The Google Web Toolkit: Using GWT RPC to Access Server-Side Data

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

Topics in This Section

- **Idea of RPC**
- **Development process**
  - Defining client-side data service interfaces
  - Making a data service servlet
  - Specifying the data source
  - Defining client-side callback classes
- **Testing in hosted mode**
  - To run using bundled server and GWT browser
- **Testing in Web mode**
  - To run using bundled server and regular browser
- **Deploying**
  - To run using regular Java server and regular browser

RPC: Big Idea

- **Write regular methods on server, not 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
  - Pass arbitrary arguments and get arbitrary results
    - Custom form of serialization handles all the parsing
- **Note**
  - GWT does not require RPC, but it is a tremendous simplification
    - If you have existing server data sources, you can still access them if they are on same server as GWT app

RPC Data Types

- **Server methods can return complex types**
  - Packing and unpacking handled automatically
    - Which is no small feat considering that the 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, and Vector
      - Those are as of GWT 1.3. GWT 1.4 may add more.
  - Custom classes that implement IsSerializable
    - Or just Serializable in GWT 1.4
  - Arrays containing any of the above types

GWT-RPC Development Steps

1. **Define client-side data service interfaces**
   - Let client call server methods without explicit HTTP
     - E.g., interfaces: DataService and DataServiceAsync
2. **Make a data service servlet**
   - Supply server information in response to Ajax event
     - Extend RemoteServiceServlet, implement interface
3. **Specify the data source**
   - Define the URL of the data service servlet
     - In Java code, give full path to service entry point
     - In BlahApp.gwt.xml, give relative path to servlet
4. **Define client-side callback classes**
   - Link the client-side interfaces to the server-side code
     - Implement AsynCallback with onSuccess and onFailure

Example 1: Getting a Random Number from the Server

Example Overview

- **Goal**
  - Press a button, call a method on the server that returns a String.
  - Display the String on the client inside a table cell
- **Point**
  - General process of calling to server
- **Assumed project setup already done**
  - Ran projectCreator
  - Ran applicationCreator
  - Imported into Eclipse
  - Removed auto-generated sample code within onModuleLoad

Defining Client-Side Data Interfaces

- **Define main interface**
  - Extend RemoteService, list server methods
    - Must be in client package
    - Cannot use Java 5+ types (no generic types or enums)
- **Define callback interface**
  - Named *MainInterfaceName*Async
    - I.e., name of callback interface must be same name as main interface, but with "Async" at the end
  - All methods take one extra arg of type AsyncCallback and return void
    - I.e., if main interface (e.g., FooService) defined
      public String bar(int n);
      callback interface (e.g., FooServiceAsync) would define
      public void bar(int n, AsyncCallback callback);

Defining Client-Side Data Interfaces: Main Interface

```
package coreservlets.client;

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

public interface DataService extends RemoteService {
  public String getButton1Data();
  public String[] getButton2Data();
  public RandomNumber getButton3Data(String range);
}
```

Defining Client-Side Data Interfaces: Callback Interface

```
package coreservlets.client;

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

public interface DataServiceAsync {
  public void getButton1Data(AsyncCallback callback);
  public void getButton2Data(AsyncCallback callback);
  public void getButton3Data(String range,
                             AsyncCallback callback);
}
```

Since main interface is called DataService, this must be called DataService*Async*.

Compared to method from main interface:
- Add AsyncCallback to end of argument list.
- Make return type void.

Making a Data Service Servlet

- **Extend RemoteServiceServlet**
  - Put in server package (or any package other than "client")
- **Implement main 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
    - You are allowed to use Java 5 or Java 6 features in the body of the code
      - Including in the body of the methods from the interface
      - Whatever version your server is running
  - Remember the interface is in client package
    - So you need to import package...client.MainInterfaceName

Making a Data Service Servlet

```
package coreservlets.server;

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

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

  ...
}
```

Client-side interface from previous step.

Java 5 or Java 6 features permitted.
(Server-side code does *not* get translated into JavaScript.)

Specifying the Data Source

- **Main idea**
  - Create a proxy to communicate with the server
  - The steps are a bit cumbersome, but you do it the same way every time, so just cut/paste from working examples
    - All code from this tutorial is online at coreservlets.com
- **Steps**
  - Call GWT.create on your main interface and cast the result to your callback interface

```
DataServiceAsync dataService =
  (DataServiceAsync)GWT.create(DataService.class);
```

  - Cast the result to ServiceDefTarget

```
ServiceDefTarget endpoint = (ServiceDefTarget)dataService;
```

  - Specify entry point URL (/moduleName/servicename)

```
endpoint.setServiceEntryPoint("/coreservlets.RPCApp/DataService");
```

Specifying the Data Source

```
public class RPCApp implements EntryPoint {
  private HTML label1, label2, label3;
  private TextBox rangeBox;
  private DataServiceAsync dataService;

  public void onModuleLoad() {
    ...
    dataService = getDataService();
  }
  ...
  private DataServiceAsync getDataService() {
    DataServiceAsync dataService =
      (DataServiceAsync)GWT.create(DataService.class);
    ServiceDefTarget endpoint =
      (ServiceDefTarget)dataService;
    endpoint.setServiceEntryPoint
                     ("/coreservlets.RPCApp/DataService");
    return(dataService);
  }
}
```

Defining Client-Side Callback Classes

- **Create a class that implements AsyncCallback**
  - onSuccess method invoked when call to server is successful. Returns an Object that represents the return value from the server.
    - Cast it to the actual return type
  - onFailure method invoked when call to server fails
  - As with event handling in Swing or the AWT, you often use inner classes so that the methods can easily access data from main app
- **Call interface method on data service**
  - Pass the expected args *plus the callback object*
  - You usually do this from a GUI event handler

