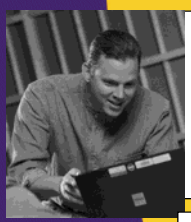
# The Google Web Toolkit:
## Using GWT RPC to Access Server-Side Data
**(GWT 2.5 Version)**

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/gwt.html






## For live Ajax & GWT training, see training courses at http://courses.coreservlets.com/.



Taught by the author of *Core Servlets and JSP*, *More Servlets and JSP*, and this tutorial. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - JSF 2, PrimeFaces, servlets/JSP, Ajax, jQuery, Android development, Java 7 or 8 programming, custom mix of topics
  - Courses available in any state or country. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring, Hibernate/JPA, GWT, Hadoop, HTML5, RESTful Web Services

**Contact hall@coreservlets.com for details**

# Topics in This Section

- **Idea of RPC**
- **Development process**
  - Defining main data service interface
  - Defining callback version of data service interface
  - Making a data service servlet
  - Creating a service proxy
  - Defining client-side callback objects
- **Examples**
  - Getting simple data from server
  - Getting complex types from server
  - Getting serializable custom classes from server
  - Sending serializable custom class to server
- **Handling asynchronous methods**
  - Performing server-dependent ops only in callback methods
- **Testing**



# RPC: Big Idea

- **Write regular methods on server; don't write servlet methods**
  - Methods take arbitrary arguments
    - Not HttpServletRequest and HttpServletResponse
  - Methods return arbitrary results
    - Strings, arrays, lists, custom classes, etc.
- **Call methods directly from client; don't make explicit HTTP requests**
  - Call server methods almost as though they were local
    - No need to create or parse query parameters
  - Pass arbitrary arguments and get arbitrary results
    - Custom form of serialization handles all the parsing

# RPC: Notes

- **Online RPC documentation is good**
  - GWT RPC tutorial
    - https://developers.google.com/web-toolkit/doc/latest/tutorial/RPC
  - Client-Server development guide
    - https://developers.google.com/web-toolkit/doc/latest/DevGuideServerCommunication
- **GWT does not require RPC**
  - It is a tremendous simplification, and should be used for all *new* GWT projects. However, if you *already* have a server-side resource that uses JSON, you can access it.
    - See client-server dev guide listed above
    - However, this approach is much lower-level, and it requires JSNI. We will cover JSNI in a later section.



# RPC Data Types

- **Server methods can accept & return complex types**
  - Packing and unpacking handled automatically
    - Even though client-side code is JavaScript (not Java) at runtime
- **Legal types**
  - Primitives
    - int, double, boolean, etc.
  - Wrappers
    - Integer, Double, Boolean, etc.
  - A subset of standard Java types
    - ArrayList, Date, HashMap, HashSet, String, etc.
      - For full list, see the JRE Emulation Reference at https://developers.google.com/web-toolkit/doc/latest/RefJreEmulation
  - Custom classes that implement Serializable
  - Arrays containing any of the above types

# GWT-RPC Development Steps

- **Define main service interface**
  - Must be under .client package
  - Implement RemoteService interface
  - Define regular methods without explicit HTTP
  - Use @RemoteServiceRelativePath to point at servlet
- **Define callback version of service interface**
  - Must be under .client package
  - If main interface is FooService, define FooServiceAsync
  - Eclipse automates this for you
    - Once you define the main service interface, Eclipse will create the Async version.
    - And, every time you add or change a method to main interface, Eclipse will add or update the Async version



# GWT-RPC Development Steps (Continued)

- **Make service servlet**
  - Must be *outside* of the .client package
  - Extend RemoteServiceServlet, implement service interface
  - Supply url-pattern in web.xml that matches relative path
- **Create service proxy**
  - Must be under .client package (usually in onModuleLoad)
  - Call GWT.create(*YourServiceInterfaceName*.class)
- **Define client-side callback objects**
  - Must be under .client package
  - With onSuccess and onFailure

# Example 1: Getting a Random Number from the Server



## Example Overview

- **Goal**
  - Press button, call method on server that returns String
  - Display String on client inside table cell
- **Point**
  - General process of calling server
- **Assumed project setup already done**
  - Clicked "g" to create new Web Application Project
  - Deleted extraneous files as described in last section
  - In auto-generated HTML file, removed everything except script tag and replaced with custom HTML
  - In auto-generated Java class, removed everything except class definition and signature of onModuleLoad

# Step 1: Define Main Service Interface

- **Interface**
  - Must be under the .client package
  - Must implement RemoteService interface
    - This is just a marker interface: no required methods
- **Methods**
  - Regular methods that accept and return normal Java types
  - Remember: Only a subset of Java 6 types are supported
- **Annotation**
  - Use @RemoteServiceRelativePath("some-path")
  - Gives relative URL of servlet that implements interface
    - Relative to GWT "appname" subfolder. So, url-pattern of servlet (in web.xml) would be /appname/some-path
      - "appname" comes from <module> element's 'rename-to' attribute from *AppName*.gwt.xml file



# Main Service Interface: Example

```
package coreservlets.client;

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

@RemoteServiceRelativePath("data-service")
public interface DataService extends RemoteService {
  public String getButton1Data();
  public String[] getButton2Data();
  public RandomNumber getButton3Data(String range);
  public void logAjaxRandom(RandomNumber clientRandom);
}
```

This is relative to GWT appname subfolder. So, by default, if main URL is http://hostname/AppName/, the URL of the data service servlet will be http://hostname/AppName/appname/data-service. Thus, the url-pattern of the servlet should be /appname/data-service to match this.

Will look at this method in this section. Next sections cover the other three methods.

# Step 2: Define Callback Version of Service Interface

- **Interface organization**
  - Name: *OriginalInterface*Async, under .client package
  - Return types: always void
  - Arguments
    - Same as in main data service interface, except with AsyncCallback<*OriginalReturnType*> as last arg
- **Example methods**
  - Main service interface (BlahService)
    - public double getSomeValue();
    - public String[] findLastNames(int n, String firstName);
  - Callback version of interface (BlahServiceAsync)
    - public void getSomeValue(AsyncCallback<Double> callback);
    - public void findLastNames(int n, String firstName, AsyncCallback<String[]> callback);



# Callback Version of Service Interface: Example

```
package coreservlets.client;

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

public interface DataServiceAsync {
  public void getButton1Data(AsyncCallback<String> callback);
  public void getButton2Data(AsyncCallback<String[]> callback);
  public void getButton3Data(String range,
                             AsyncCallback<RandomNumber> callback);
  public void logClientRandom(RandomNumber clientRandom,
                              AsyncCallback<Void> callback);
}
```

Changed return type from String to void.

Added AsyncCallback<String> to end of argument list. String is return type of original (non-callback) version of getButton1Data.

None of this code was written by hand. Given the main service interface, Eclipse automates creating this Async interface and defining each of the methods.

