to construct your callback object.

Step 3: Create a callback object

The third step in calling the RPC service is to create a callback object. This object implements the GWT `com.google.gwt.user.client.rpc.AsyncCallback` interface and is executed when a result is returned from the server. As you may recall, you added an `AsyncCallback` parameter to every method in the asynchronous service interface. The callback object is passed as this additional parameter. Here, you create an anonymous object instance that implements `AsyncCallback`:

```
AsyncCallback callback = new AsyncCallback() {

  public void onFailure (Throwable caught) {
    GWT.log("RPC error", caught);
  }

  public void onSuccess (Object result) {
    GWT.log("RPC success", null);
  }

};
```

The `AsyncCallback` interface has two methods that must be implemented: `onSuccess()` and `onFailure()`. If an error occurs where the service can't be reached, or if the server-side method throws an exception, the `onError()` method is called with the exception that occurred. If the call is successful, then the `onSuccess()` method is called, and it receives the return value of the remote method call. The previous sample uses the `GWT.log()` method to log information to the hosted-mode console. This would be replaced with code that does something with the resulting object and handles the error in an appropriate manner for the application.

Step 4: Make the remote service call

The fourth and final step in calling a remote service is to make the call. This is a little anticlimatic because it all comes down to one line of code:

```
serviceProxy.getStatusData(callback);
```

This method kicks off a chain of events. Any parameters other than the callback object are serialized and passed to the remote service, and the appropriate callback method is executed based on the server response.

10.3 Project summary

The GWT-RPC mechanism is simple; but with all the details, it's easy to lose site of the overall architecture. As promised earlier, we'll provide a summary of GWT-RPC, using a lot of visuals to make it easy to understand the system as a whole. We'll begin with an overview of the files in the project and then look at the server-side code, followed by the code on the client.

10.3.1 Project overview

The entire project thus far includes only seven files. Figure 10.9 provides a list of these files. It contains the usual configuration file, project HTML page, and entry point. That leaves only four files that represent the concepts covered in this chapter.

The *service interface* defines the remote service. It extends `RemoteService` and may only reference parameters and return values that can be serialized by GWT. When a method accepts arguments that implement `java.util.Collection`, or returns a `Collection`, you use the `@gwt.typeArgs` annotation to provide a hint for the GWT compiler, letting it know what types of objects it can contain. Your methods may declare that they throw exceptions, as long as any exception thrown can be serialized by GWT.

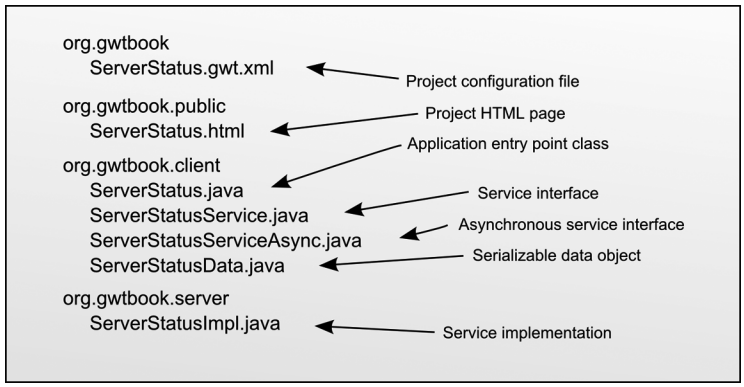

Figure 10.9 An overview of the Server Status project files you've created

The *asynchronous service interface* contains the same method names as the service interface, but with altered method signatures. Each method's return type is changed to `void`, and an `AsyncCallback` parameter is added as the last argument to each method.

The project may contain *serializable data objects*, each of which implements the `IsSerializable` interface. GWT serializes any field not marked as transient or final in the class, and every field not marked as transient must be of a GWT serializable type.

The *service implementation* is in a package outside the other client-side code because it doesn't need to be compiled to JavaScript by the GWT compiler. Typically, server-side code lives in a package at the same level as the client side code, in a package with the last part named `server`. The service implementation extends the `RemoteServiceServlet` and implements the service interface.

That is a description of each part of the project in a nutshell. In practice, the only difference between this service and other services you write will be the number of serializable data objects. You may also find that, in a large project, you wish to reorganize the package structure and place all the interfaces and data objects relating to a single service into a package by themselves.

Let's take a closer look at the server side of the project.

10.3.2 Server-side service implementation

Figure 10.10 presents a class diagram of all the classes on the server and their relationship to each other. We've used the stereotype `<<client-side>>` to mark those classes that are used on the client side as well as the server and need to be compiled

by the GWT compiler. Those classes must meet the requirements for client-side code, including referencing only classes that are a part of the JRE Emulation Library, or client-side user-defined classes. All other classes may take advantage of Java 5 syntax and use any classes that are available on the server.

The server-side service implementation is just a servlet, because it extends `RemoteServiceServlet`, which in turn extends `HttpServlet`. When you deploy your service. you install it on the server just like any other servlet.

Finally, let's look at the client side of the project.

10.3.3 Calling the service from the client

You call the remote service by creating an instance of a service proxy object, specifying the URL of the service, and calling the server-side method on the object.