Defining Client-Side Callback Classes

```
public class RPCApp implements EntryPoint {
  ...
  private class Button1Listener implements ClickListener {
    public void onClick(Widget sender) {
      dataService.getButton1Data(new Button1Callback());
    }
  }

  private class Button1Callback implements AsyncCallback {
    public void onSuccess(Object serverData) {
      String result = (String)serverData;
      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 client-side interface (DataService), getButton1Data takes no args.
In callback interface (DataServiceAsync), getButton1Data takes one extra arg: a callback.

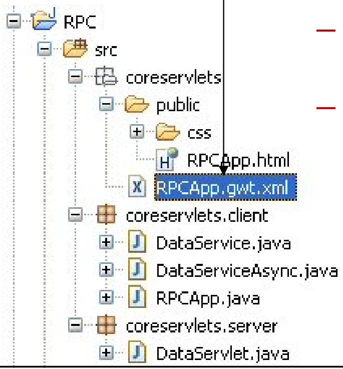
In main client-side interface (DataService), getButton1Data returns a String. So, cast result to String.

Testing in Hosted Mode

- **Recall from introductory section**
  - Client runs purely in Java
    - Not translated into JavaScript
  - Uses GWT-bundled browser that interprets HTML and Java
  - Very compatible with how regular browser will work with JavaScript
- **New addition**
  - Server-side code will run in Tomcat version that is bundled with GWT

Setting Up Hosted Mode

- **Only extra step is to define relative URL that the embedded server (Tomcat) will use**
  - In Java code, we use /moduleName/serviceName
    - Java Web app developers often call the /moduleName part "the Web app prefix" or "the context root" or "the context path"
  - So, relative URL will be /serviceName (don't forget the leading /)
- **Define the servlet path in BlahApp.gwt.xml**



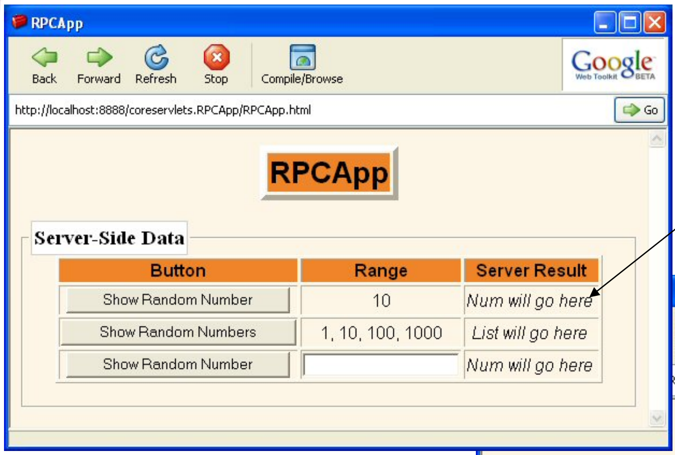
  - Under src/packageName/public/
  - Code to add:

```
<servlet path="/DataService"
         class="coreservlets.server.DataServlet"/>
```

URL relative to module. Java code uses /coreservlets.RPCApp/DataService, so this gives just /DataService.

Fully qualified servlet class name.

Testing in Hosted Mode (Run > Run > Java Application> RPCApp)



http://localhost:8888/coreservlets.RPCApp/RPCApp.html

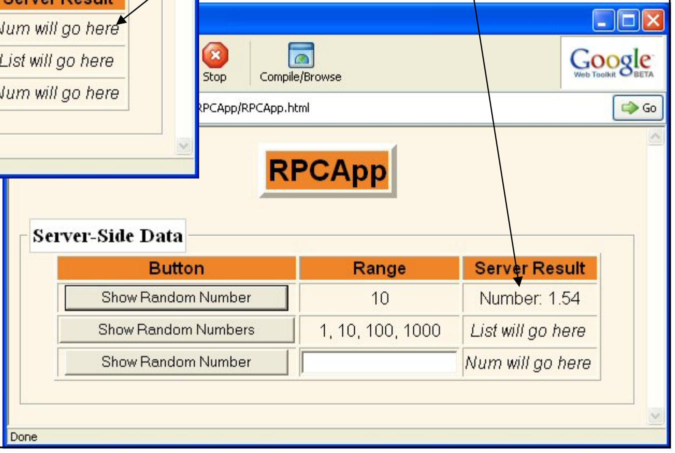
RPCApp

Server-Side Data

Button	Range	Server Result
Show Random Number	10	Num will go here
Show Random Numbers	1, 10, 100, 1000	List will go here
Show Random Number		Num will go here

When app first comes up.

After pressing button.



RPCApp/RPCApp.html

RPCApp

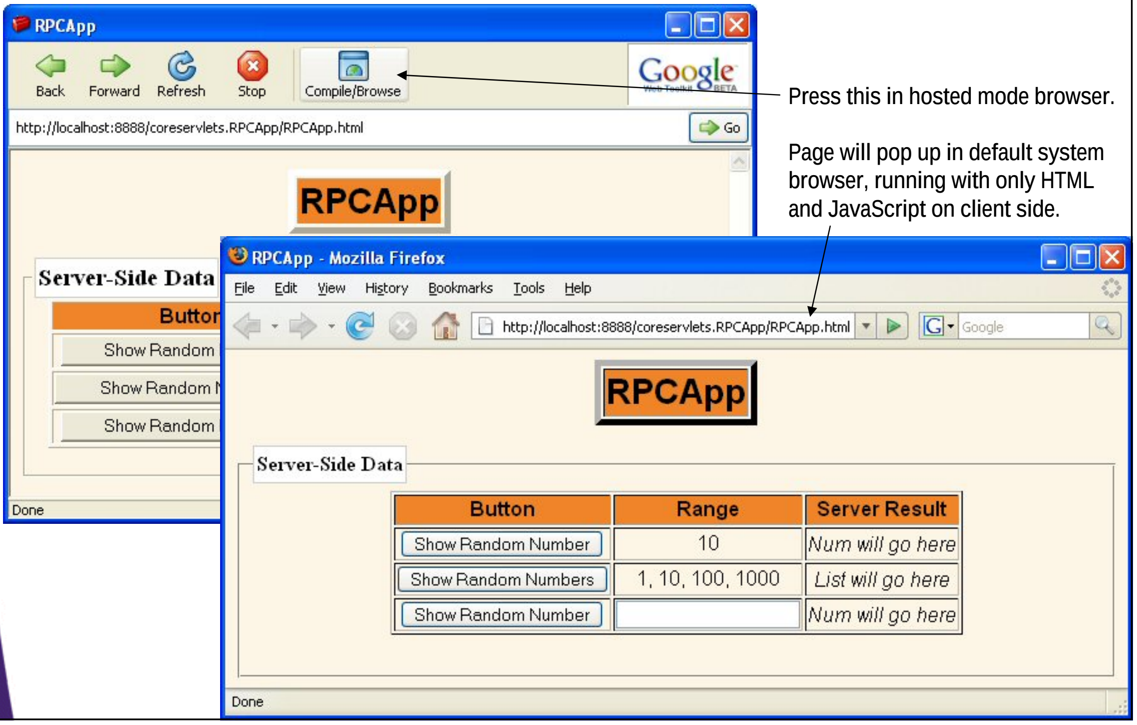
Server-Side Data

Button	Range	Server Result
Show Random Number	10	Number: 1.54
Show Random Numbers	1, 10, 100, 1000	List will go here
Show Random Number		Num will go here

Testing in Web Mode

- **Idea**
  - Running in Web mode means running in a regular browser with all client-side Java code converted to JavaScript
- **Steps**
  - Run the application in hosted mode
  - Click the "Compile/Browse" button at top of hosted-mode browser. Result:
    - Creates a folder in your project called www
      - Note: in Eclipse, right-click on main project name and select "Refresh" to see the newly created directories and files
    - Creates a folder inside www matching module name
      - E.g., coreservlets.RPCApp
    - Puts HTML and translated JavaScript files in this folder
    - Loads the main HTML file in your default browser

Testing in Web Mode

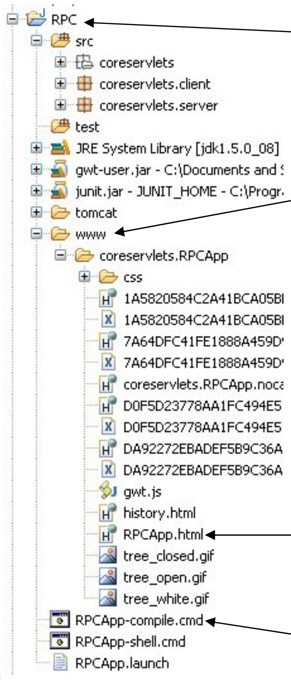


Button	Range	Server Result
Show Random Number	10	*Num will go here*
Show Random Numbers	1, 10, 100, 1000	*List will go here*
Show Random Number		*Num will go here*

Press this in hosted mode browser.

Page will pop up in default system browser, running with only HTML and JavaScript on client side.

Testing in Web Mode: Resultant Files



If you want to see the files that get built for Web mode, right-click here and select Refresh.

This folder is built when you pressed Compile/Browse.
Notes:
- To see these files in Eclipse, right-click on main project name (RPC at top) and choose Refresh
- Don't edit files in this folder. They will be overwritten next time you rebuild the client-side code.

You can navigate to the www folder outside Eclipse and load RPCApp.html in a browser. This will let you run all client-side code, but not server-side code.

You can also build the www folder and all client-side code by double-clicking this file. You can do so from within Eclipse.

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
    - Basically, nothing extra is required. 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. Far simpler than in most Ajax tools:
      - You are building the String in Java instead of JavaScript
      - You are dealing with a normal array instead of XML

Client-Side Data Interfaces: Main Interface

- **This is the same class (DataService) already shown in previous section.**
  - Just added one method to it