# Step 3: Make Service Servlet

- **Extend RemoteServiceServlet**
  - Put in server package (or any package other than "client")
- **Implement service interface from step 1**
  - Implement the methods in that interface
    - Normal methods with the arguments and return types exactly as listed in the interface definition
  - May use any Java features in method body
    - Whatever Java version your server is running
      - You can even use lambdas is your server is running Java 8!
    - No restriction to use GWT-client subset of Java types
  - Remember the interface is in client package
    - So you need to import mainPackage.client.MainInterface
- **Define url-pattern in web.xml**
  - Should be /appname/path-given-in-annotation



# Service Servlet: Example

```
package coreservlets.server;

import com.google.gwt.user.server.rpc.*;
import coreservlets.client.*;

public class DataServiceImpl
                          extends RemoteServiceServlet
                          implements DataService {
  public String getButton1Data() {
    String result =
      String.format("Number: %.2f", Math.random()*10);
    return(result);
  }
  ...
```

Service interface from step 1.

All Java features permitted. GWT does not support String.format, but server-side code can still use it. Code in "server" package does *not* get compiled into JavaScript.
*No code outside the ..client.* package gets compiled into JavaScript.*

# Data Service Servlet: web.xml Settings

```
…
 <servlet>
    <servlet-name>
      Servlet that Provides DataService
    </servlet-name>
    <servlet-class>
      coreservlets.server.DataServiceImpl
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>
      Servlet that Provides DataService
    </servlet-name>
    <url-pattern>/gwtrpc/data-service</url-pattern>
  </servlet-mapping>
…
```

"gwtrpc" is the lowercase version of project name. This folder created automatically by GWT within "war".

"data-service" matches @RemoteServiceRelativePath annotation from the DataService interface.



# Step 4: Create Service Proxy

- **Idea**
  - Must be under .client package (usually in onModuleLoad)
  - Create instance of callback (Async) version of interface
  - Call server methods, passing in a callback handler
  - The callback handler gets invoked with server data
- **Steps**
  - Main code
    - YourServiceInterfaceAsync serviceProxy = GWT.create(YourServiceInterface.class); serviceProxy.someServerMethod(arg1, arg2, new CallbackHandler());
  - Callback handler
    - Implements AsyncCallback<MethodReturnType>
    - Has onSuccess and onFailure

# Creating Service Proxy: Example

```
public class GwtRpc implements EntryPoint {
  private DataServiceAsync serviceProxy;
  private HTML label1, label2, label3, label4;
  private TextBox rangeBox;

  public void onModuleLoad() {
    serviceProxy = GWT.create(DataService.class);
    Button button1 = new Button("Show Random Number");
    label1 = new HTML("<i>Num will go here</i>");
    button1.addClickHandler(new Button1Handler());
    RootPanel.get("button1").add(button1);
    RootPanel.get("label1").add(label1);
    …
  }
  …
}
```

Callback interface type from step 2.

Main interface type from step 1.



# Step 5: Define Client-Side Callback Objects

- **Make a class that implements AsyncCallback<OrigMethodReturnType>**
  - Must be under .client package
  - onSuccess
    - Invoked when call to server is successful. Returns whatever type the original service method returns
  - onFailure
    - Invoked when call to server fails
  - As with event handling in Swing, SWT, or AWT, you often use inner classes so that the methods can easily access data from main app
- **Call interface method on service proxy**
  - Pass the expected args *plus the callback object*
  - You usually do this from a GUI event handler

# Defining Client-Side Callback Objects: Details

- **Main data service interface**
  - public String getMessage(int level);
- **Callback version of data service interface**
  - public void getMessage(int level,
    AsyncCallback<String> callback);
- **Callback class**
  - private class SomeCallback
    implements AsyncCallback<String> {
    public void onSuccess(String result) {…}
    public void onFailure(Throwable caught) {…}
    }
- **Invoking service proxy**
  - serviceProxy.getMessage(someInt, new SomeCallback());



# Defining Client-Side Callback Objects: Example

```
package coreservlets.client;

public class GwtRpcApplication implements EntryPoint {
  …
  private class Button1Handler implements ClickHandler {
    public void onClick(ClickEvent event) {
      serviceProxy.getButton1Data(new Button1Callback());
    }
  }

  private class Button1Callback implements AsyncCallback<String>{
    public void onSuccess(String result) {
      label1.setHTML(result);
    }

    public void onFailure(Throwable caught) {
      Window.alert("Unable to get data from server.");
    } }
```

This is the event handler for the first pushbutton.

In main service interface (DataService), getButton1Data takes no args.
In callback version (DataServiceAsync), getButton1Data takes one extra arg: a callback.

In main service interface (DataService), getButton1Data returns a String. So, use String here.

# Auto-Generated HTML File: Top (war/GwtRpc.html)

```
<!doctype html>
<html>
<head><title>Using GWT RPC</title>
<link rel="stylesheet"
      href="./css/styles.css"
      type="text/css"/>
<script type="text/javascript" language="javascript"
        src="gwtrpc/gwtrpc.nocache.js"></script>
</head>
<body>
<div align="center"><br/>
<table border="5">
  <tr><th class="title">Using GWT RPC</th></tr>
</table>
<p/>
```

Created automatically by GWT and left unchanged. Except for this line, all of HTML file edited by hand.



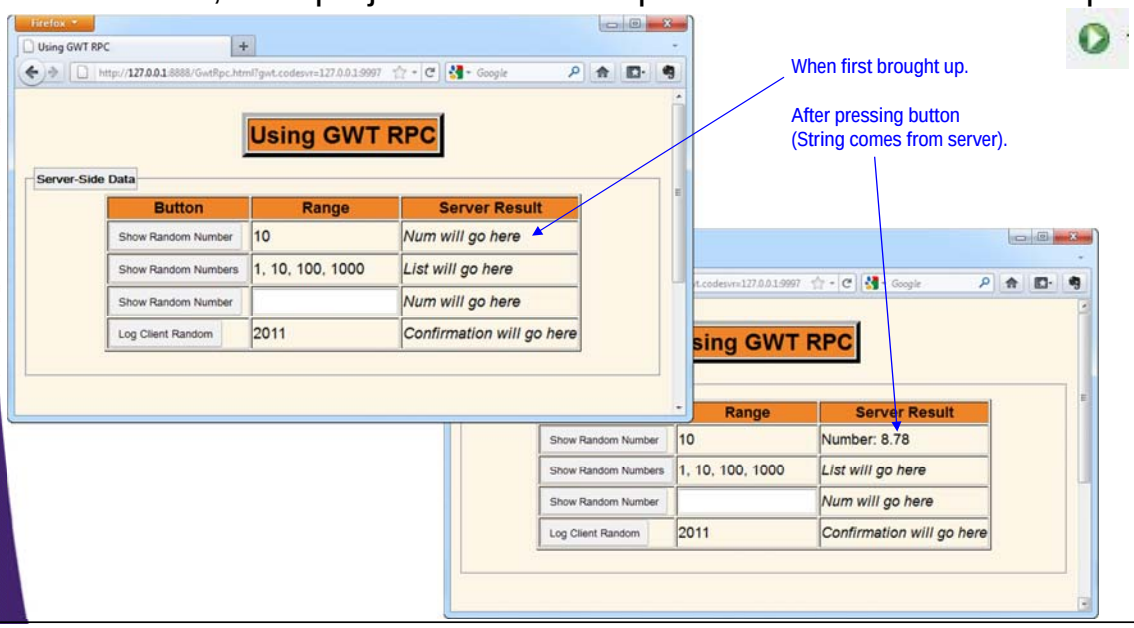
# Main HTML File: Continued (war/GwtRpc.html)

```
<fieldset>
<legend>Server-Side Data</legend>
<table border="1">
  <tr><th>Button</th>
      <th>Range</th>
      <th>Server Result</th></tr>
  <tr><td id="button1"></td>
      <td>10</td>
      <td id="label1"></td></tr>
  <tr><td id="button2"></td>
      <td>1, 10, 100, 1000</td>
      <td id="label2"></td></tr>
  <tr><td id="button3"></td>
      <td id="rangeBox"></td>
      <td id="label3"></td></tr>
  <tr><td id="button4"></td>
      <td id="range4">2011</td>
      <td id="label4"></td></tr>
</table><br/>
</fieldset>
</div></body></html>
```