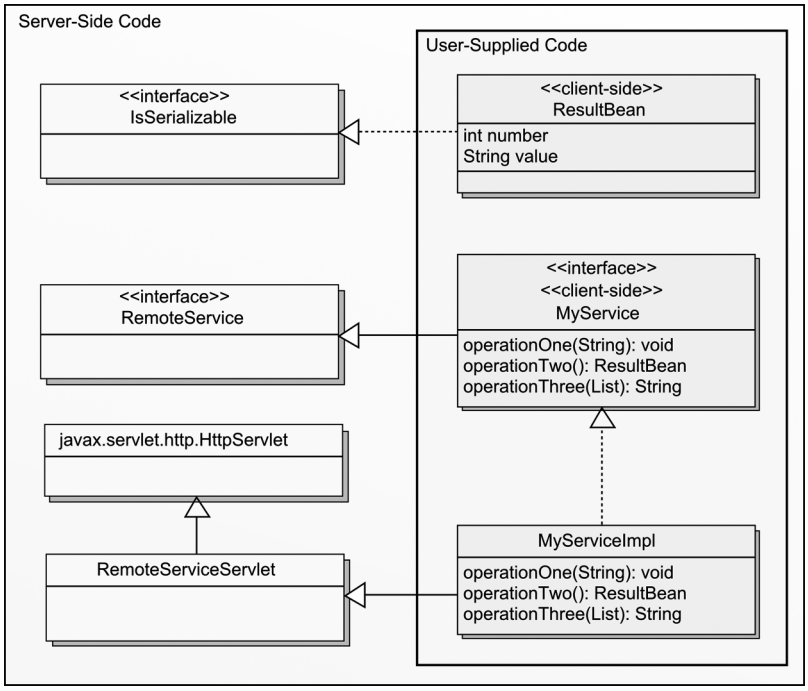

Figure 10.10 Overview of the server-side classes used for the Server Status project, but user generated and GWT supplied

The last parameter to any such method call is a callback handler, which is executed following the server returning a result. Listing 10.4 is a complete listing of the entry-point class that sets up and calls the remote service.

Listing 10.4 Client-side implementation

```
package org.gwtbook.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;

public class ServerStatus implements EntryPoint
{
  public void onModuleLoad ()
  {
    ServerStatusServiceAsync serviceProxy =
        (ServerStatusServiceAsync)
        GWT.create(ServerStatusService.class);

    ServiceDefTarget target = (ServiceDefTarget) serviceProxy;
    target.setServiceEntryPoint(GWT.getModuleBaseURL()
        + "server-status");

    AsyncCallback callback = new AsyncCallback()
    {
      public void onFailure (Throwable caught)
      {
        GWT.log("Error", caught);
      }

      public void onSuccess (Object result)
      {
        ServerStatusData data = (ServerStatusData) result;
        GWT.log("Server Name: " + data.serverName, null);
        GWT.log("Free Memory: " + data.freeMemory, null);
        GWT.log("Max Memory: " + data.maxMemory, null);
      }
    };

    serviceProxy.getStatusData(callback);
  }
}
```

10.4 Summary

In this chapter, you created an RPC service using the GWT-RPC mechanism and called the service from the web browser, passing serialized Java objects. This chapter covers the basics of how to call a remote service, but we haven't yet discussed how to solve common real-world problems. For instance, how do you continuously poll a server for updates, and what is the best way to architect client-side RPC? In chapter 11, we'll answer both of these questions as we finish the Server Status project and take a hard look at client-side architecture.

AJAX / JAVA

GWT IN ACTION

Robert Hanson and Adam Tacy

With Ajax you can create great interfaces, but testing and team development are difficult and poor tool support adds to the hassle. Not so with GWT. The Google Web Toolkit is an open source Java framework that turns Java code into browser-neutral JavaScript and HTML. Along the way you get the power and tools of the full Java platform. GWT includes a complete widget library and good RPC support for full application development.

GWT in Action is a clearly written, comprehensive tutorial on the Google Web Toolkit. It covers the GWT development cycle, from setting up your programming environment, to building the application, then deploying it to the web server. By following a running example, you'll master widgets, panels, and events: the basic building blocks of a GWT application. Then you'll learn how to reuse existing JavaScript components, communicate via GWT-RPC, and interoperate with JSON. Along the way, you'll pick up great techniques for internationalization, testing, and deployment. This book assumes a working knowledge of Java.

What's Inside:

- Build a full-scale GWT project
- Master GWT-RPC and JSON interaction
- Create flexible and scalable applications
- Try the GWT Dashboard live at www.manning.com/GWTinAction

Robert Hanson is a US-based Internet engineer and creator of the popular open source GWT Widget Library. **Adam Tacy** is a consultant working for WM-Data in Sweden and a contributor to the GWT Widget Library.

For more information, code samples, and to purchase an ebook visit www.manning.com/GWTinAction

"... impressive quality and thoroughness. Wonderful!"

—Bernard Farrell, Software Architect, Kronos Inc.

"How to 'think in GWT.' The code: concise, efficient, thorough, and plentiful."

—Scott Stirling
Senior Consultant at USi
an AT&T Company

"Perfect for Java developers struggling with JavaScript."

—Carlo Gottiglieri
Java Developer, Sytel-Reply

"A real nitty-gritty tutorial on the rich features of GWT."

—Andrew Grothe
COO Eliptic Webwise, Inc.

ISBN-10: 1-933988-23-1
ISBN-13: 978-1-933988-23-8

$49.99 / Can $64.99