```
package coreservlets.client;

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

public interface DataService extends RemoteService {
  public String getButton1Data();
  public String[] getButton2Data();
  public RandomNumber getButton3Data(String range);
}
```

Client-Side Data Interfaces: Callback Interface

- **This is the same class (DataServiceAsync) already shown in previous section.**
  - Just added one method to it

```
package coreservlets.client;

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

public interface DataServiceAsync {
  public void getButton1Data(AsyncCallback callback);
  public void getButton2Data(AsyncCallback callback);
  public void getButton3Data(String range,
                             AsyncCallback callback);
}
```

Data 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);
  }
  ...
}
```

Specifying the Data Source (No Change from Previous Example)

```
public class RPCApp implements EntryPoint {
  private HTML label1, label2, label3;
  private TextBox rangeBox;
  private DataServiceAsync dataService;

  public void onModuleLoad() {
    ...
    dataService = getDataService();
  }
  ...
  private DataServiceAsync getDataService() {
    DataServiceAsync dataService =
      (DataServiceAsync)GWT.create(DataService.class);
    ServiceDefTarget endpoint =
      (ServiceDefTarget)dataService;
    endpoint.setServiceEntryPoint
                     ("/coreservlets.RPCApp/DataService");
    return(dataService);
  }
}
```

Client-Side Callback Classes

```
public class RPCApp implements EntryPoint {
  ...

  private class Button2Listener implements ClickListener {
    public void onClick(Widget sender) {
      dataService.getButton2Data(new Button2Callback());
    }
  }
```

This is the event handler for the second pushbutton.

In main client-side interface (DataService), getButton2Data takes no args.
In callback interface (DataServiceAsync), getButton2Data takes one extra arg: a callback.