The ids are referenced in GwtRpc.java.

# Testing in Development Mode

- R-click project name and Run As → Web Application
  - Or, click project name in Eclipse. Press Run button at top



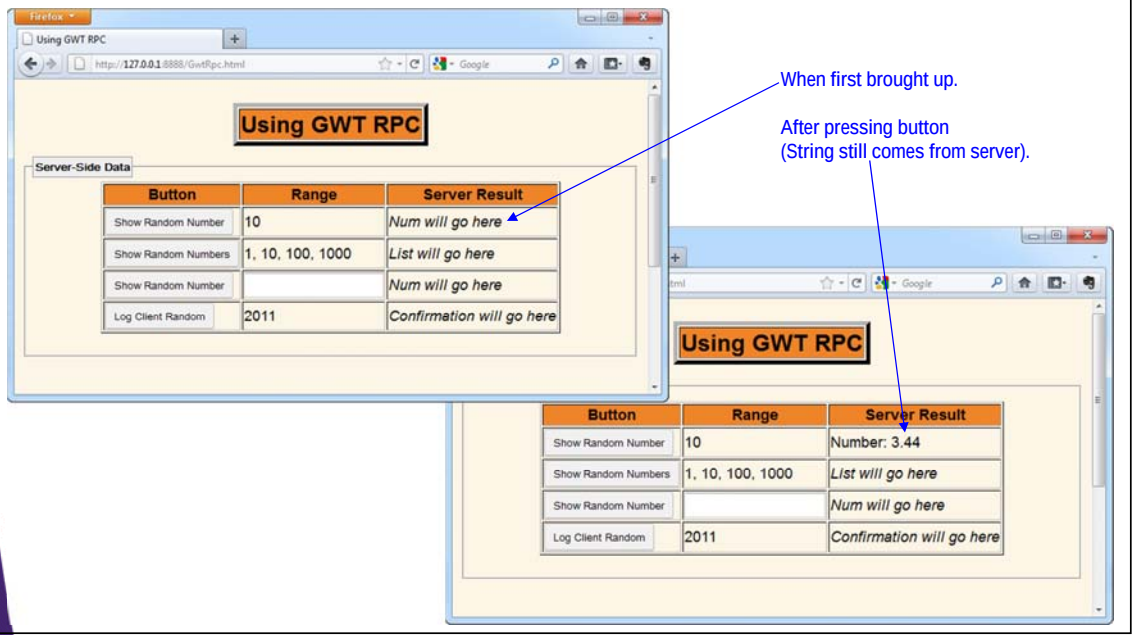



# Testing in Production Mode

- **Idea**
  - Running in a regular browser with all client-side Java code converted to JavaScript
    - Do most testing/debugging in development mode, but use Web mode occasionally to check browser support
    - Uses embedded version of Jetty
- **Steps**
  - Run the application in development mode
  - R-click project, then Google → GWT Compile
    - Or, click project, then click red toolbox at top
    - Creates JavaScript files in war/appname folder
  - Change URL by removing ?gwt.codesvr=…
    - Can copy URL to other browsers to test cross-browser compatibility

# Testing in Production Mode

- Did Google → GWT Compile, then changed URL



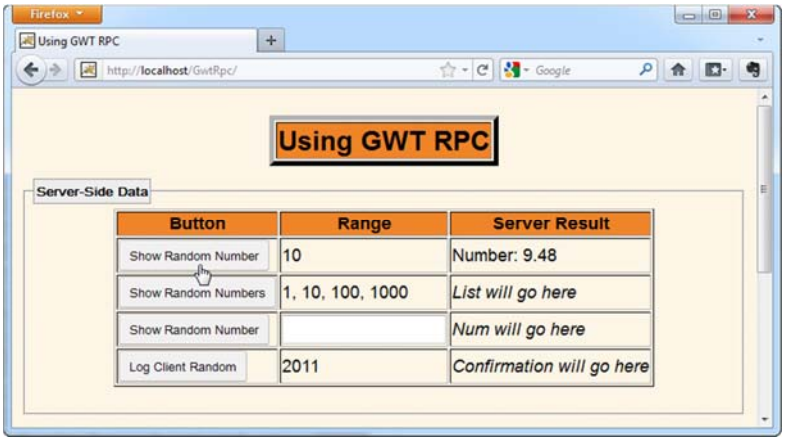



# Testing in Deployed Mode

- **Idea**
  - Deployed mode is similar to production mode
    - Running in a regular browser with all client-side Java code converted to JavaScript
    - However, uses normal Web App structure running on a Java server of your choice on any machine
    - Do initial debugging in development mode, do occasional testing in production mode, test in deployed mode at end
- **Steps**
  - Run the application in development mode
  - Do Google → GWT Compile (or click red toolbox at top)
  - Find the "war" folder on filesystem
    - Copy/rename and deploy as exploded WAR
    - Or, copy contents (not "war" folder itself) into file.zip, then rename file.zip to file.war and deploy as WAR file

# Testing in Deployed Mode

- **Copied workspace/GwtRpc/war folder**
  - Renamed to GwtRpc to keep similar-looking URL
    - GwtRpc.html is welcome-file, so name can be omitted
  - Deployed to Apache Tomcat on *localhost*
    - Could have used any Java-capable server