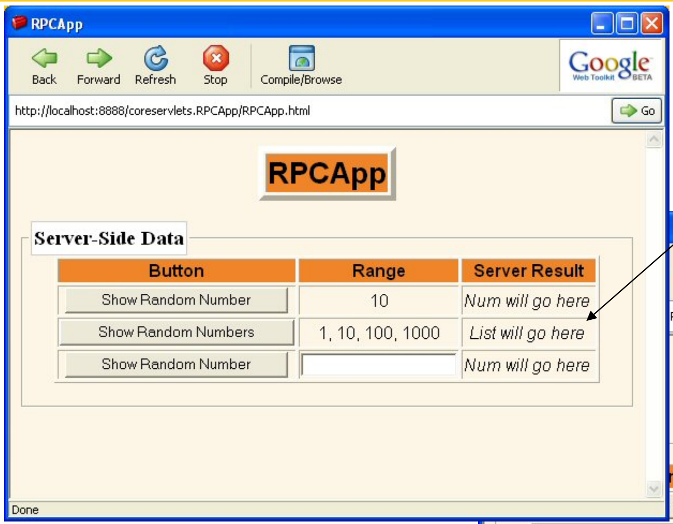
Client-Side Callback Classes (Continued)

```
private class Button2Callback implements AsyncCallback {
  public void onSuccess(Object serverData) {
    String[] listItems = (String[])serverData;
    String result = "<ul>\n";
    for(int i=0; i<listItems.length; i++) {
      result = result + "<li>" + listItems[i] + "</li>\n";
    }
    result = result + "</ul>";
    label2.setHTML(result);
  }

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

In main client-side interface (DataService), getButton2Data returns an array of String. So, cast result to String[]. GWT lets you send primitives, wrapper types, arrays, a subset of standard Java objects (ArrayList, Date, HashMap, HashSet, String, and Vector), and custom classes that implement the GWT IsSerializable interface. Network marshalling handled automatically.

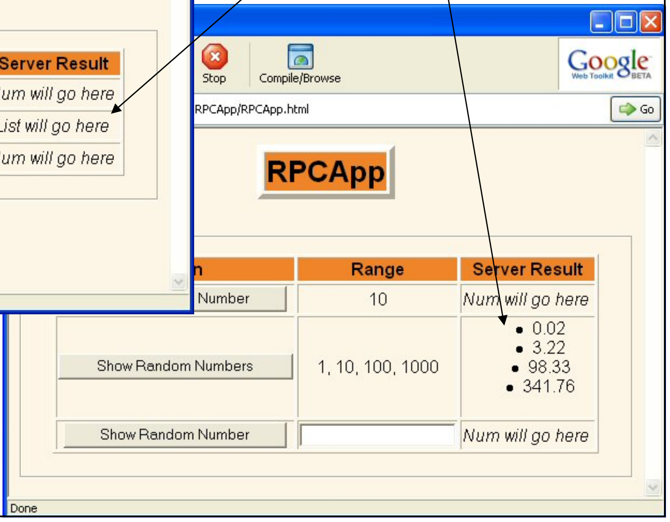
Testing in Hosted Mode
(Run > Run > Java Application> RPCApp)



Button	Range	Server Result
Show Random Number	10	Num will go here
Show Random Numbers	1, 10, 100, 1000	List will go here
Show Random Number		Num will go here

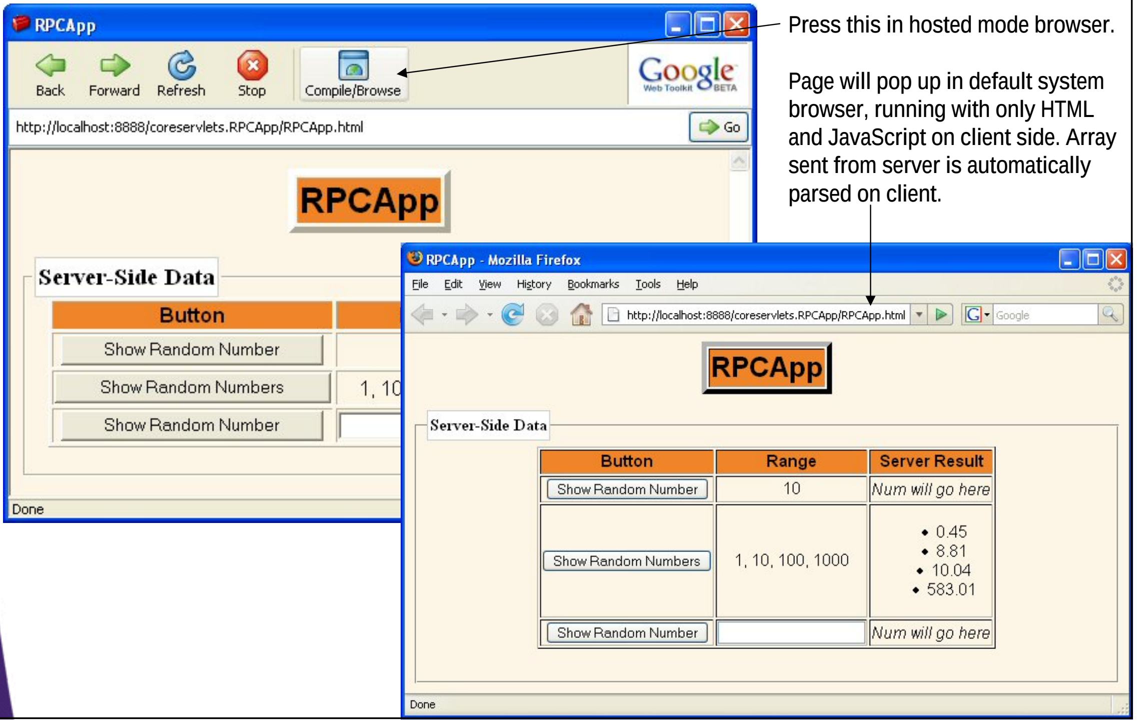
When app first comes up.

After pressing button.



n	Range	Server Result
Number	10	Num will go here
Show Random Numbers	1, 10, 100, 1000	• 0.02 • 3.22 • 98.33 • 341.76
Show Random Number		Num will go here

Testing in Web Mode



RPCApp

Server-Side Data

Button	Range	Server Result
Show Random Number	10	*Num will go here*
Show Random Numbers	1, 10, 100, 1000	• 0.45 • 8.81 • 10.04 • 583.01
Show Random Number		*Num will go here*

Press this in hosted mode browser.

Page will pop up in default system browser, running with only HTML and JavaScript on client side. Array sent from server is automatically parsed on client.

Example 3: Getting a Serialized Object from the Server

Example Overview

- **Goal**
  - Press a button, pass a value to a server side method
  - The server-side method returns a custom RandomNumber object
  - On the client, extract data from the RandomNumber by calling getting methods
- **Points**
  - Passing data to server-side methods
  - Returning serializable data types to client
    - You implement the GWT IsSerializable interface
    - GWT manages the network marshalling
      - Even though client-side code is really in JavaScript!

Client-Side Data Interfaces: Main Interface

- **This is the same class (DataService) already shown in previous section.**
  - Just added one method to it

```
package coreservlets.client;

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

public interface DataService extends RemoteService {
  public String getButton1Data();
  public String[] getButton2Data();
  public RandomNumber getButton3Data(String range);
}
```

This is a custom class that implements the GWT IsSerializable interface.

Client-Side Data Interfaces: Callback Interface

- **This is the same class (DataServiceAsync) already shown in previous section.**
  - Just added one method to it

```
package coreservlets.client;

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

public interface DataServiceAsync {
  public void getButton1Data(AsyncCallback callback);
  public void getButton2Data(AsyncCallback callback);
  public void getButton3Data(String range,
                             AsyncCallback callback);
}
```

Data 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));
  }
}
```

Specifying the Data Source (No Change from Previous Example)

```
public class RPCApp implements EntryPoint {
  private HTML label1, label2, label3;
  private TextBox rangeBox;
  private DataServiceAsync dataService;

  public void onModuleLoad() {
    ...
    dataService = getDataService();
  }
  ...
  private DataServiceAsync getDataService() {
    DataServiceAsync dataService =
      (DataServiceAsync)GWT.create(DataService.class);
    ServiceDefTarget endpoint =
      (ServiceDefTarget)dataService;
    endpoint.setServiceEntryPoint
                 ("/coreservlets.RPCApp/DataService");
    return(dataService);
  }
}
```

GWT Serializable Classes

- **Classes must implement IsSerializable**
  - In com.google.gwt.user.client.rpc package
  - This interface is just a marker
    - No required methods
    - Must have zero-argument constructor
- **Server-side code uses this class normally**
  - A regular Java class
- **Client side Java code casts return value of callback (in onSuccess) to this type**
  - Then uses it like a normal Java class, even though in Web mode it really becomes JavaScript
- **GWT 1.4 lets you use java.io.Serializable**
  - So classes that already implement Serializable don't have to change
    - E.g., Hibernate data objects or RMI classes
    - But it is still GWT serialization, not regular Java serialization

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

```
package coreservlets.client;
import com.google.gwt.user.client.rpc.*;

public class RandomNumber implements IsSerializable {
  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 IsSerializable interface contains no methods, so classes that implement IsSerializable need no special methods. Subclasses of serializable classes are automatically serializable.

Client-Side Callback Classes

```
public class RPCApp implements EntryPoint {
  ...

  private class Button3Listener implements ClickListener {
    public void onClick(Widget sender) {
      String range = rangeBox.getText();
      dataService.getButton3Data(range,
                                 new Button3Callback());
    }
  }
```

This is the event handler for the third pushbutton.

The server-side method expects a String as an argument.
Get the String from a GWT TextBox (i.e., an HTML textfield).

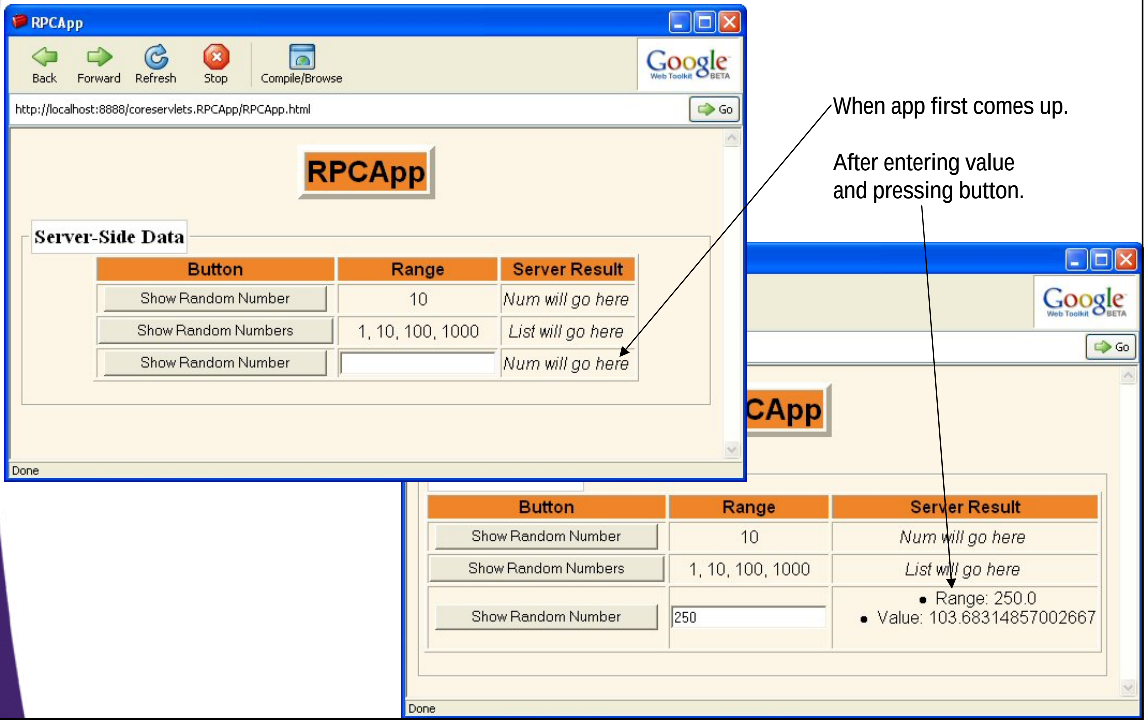
In main client-side interface (DataService), getButton3Data takes one arg (range).
In callback interface (DataServiceAsync), getButton3Data takes one extra arg: a callback.

Client-Side Callback Classes (Continued)