# Example 2: Getting an Array of Numbers from the Server

# Example Overview

- **Goal**
  - Press a button, call a method on the server that returns an array of Strings.
  - On the client, build a bulleted list out of the array
- **Points**
  - Returning complex data types to client
    - Nothing extra is required, unlike with JSON-RPC.
    - GWT will automatically package and parse arrays.
  - Building HTML on the client
    - In many GWT apps, you manipulate GWT widgets based on the server data. But you are also allowed to explicitly build HTML strings. Simpler than in many Ajax tools
      - You are dealing with a normal array instead of JSON or XML
      - Can use Java-style loops & String methods (but not String.format)



# Step 1: Main Service Interface

```
package coreservlets.client;

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

@RemoteServiceRelativePath("data-service")
public interface DataService extends RemoteService {
  public String getButton1Data();
  public String[] getButton2Data();
  public RandomNumber getButton3Data(String range);
  public void logClientRandom(RandomNumber
                              clientRandom);
}
```

# Step 2: Callback Version of Service Interface

```
package coreservlets.client;

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

public interface DataServiceAsync {
  public void getButton1Data(AsyncCallback<String> callback);
  public void getButton2Data(AsyncCallback<String[]> callback);
  public void getButton3Data(String range,
                             AsyncCallback<RandomNumber> callback);
  public void logClientRandom(RandomNumber clientRandom,
                              AsyncCallback<Void> callback);
}
```

Added AsyncCallback<String[]> to end of argument list. String[] is return type of original (non-callback) version of getButton2Data.

Changed return type from String[] to void.



# Step 3: Service Servlet

```
public class DataServiceImpl
                          extends RemoteServiceServlet
                          implements DataService {
  public String getButton1Data() {
    String result =
      String.format("Number: %.2f", Math.random()*10);
    return(result);
  }

  public String[] getButton2Data() {
    String[] results =
      { String.format("%.2f", Math.random()),
        String.format("%.2f", Math.random() * 10),
        String.format("%.2f", Math.random() * 100),
        String.format("%.2f", Math.random() * 1000) };
    return(results);
  }
  ...
}
```

As before:
- Wrote normal method with normal arguments and return values (not servlet method)
- Used String.format even though it is not supported by GWT on the client
- Gave url-pattern of "data-service" in web.xml

# Step 4: Service Proxy

```
public class GwtRpc implements EntryPoint {
  private DataServiceAsync serviceProxy;
  private HTML label1, label2, label3, label4;
  private TextBox rangeBox;

  public void onModuleLoad() {  // Main entry point
    serviceProxy = GWT.create(DataService.class);
    …
    label2 = new HTML("<i>List will go here</i>");
    button2.addClickHandler(new Button2Handler());
    RootPanel.get("button2").add(button2);
    RootPanel.get("label2").add(label2);
    …
  }
  …
}
```

Service proxy definition unchanged from previous examples.
But code above also shows GUI controls specific to this example.



# Step 5: Client-Side Callback Objects

```
public class GwtRpcApplication implements EntryPoint {
  …
  private class Button2Handler implements ClickHandler {
    public void onClick(ClickEvent event) {
      serviceProxy.getButton2Data(new Button2Callback());
    }
  }
```

# Step 5: Client-Side Callback Objects (Continued)

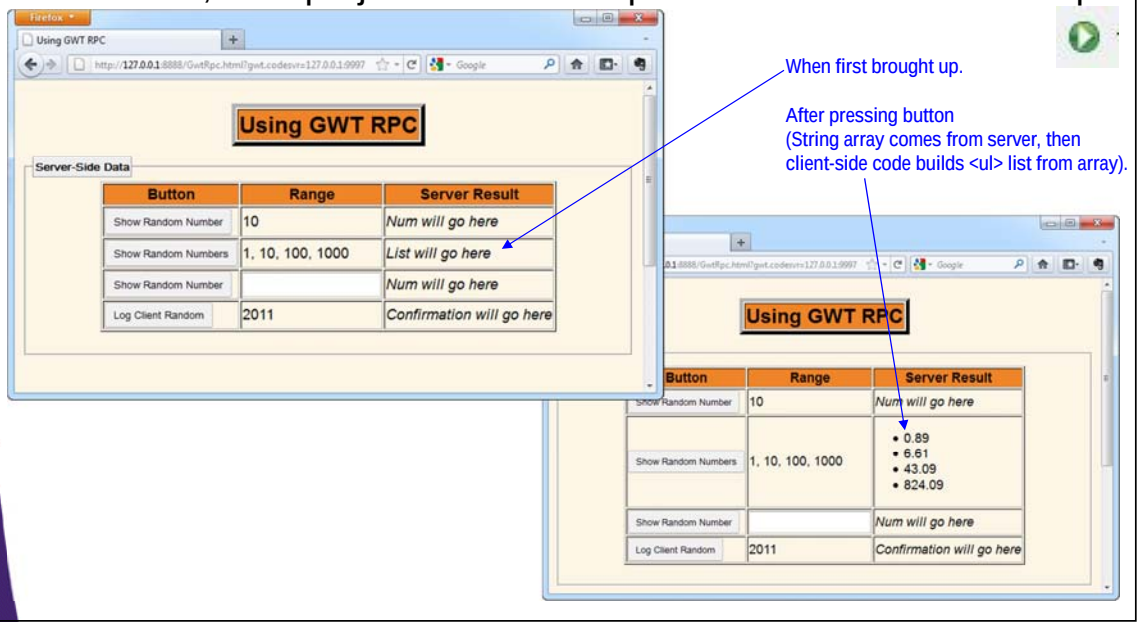
```
private class Button2Callback
        implements AsyncCallback<String[]> {
  public void onSuccess(String[] listItems) {
    String result = "<ul>\n";
    for(String item: listItems) {
      result = result + "<li>" + item + "</li>\n";
    }
    result = result + "</ul>";
    label2.setHTML(result);
  }

  public void onFailure(Throwable caught) {
    Window.alert("Unable to get data from server.");
  }
}
```

In main data service interface (DataService), getButton2Data returns String array. So, use String[] here.