```
  private class Button3Callback implements AsyncCallback {
    public void onSuccess(Object serverData) {
      RandomNumber number = (RandomNumber)serverData;
      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);
    }
```

In main client-side interface (DataService), getButton3Data returns a RandomNumber. So, cast result to RandomNumber. GWT lets you send primitives, wrapper types, arrays, a subset of standard Java objects (ArrayList, Date, HashMap, HashSet, String, and Vector), and custom classes that implement the GWT IsSerializable interface. Network marshalling handled automatically.

Testing in Hosted Mode
(Run > Run > Java Application> RPCApp)



RPCApp

Server-Side Data

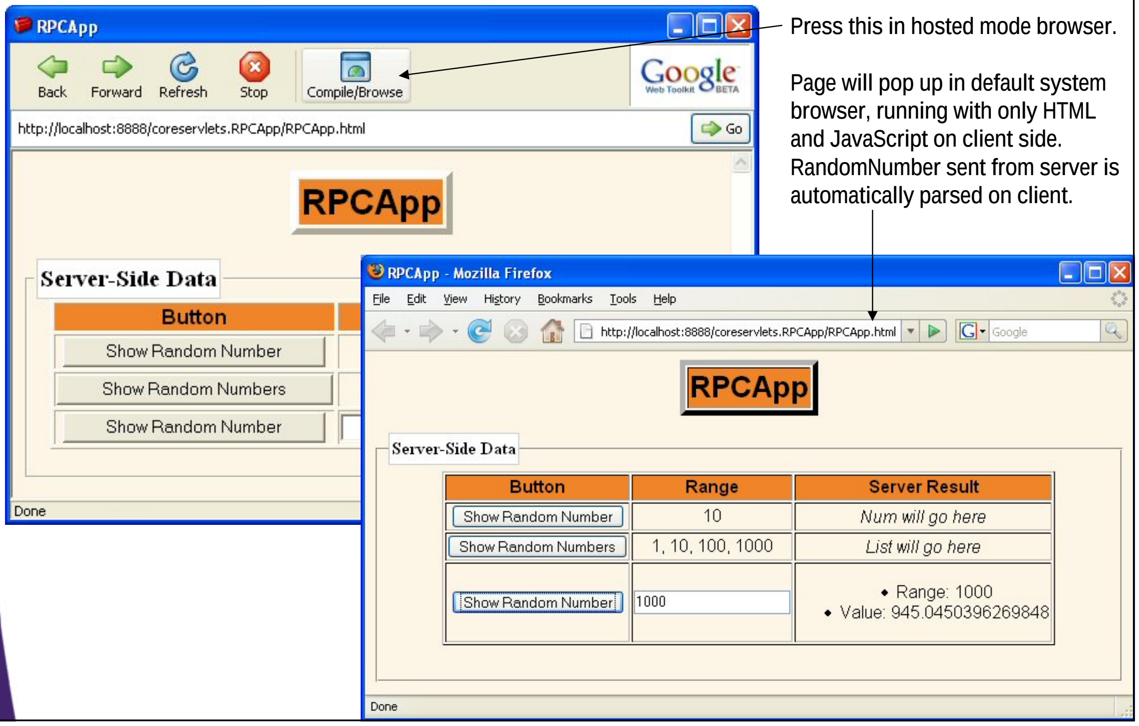
Button	Range	Server Result
Show Random Number	10	*Num will go here*
Show Random Numbers	1, 10, 100, 1000	*List will go here*
Show Random Number		*Num will go here*

Button	Range	Server Result
Show Random Number	10	*Num will go here*
Show Random Numbers	1, 10, 100, 1000	*List will go here*
Show Random Number	250	• Range: 250.0 • Value: 103.68314857002667

When app first comes up.

After entering value and pressing button.

Testing in Web Mode



RPCApp

Server-Side Data

Button	Range	Server Result
Show Random Number	10	*Num will go here*
Show Random Numbers	1, 10, 100, 1000	*List will go here*
Show Random Number	1000	• Range: 1000 • Value: 945.0450396269848

Press this in hosted mode browser.

Page will pop up in default system browser, running with only HTML and JavaScript on client side. RandomNumber sent from server is automatically parsed on client.

Handling Asynchronous Methods

Issue

- **Callback methods (onSuccess and onFailure) are automatically asynchronous**
  – Even though you see no explicit threading code, onSuccess and onFailure return immediately and run in the background
- **So, you cannot perform side effects that code after the callback depends on**
  – Instead, perform side effects from within the method
  – This is particularly important for the onSuccess method

Example: Common Code

```
public class RPCApp implements EntryPoint {
  private HTML label1;
  private String message = "no message yet";

  public void onModuleLoad() {
    Button button1 = new Button("Click Me");
    label1 = new HTML("<i>Message will go here</i>");
    ...
    button1.addClickListener(new Button1Listener());
    ...
  }
```

Example: Wrong Approach

```
private class Button1Listener implements ClickListener {
  public void onClick(Widget sender) {
    dataService.getMessage(new Button1Callback());
    label1.setHTML(message);
  }
}

private class Button1Callback implements AsyncCallback {
  public void onSuccess(Object serverData) {
    message = (String)serverData;
  }

  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 onSuccess returns immediately and then runs in the background, line in red above runs before the message string has changed.

Example: Right Approach

```
private class Button1Listener implements ClickListener {
  public void onClick(Widget sender) {
    dataService.getMessage(new Button1Callback());
  }
}

private class Button1Callback implements AsyncCallback {
  public void onSuccess(Object serverData) {
    String message = (String)serverData;
    label1.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 still shows "some cool message" once the server returns. Line in red above is guaranteed to run after the message string has been set.

Deploying GWT-RPC Applications

Issue

- **What testing in Web mode does:**
  - Verifies that JavaScript (translated Java code) and HTML work properly in normal browser
    - You can cut/paste URL to multiple browsers to test cross-browser compatibility
  - Verifies that communication with server and data serialization work correctly
    - Uses embedded version of Tomcat bundled with GWT
- **What testing in Web mode does not do:**
  - Does not deploy to regular Java server
    - Does not build WAR file with correct JAR files included
    - Does not set url-pattern of servlets
  - Does not verify that URLs work on deployment server
  - Does not change name of HTML file

Steps to Deploying on External Server

- **Test in Web mode as shown previously**
  - This will make sure latest client code is translated
    - Be sure to R-click on project name and select Refresh
  - Also, since deployment is clumsy, you want to deploy at the very end after all testing and updating is finished
- **Create a regular Eclipse Web project**
  - Copy files from www/package.AppName to WebRoot (MyEclipse) or WebDeploy (Eclipse)
  - Optionally, rename AppName.html to index.html
  - Add gwt-servlet.jar to WEB-INF/lib of new app
  - Copy Java code from src in old app to src in new app
  - Define context path of new app to be /package.AppName
  - Define url-pattern of servlet to be /data-service-name
- **Deploy new project normally**

Creating Eclipse Deployment Project

After testing in Web mode, click here and select Refresh.

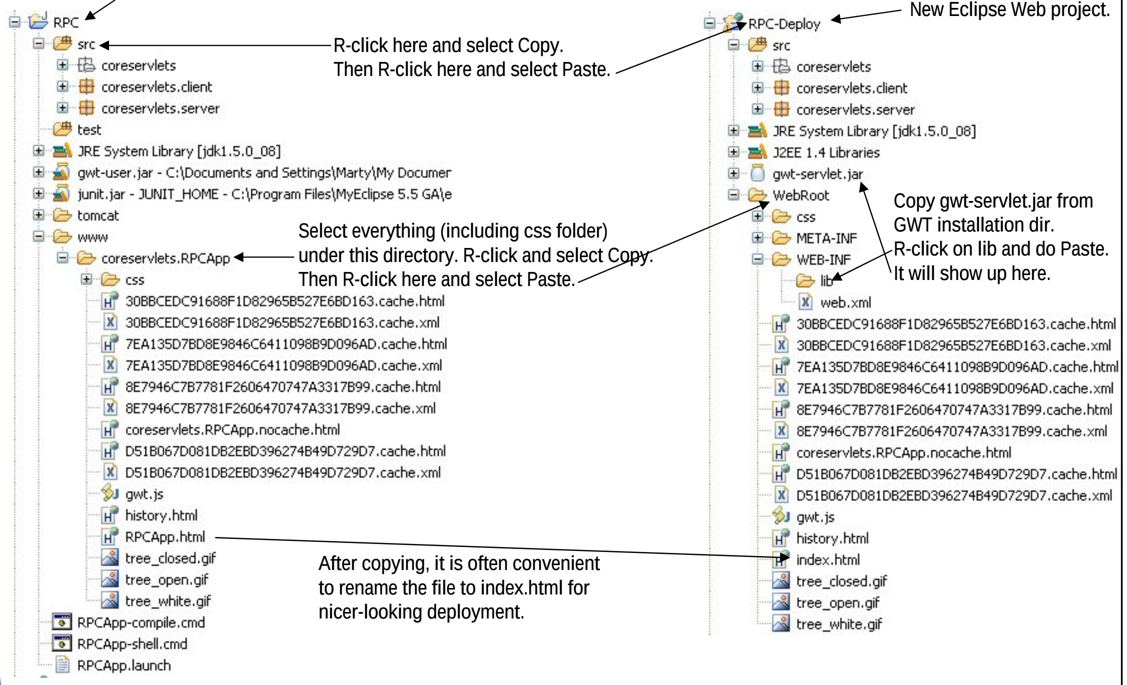
R-click here and select Copy. Then R-click here and select Paste.

New Eclipse Web project.

Select everything (including css folder) under this directory. R-click and select Copy. Then R-click here and select Paste.

Copy gwt-servlet.jar from GWT installation dir. R-click on lib and do Paste. It will show up here.

After copying, it is often convenient to rename the file to index.html for nicer-looking deployment.