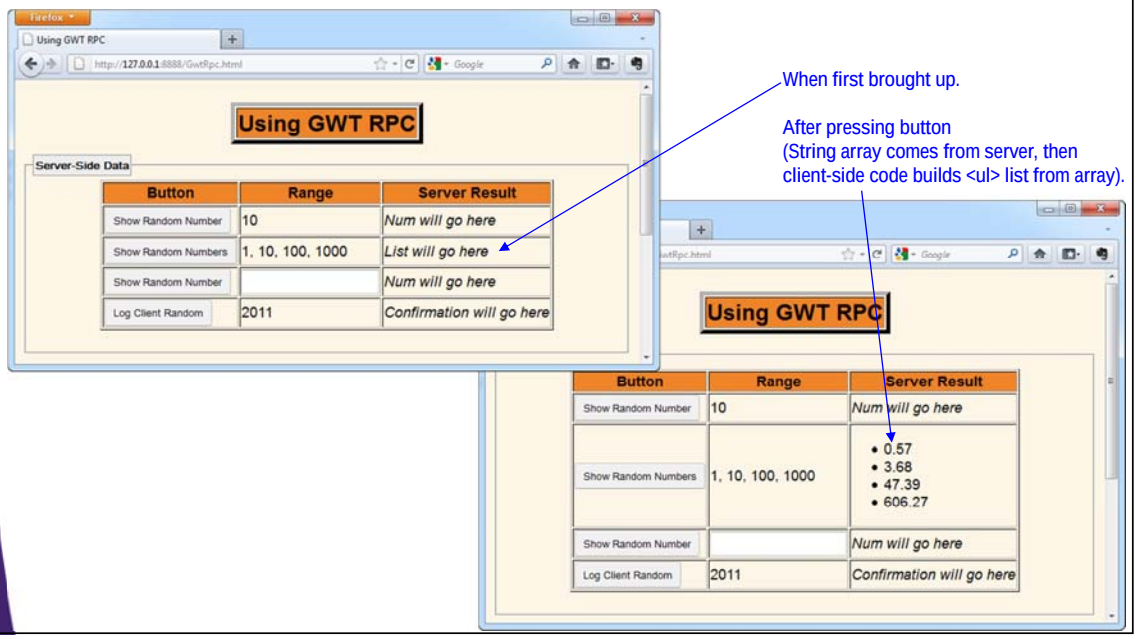
# Testing in Development Mode

- R-click project name and Run As → Web Application
  - Or, click project name in Eclipse. Press Run button at top



# Testing in Production Mode

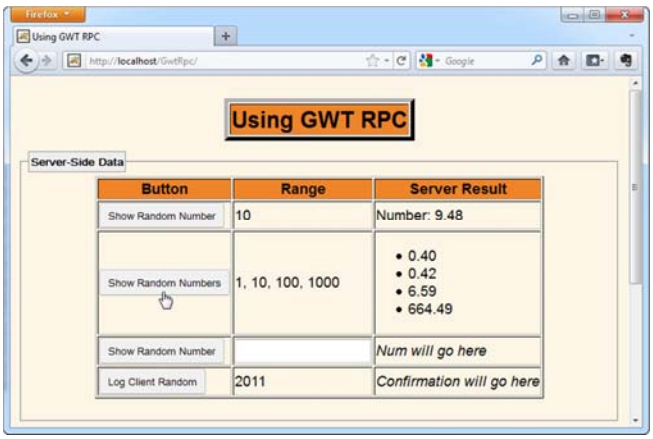
– Did Google → GWT Compile, then changed URL






# Testing in Deployed Mode

- **Deployed to Tomcat on localhost**
  - Copied "war" folder and renamed to "GwtRpc"
  - Or, made GwtRpc.zip, put *contents* of "war" (not "war" itself) inside it, and then renamed to GwtRpc.war.
    - Either way, deployed result to Web apps folder of server



# Example 3: Getting a Serialized Object from the Server



## Example Overview

- **Goal**
  - Press a button, pass a value to a server-side method
  - Server method returns a custom RandomNumber object
  - On the client, extract data from the RandomNumber by calling accessor methods
    - I.e., use normal Java approach. No parsing required.
- **Points**
  - Passing data to server-side methods
  - Returning Serializable data types to client
    - You implement Serializable
    - GWT manages the network marshalling/unmarshalling
      - Even though client-side code is really in JavaScript!

# Step 1: Main Data Service Interface

```
package coreservlets.client;

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

@RemoteServiceRelativePath("data-service")
public interface DataService extends RemoteService {
  public String getButton1Data();
  public String[] getButton2Data();
  public RandomNumber getButton3Data(String range);
  public void logClientRandom(RandomNumber
                              clientRandom);
}
```



# Step 2: Callback Version of Service Interface

```
package coreservlets.client;

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

public interface DataServiceAsync {
  public void getButton1Data(AsyncCallback<String> callback);
  public void getButton2Data(AsyncCallback<String[]> callback);
  public void getButton3Data(String range,
                             AsyncCallback<RandomNumber> callback);
  public void logClientRandom(RandomNumber clientRandom,
                              AsyncCallback<Void> callback);

}
```

Added AsyncCallback<RandomNumber> to end of argument list. RandomNumber is return type of original (non-callback) version of getButton3Data.

Changed return type from RandomNumber to void.

# Step 3: Service Servlet

```
public class DataServlet extends RemoteServiceServlet
                         implements DataService {
  public String getButton1Data() {
    String result =
      String.format("Number: %.2f", Math.random()*10);
    return(result);
  }

  public String[] getButton2Data() {
    String[] results =
      { String.format("%.2f", Math.random()),
        String.format("%.2f", Math.random() * 10),
        String.format("%.2f", Math.random() * 100),
        String.format("%.2f", Math.random() * 1000) };
    return(results);
  }

  public RandomNumber getButton3Data(String rangeString) {
    return(new RandomNumber(rangeString));
  }
}
```



# Serializable Classes

- **Classes implement java.io.Serializable**
  - Or com.google.gwt.user.client.rpc.IsSerializable
  - This interface is just a marker
    - No required methods
    - Must have zero-argument (default) constructor.
      - Of course, if you define *no* constructors, Java gives you a 0-arg one.
    - Property types must also follow Serializable rules
- **Server-side code uses this class normally**
  - Normal Java class, but class must restrict itself to the Java subset supported by GWT (no printf or String.format, etc)
- **Client-side code declares argument to onSuccess to be of this type**
  - Then uses it like a normal Java class, even though in production or deployed mode it really becomes JavaScript

# Code for RandomNumber (Used by client *and* server)

```
package coreservlets.client;
import java.io.*;

public class RandomNumber implements Serializable {
  private double range = 0;
  private double value;
  private boolean isRangeLegal;

  public RandomNumber(String rangeString) {
    try {
      range = Double.parseDouble(rangeString);
      isRangeLegal = true;
    } catch(NumberFormatException nfe) {}
    if (range <= 0) {
      range = 1.0;
      isRangeLegal = false;
    }
    value = Math.random()*range;
  }

  public RandomNumber() {
    this("-1");
  }
```

You are required to have a zero-arg constructor, even though in this case the explicit Java code never uses this constructor.



# Code for RandomNumber (Continued)

```
  public double getRange() {
    return (range);
  }

  public double getValue() {
    return (value);
  }

  public boolean isRangeLegal() {
    return (isRangeLegal);
  }
}
```

The Serializable interface contains no methods, so classes that implement Serializable need no special methods. Subclasses of serializable classes are automatically serializable.

# Step 4: Service Proxy

```
public class GwtRpc implements EntryPoint {
  private DataServiceAsync serviceProxy;
  private HTML label1, label2, label3, label4;
  private TextBox rangeBox;

  public void onModuleLoad() {  // Main entry point
    serviceProxy = GWT.create(DataService.class);
    …
    Button button3 = new Button("Show Random Number");
    rangeBox = new TextBox();
    label3 = new HTML("<i>Num will go here</i>");
    button3.addClickHandler(new Button3Handler());
    RootPanel.get("button3").add(button3);
    RootPanel.get("rangeBox").add(rangeBox);
    RootPanel.get("label3").add(label3);    …
  } … }
```

Service proxy definition unchanged from previous examples.
But code above also shows GUI controls specific to this example.



# Step 5: Client-Side Callback Objects

```
public class GwtRpcApplication implements EntryPoint {
  …
 private class Button3Handler implements ClickHandler {
    public void onClick(ClickEvent event) {
      String range = rangeBox.getText();
      serviceProxy.getButton3Data(range,
                                  new Button3Callback());
    }
  }
```

# Step 5: Client-Side Callback Objects (Continued)

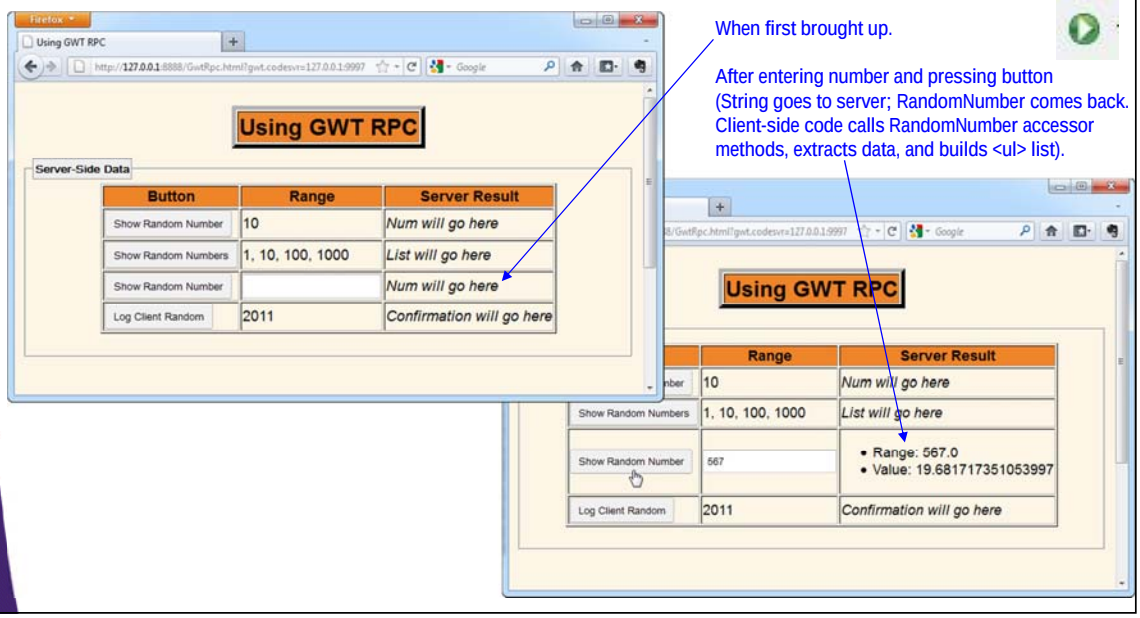
```
private class Button3Callback implements
        AsyncCallback<RandomNumber> {
   public void onSuccess(RandomNumber number) {
     String range = "Range: " + number.getRange();
     if (!number.isRangeLegal()) {
       range = range + " (default due to illegal range)";
     }
     String value = "Value: " + number.getValue();
     String result = "<ul>\n" +
                     "<li>" + range + "</li>\n" +
                     "<li>" + value + "</li>\n" +
                     "</ul>";
     label3.setHTML(result);
   }
   public void onFailure(Throwable caught) {
     Window.alert("Unable to get data from server.");
   }
 }
```

In main service interface (DataService), getButton3Data returns RandomNumber. So, use RandomNumber here.



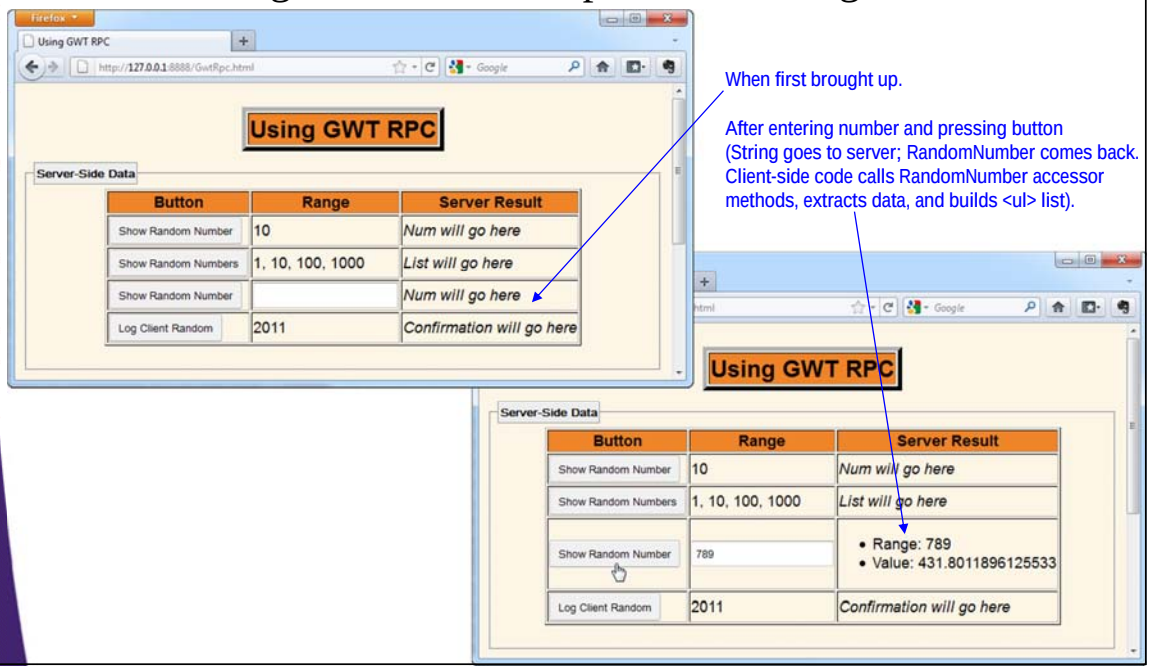
# Testing in Development Mode

- R-click project name and Run As → Web Application
  - Or, click project name in Eclipse. Press Run button at top