```
RPC
  src
    coreservlets
    coreservlets.client
    coreservlets.server
  test
  JRE System Library [jdk1.5.0_08]
  gwt-user.jar - C:\Documents and Settings\Marty\My Documer
  junit.jar - JUNIT_HOME - C:\Program Files\MyEclipse 5.5 GA\e
  tomcat
  www
    coreservlets.RPCApp
      css
      30BBCEDC91688F1D82965B527E6BD163.cache.html
      30BBCEDC91688F1D82965B527E6BD163.cache.xml
      7EA135D7BD8E9846C6411098B9D096AD.cache.html
      7EA135D7BD8E9846C6411098B9D096AD.cache.xml
      8E7946C7B7781F2606470747A3317B99.cache.html
      8E7946C7B7781F2606470747A3317B99.cache.xml
      coreservlets.RPCApp.nocache.html
      D51B067D081DB2EBD396274B49D729D7.cache.html
      D51B067D081DB2EBD396274B49D729D7.cache.xml
      gwt.js
      history.html
      RPCApp.html
      tree_closed.gif
      tree_open.gif
      tree_white.gif
  RPCApp-compile.cmd
  RPCApp-shell.cmd
  RPCApp.launch
```

```
RPC-Deploy
  src
    coreservlets
    coreservlets.client
    coreservlets.server
  JRE System Library [jdk1.5.0_08]
  J2EE 1.4 Libraries
  gwt-servlet.jar
  WebRoot
    css
    META-INF
    WEB-INF
      lib
      web.xml
    30BBCEDC91688F1D82965B527E6BD163.cache.html
    30BBCEDC91688F1D82965B527E6BD163.cache.xml
    7EA135D7BD8E9846C6411098B9D096AD.cache.html
    7EA135D7BD8E9846C6411098B9D096AD.cache.xml
    8E7946C7B7781F2606470747A3317B99.cache.html
    8E7946C7B7781F2606470747A3317B99.cache.xml
    coreservlets.RPCApp.nocache.html
    D51B067D081DB2EBD396274B49D729D7.cache.html
    D51B067D081DB2EBD396274B49D729D7.cache.xml
    gwt.js
    history.html
    index.html
    tree_closed.gif
    tree_open.gif
    tree_white.gif
```

Eclipse Deployment Project: URL Pattern for RPC Servlet

- **In original Eclipse project**
  - *AppName*.gwt.xml had this:
    <servlet path="/rpc-servlet-address"
    class="package.server.RPCServletClass"/>
  - Code in *AppName*.java had this:
    endpoint.setServiceEntryPoint
    ("/package.AppName/rpc-servlet-address");
- **In new Eclipse project**
  - Edit web.xml and give this:

```
<servlet>
  <servlet-name>some-name</servlet-name>
  <servlet-class>package.server.RPCServletClass</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>some-name</servlet-name>
   <url-pattern>/rpc-servlet-address</rpc-servlet-address>
</servlet-mapping>
```

URL Pattern: Original Project

- **src/coreservlets/public/RPCApp.gwt.xml**

```
<module>
  ...
  <servlet path="/DataService"
           class="coreservlets.server.DataServlet"/>
</module>
```

- **src/coreservlets/client/RPCApp.java**

```
private DataServiceAsync getDataService() {
  DataServiceAsync dataService =
    (DataServiceAsync)GWT.create(DataService.class);
  ServiceDefTarget endpoint =
    (ServiceDefTarget)dataService;
  endpoint.setServiceEntryPoint
              ("/coreservlets.RPCApp/DataService");
  return(dataService);
}
```

URL Pattern: New Project

- **WebRoot/WEB-INF/web.xml**

```
<web-app ...>
  <servlet>
    <servlet-name>RPC-Servlet</servlet-name>
    <servlet-class>
      coreservlets.server.DataServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RPC-Servlet</servlet-name>
    <url-pattern>/DataService</url-pattern>
  </servlet-mapping>

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

Eclipse Deployment Project: Context Root for Deployment

- **In original Eclipse project**
  - Code in *AppName*.java had this:

```
endpoint.setServiceEntryPoint
    ("/package.AppName/rpc-servlet-address");
```

- **In new Eclipse project**
  - Edit project properties and set /package.AppName as the context root. E.g., in MyEclipse, R-click on project name and choose Properties, then Web.