# Testing in Production Mode

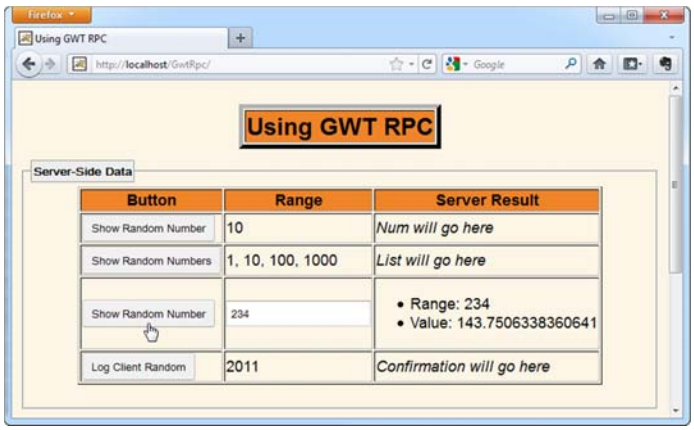
– Did Google → GWT Compile, then changed URL






# Testing in Deployed Mode

- **Deployed to Tomcat on localhost**
  - Copied "war" folder and renamed to "GwtRpc"
  - Or, made GwtRpc.zip, put *contents* of "war" (not "war" itself) inside it, and then renamed to GwtRpc.war.
    - Either way, deployed result to Web apps folder of server



# Example 4: Sending Serialized Object to the Server



## Example Overview

- **Goal**
  - Press a button, construct a custom RandomNumber on the client, send it to the server
  - Server method returns 'void' and logs number
    - Uses regular servlet logging capabilities
  - On the client, use onSuccess as acknowledgement of receipt and display client-side value
- **Points**
  - Passing data to server-side methods
    - GWT manages the network marshalling/unmarshalling
      - Even though client-side code is really in JavaScript!
  - Returning Void data type to client

# Step 1: Main Service Interface

```
package coreservlets.client;

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

@RemoteServiceRelativePath("data-service")
public interface DataService extends RemoteService {
  public String getButton1Data();
  public String[] getButton2Data();
  public RandomNumber getButton3Data(String range);
  public void logClientRandom(RandomNumber clientRandom);
}
```



# Step 2: Callback Version of Service Interface

```
package coreservlets.client;

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

public interface DataServiceAsync {
  public void getButton1Data(AsyncCallback<String> callback);
  public void getButton2Data(AsyncCallback<String[]> callback);
  public void getButton3Data(String range,
                             AsyncCallback<RandomNumber> callback);
  public void logClientRandom(RandomNumber clientRandom,
                              AsyncCallback<Void> callback);

}
```

Added AsyncCallback<Void> to end of argument list. void is return type of original (non-callback) version of logClientRandom.

Return type of void stays the same

# Step 3: Service Servlet

```
public class DataServlet extends RemoteServiceServlet
                         implements DataService {
  …

 public RandomNumber getButton3Data(String rangeString) {
    return (new RandomNumber(rangeString));
  }


  public RandomNumber getButton3Data(String rangeString) {
    return(new RandomNumber(rangeString));
  }

  public void logClientRandom(RandomNumber clientRandom) {
    log("Random number sent from client is: " +
         clientRandom.getValue());
    System.out.println("Random number '" +
         clientRandom.getValue() + "' was logged.");
  }
}
```



# Step 4: Service Proxy

```
public class GwtRpc implements EntryPoint {
  private DataServiceAsync serviceProxy;
  private HTML label1, label2, label3, label4;
  private TextBox rangeBox;

  public void onModuleLoad() {  // Main entry point
    serviceProxy = GWT.create(DataService.class);
    …
    Button button4 = new Button("Log Client Random");
    label4 = new HTML("<i>Confirmation will go here</i>");
    RootPanel.get("button4").add(button4);
    RootPanel.get("label4").add(label4);
    button4.addClickHandler(new Button4Handler());
  }
    …
  } … }
```

Service proxy definition unchanged from previous examples.
But code above also shows GUI controls specific to this example.

# Step 5: Client-Side Callback Objects

```
public class GwtRpcApplication implements EntryPoint {
  …
 private class Button4Handler implements ClickHandler {
   public void onClick(ClickEvent event) {
      String range = RootPanel.get("range4")
         .getElement().getInnerText();
      RandomNumber clientRandom = new RandomNumber(range);
      serviceProxy.logClientRandom(clientRandom,
                              new LogCallback(clientRandom));
    }
  }
}
```



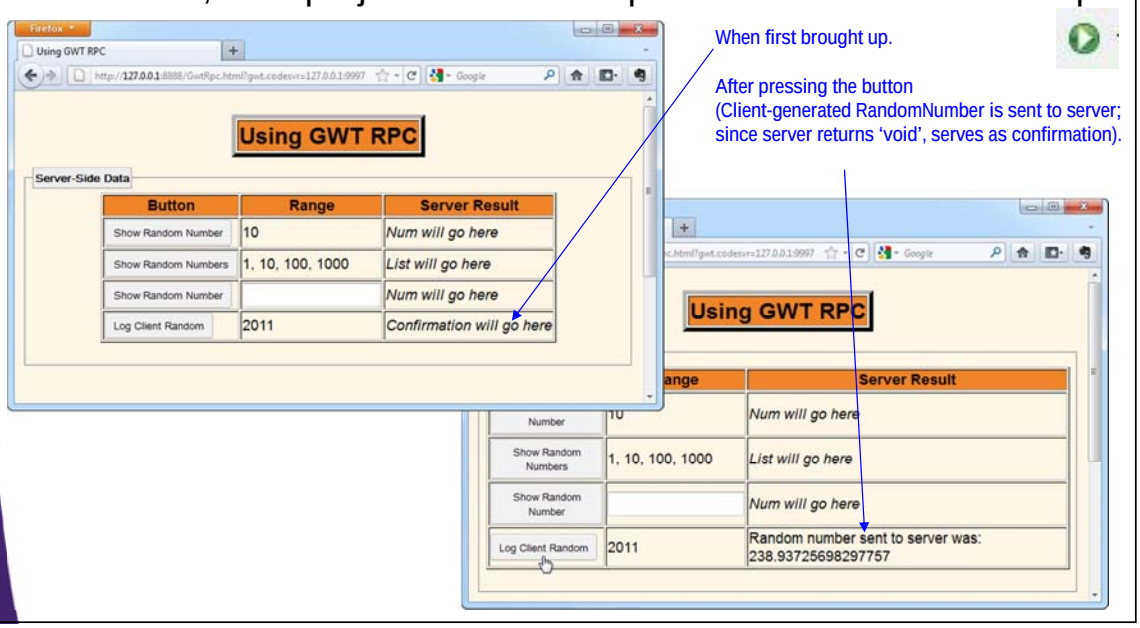
# Step 5: Client-Side Callback Objects (Continued)

```
private class LogCallback implements AsyncCallback<Void> {
   RandomNumber clientRandom;

   public LogCallback(RandomNumber clientRandom) {
     this.clientRandom = clientRandom;
   }

   public void onFailure(Throwable caught) {
     Window.alert("Unable to log message to the server.");
   }

   public void onSuccess(Void result) {
     String message = "Random number sent to server was: "
         + clientRandom.getValue();
     label4.setHTML(message);
   }
 }
```

In main service interface (DataService), logClientRandom returns void. So, use Void data type here.

# Testing in Development Mode

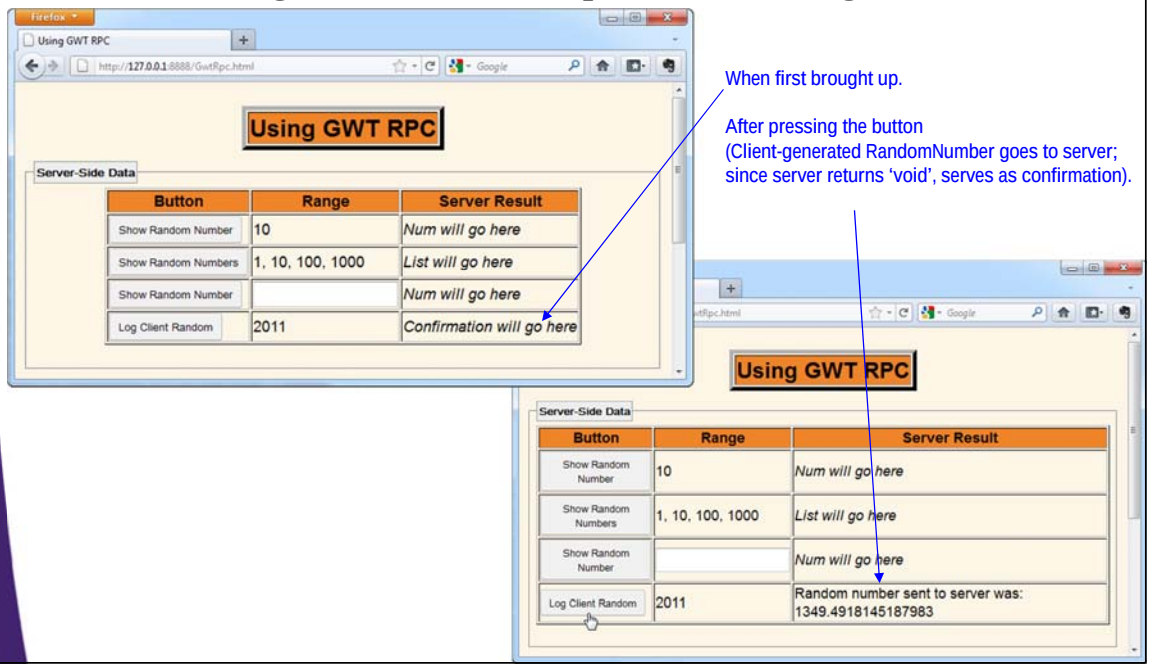
- R-click project name and Run As → Web Application
  - Or, click project name in Eclipse. Press Run button at top





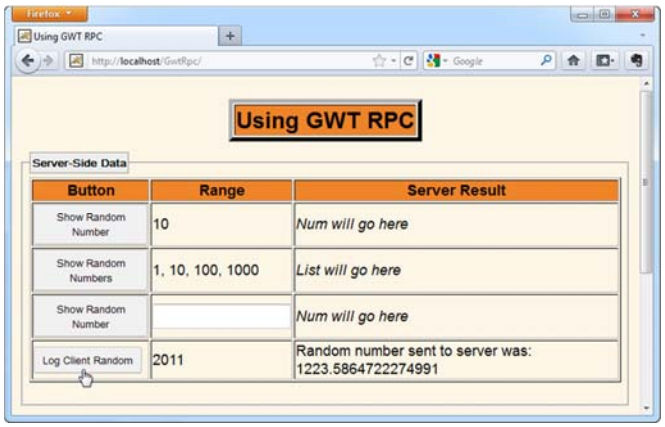

# Testing in Production Mode

- Did Google → GWT Compile, then changed URL



# Testing in Deployed Mode

- **Deployed to Tomcat on localhost**
  - Copied "war" folder and renamed to "GwtRpc"
  - Or, made GwtRpc.zip, put *contents* of "war" (not "war" itself) inside it, and then renamed to GwtRpc.war.
    - Either way, deployed result to Web apps folder of server








# Handling Asynchronous Methods

# Issue

- **Service proxy methods are automatically asynchronous**
  - Even though you see no explicit threading code, serviceProxy.someMethod returns immediately and runs in the background
- **So, onSuccess must do all the work directly**
  - This is obvious to JavaScript programmers who do normal Ajax programming.
    - But, Java programmers can be confused because they don't see multithreading cues like Runnable or Thread.



# Example: Shared Code

```
public class SomeApp implements EntryPoint {
  private SomeServiceAsync serviceProxy;
  private HTML label;
  private String message = "no message yet";

  public void onModuleLoad() {
    serviceProxy = GWT.create(SomeService.class);
    Button button = new Button("Click Me");
    label = new HTML("<i>Message will go here</i>");
    ...
    button.addClickHandler(new ButtonHandler());
    ...
  }
```

# Example: Wrong Approach

```
private class ButtonHandler implements ClickHandler {
  public void onClick(ClickEvent event) {
    serviceProxy.getMessage(new ButtonCallback());
    label.setHTML(message);
  }
}

private class ButtonCallback
        implements AsyncCallback<String> {
  public void onSuccess(String message) {
    this.message = message;
  }

  public void onFailure(Throwable caught) {
    Window.alert("Unable to get data from server.");
  }
}
```

Instance variable.

Even if server returns "some cool message", label shows "no message yet". Because getMessage returns immediately and then runs in the background, line in red above runs before the message string has changed.



# Example: Right Approach

```
private class ButtonHandler implements ClickHandler {
  public void onClick(ClickEvent event) {
    serviceProxy.getMessage(new ButtonCallback());
  }
}

private class ButtonCallback
        implements AsyncCallback<String> {
  public void onSuccess(String message) {
    label.setHTML(message);
  }

  public void onFailure(Throwable caught) {
    Window.alert("Unable to get data from server.");
  }
}
```

No need for instance variable.

Even if it takes server 10 seconds to return "some cool message", label will show "some cool message" once the server returns. Line in red above is guaranteed to run after the message string has been returned.

# Wrap-Up



# Summary

- **Define main service interface**
  - Extend RemoteService, use @RemoteServiceRelativePath
- **Define Async version of service interface**
  - BlahServiceAsync: add AsyncCallback<Type> to end of args
- **Make data service servlet**
  - Extend RemoteServiceServlet, implement service interface
  - Supply url-pattern in web.xml that matches relative path
- **Create service proxy**
  - Call GWT.create(*YourServiceInterfaceName*.class)
- **Define client-side callback objects**
  - With onSuccess and onFailure
- **Test**
  - Development mode, production mode, deployed mode

# Questions?