Context Root: Original Project

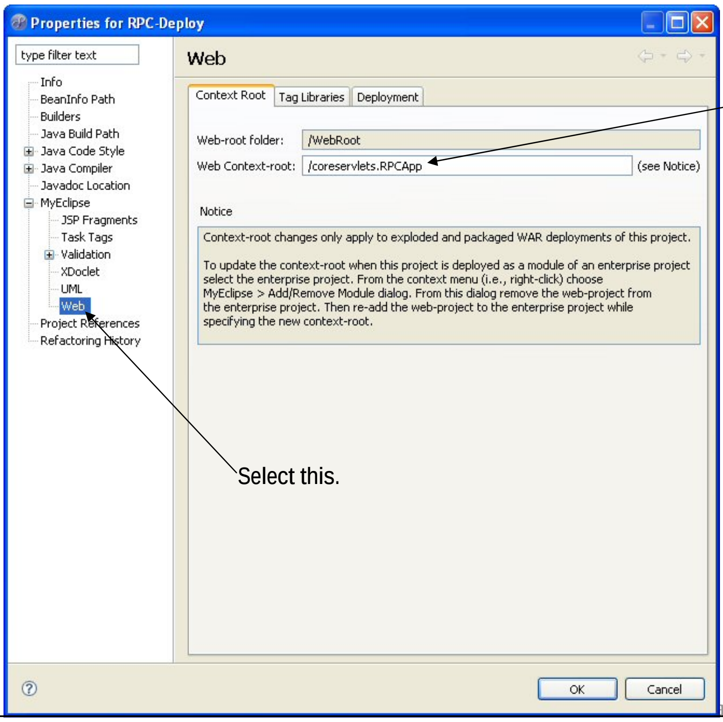
- **src/coreservlets/public/RPCApp.gwt.xml**

```
<module>
  ...
  <servlet path="/DataService"
           class="coreservlets.server.DataServlet"/>
</module>
```

- **src/coreservlets/client/RPCApp.java**

```
private DataServiceAsync getDataService() {
  DataServiceAsync dataService =
    (DataServiceAsync)GWT.create(DataService.class);
  ServiceDefTarget endpoint =
    (ServiceDefTarget)dataService;
  endpoint.setServiceEntryPoint
               ("/coreservlets.RPCApp/DataService");
  return(dataService);
}
```

Context Root: New Project
(R-Click Project Name and Choose "Properties")


Properties for RPC-Deploy

type filter text

- Info
- BeanInfo Path
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
- Javadoc Location
- MyEclipse
  - JSP Fragments
  - Task Tags
  - Validation
  - XDoclet
  - UML
  - Web
- Project References
- Refactoring History

Web

Context Root | Tag Libraries | Deployment

Web-root folder: /WebRoot

Web Context-root: /coreservlets.RPCApp (see Notice)

Notice

Context-root changes only apply to exploded and packaged WAR deployments of this project.

To update the context-root when this project is deployed as a module of an enterprise project select the enterprise project. From the context menu (i.e., right-click) choose MyEclipse > Add/Remove Module dialog. From this dialog remove the web-project from the enterprise project. Then re-add the web-project to the enterprise project while specifying the new context-root.

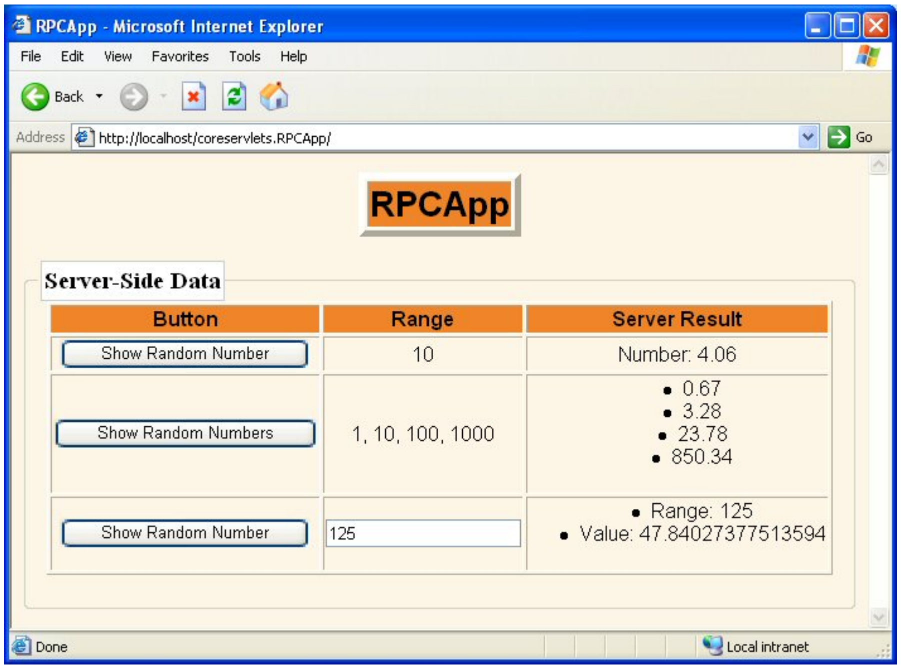
OK Cancel

Select this.

Should match the context path from the Java code.

Due to JavaScript security restrictions (same source requirement), you must deploy the client part (HTML and JavaScript) from the same host on which you deploy the server-side code.

Final Deployment
(Deploy New Project Normally)


RPCApp - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://localhost/coreservlets.RPCApp/

RPCApp

Server-Side Data

Button	Range	Server Result
Show Random Number	10	Number: 4.06
Show Random Numbers	1, 10, 100, 1000	• 0.67 • 3.28 • 23.78 • 850.34
Show Random Number	125	• Range: 125 • Value: 47.84027377513594

Done Local intranet

Context Root: Simple Approach (Shown Earlier)

- **Context root of original project is /package.AppName**
  - Done automatically by GWT
- **Java code specifies this context path**

```
endpoint.setServiceEntryPoint
       ("/package.AppName/rpc-servlet-address");
```

- **Set context root of new project to /package.AppName**
  - Done manually
- **URLs of original project and deployed project match closely**

Context Root: Alternative Approach

- **Java code chooses context path depending on whether it is running in hosted mode**

```
String prefix = GWT.getModuleBaseURL();
if (GWT.isScript()) {
  prefix = "/whatever";
}
endpoint.setServiceEntryPoint
       (prefix + "/rpc-servlet-address");
```

Returns "/package.AppName"

- **Set context root of new project to /whatever**
  - Done manually
- **URLs of original project and deployed project need not be similar**

Summary

- **Define client-side data service interfaces**
  - E.g.: DataService and DataServiceAsync
- **Make a data service servlet**
  - Extend RemoteServiceServlet, implement interface
- **Specify the data source**
  - In Java code, give full path to relative service entry point
  - In BlahApp.gwt.xml, give relative path to servlet
- **Define client-side callback classes**
  - Implement AsynCallback with onSuccess and onFailure
- **Test**
  - Hosted mode
  - Web mode
- **Deploy**
  - Copy to new Web project that uses gwt-servlet.jar
  - Set context path and url-pattern

Questions?