JavaOne Hands-On Lab

Developing Revolutionary Web Applications, Using Ajax Push or Comet

LAB-5558

LAB-5558: Developing Revolutionary Web Applications, Using Ajax Push or Comet

Expected Duration: 120 minutes

Contacts: Carol McDonald Doris Chen Justin Bolter

Last Updated: May 11th, 2009

Join the asynchronous Web revolution! Emerging Ajax techniques -- variously called Ajax Push, Comet, and HTTP streaming -- are bringing revolutionary changes to Web application interactivity, moving the Web into the Participation Age. Because Ajax-based applications are almost becoming the de facto technology for designing Web-based applications, it is more and more important that such applications react on the fly, to both client and server events. Ajax can be used to enable the browser to request information from the Web server but does not allow a server to push updates to a browser. Comet solves this problem. It is a technology that enables Web servers to asynchronously push events to Web clients, enabling event driven operations and functions, previously unheard of with traditional Web applications, to approach the capabilities of desktop applications.

In this lab, you will use NetBeans IDE to rapidly develop rich web application and then you will use Glassfish Application server to deploy the application. This lab provides an brief introduction to the asynchronous web, AJAX polling, long polling, and Streaming, explaining the Bayeux protocol, Cometd, Grizzly Comet implementation on GlassFish. Different approaches and best practices to develop comet application will also be discussed. You will learn how to develop the chat application, how to implement distance learning slideshow application, how to access Restful web services and database, how to manage a chat application from the server and how to develop a two-player distributed game application. Attendees will take away the tactics they need in order to add multiuser collaboration, notification and other Comet features to their application, whether they develop with Dojo, jQuery, jMaki, or Prototype and whether they deploy on Jetty, Tomcat, or the GlassFish Application Server.

Prerequisites

This hands-on lab assumes you have some basic knowledge of, or programming experience in, the following technologies:

- Java™ language programming
- Exposure to AJAX, Java Script programming preferred

System Requirements

- Supported OSes: Solaris™M 8/9/10 Operating System (OS), OpenSolaris™ Operating System (OS), Linux, Windows, Mac OS X 10.4+
- Memory requirement: 512MB
- Disk space requirement: 650-750 MB depends on the OS. See the requirement for more detail.

Software Needed For This Lab

Please install the following set of software. If you have any questions on installation, please feel free to send questions to the lab forum mentioned below.

- A Java SE software installation, version 1.6.0 or more recent
- NetBeans IDE 6.5.1 and above.
- GlassFish v3 version glassfish-v3-ea-b44.zip

Notations Used in This Documentation

- <lab_root> - directory into which lab zip file is unzipped

  - This document uses <lab_root> to denote the directory under which you have unzipped the lab zip file of this hands-on lab. The name of the lab zip file of this hands-on lab is **5558_cometajax.zip**.
  - Once you unzipped the lab zip file under <lab_root>, it will create a subdirectory called **cometajax**. For example, under Solaris, if you have unzipped the lab zip file in the **/home/dant** directory, it will create **/home/dant /cometajax** directory.

- This lab will use multiple Firefox profiles. Instructions to set this up are in Exercise 0.
  FOR JAVAONE 2009 MACHINE PROVIDED LABS: We have setup the profiles for you already. To launch Firefox, open a terminal window and type the command: `ff`
  Then select User 1 or User 2 depending on if this is the first or second browser you are opening.
- The lab machines are running Java Desktop System over Solaris 10
- In order to open a terminal window in Java Desktop System, right click any point in the background of the desktop, and select Open Terminal in the pop-up menu.
- The following source code editors are provided on the lab machines
  - vi (type vi in a terminal window)
  - emacs (type emacs in a terminal window)
  - jedit (type jedit in a terminal window)
  - NetBeans IDE (either click NetBeans IDE icon or type netbeans in a terminal window)

Lab Exercises

- Exercise 0: Install and configure the lab environment (10 minutes)
- Exercise 1: Build a comet chat application (20 minutes)
- Exercise 2: Build a comet slideshow application (20 minutes)
- Exercise 3: Server Interact with Chat Application (20 minutes)
- Exercise 4: Build a two way tic-tac-toe game (40 minutes)

Additional Resources

- Lab presentation
- Cometd framework and its Bayeux protocol support in Grizzly – Jean-Francois Arcand's blog
- Using the Grizzly Comet API – Sun document on developing web applications
- Dojo Toolkit – Dojo home page
- dojo.cometd – Cometd.org page at the Dojo Foundation
- Grizzly – Project Grizzly home page
- GlassFish Application Server – home page for the GlassFish Community
- The Aquarium – news from the GlassFish Community

Where To Go For Help

- For general questions and feedback on JavaOne Hands-On Labs, please use the JavaONE_HOL_Forum.
- You can send your questions about grizzly and comet to the grizzly public users alias:
  - users@grizzly.dev.java.net

Exercise 0: Install and Configure Lab Environment (10 minutes)

In addition to the contents of this lab's zip file you should have the following installed on your computer:

- A Java SE software installation, version 1.6.0 or more recent
- NetBeans IDE 6.5.1 .
- GlassFish v3 version glassfish-v3-ea-b44.zip

Glassfish configuration

FOR JAVAONE 2009 MACHINE PROVIDED LABS: Glassfish will automatically start when you login to the machine. However there are a few things you need to know.

When you first login into your machine Netbeans will start. You will be prompted by Glassfish to enter the admin username and password. They are:

- Username: **admin**
- Password: **adminadmin**

The next important thing to note is that you should not attempt to start or stop Glassfish from within Netbeans. Because of the way Glassfish is configured on these machines, you will need to open a terminal if you want to start or stop the Glassfish service. To do so issue one of these commands in a terminal window:

- Stop Glassfish server: `gf_stop`
- Start Glassfish server: `gf_start`
- Check Glassfish status: `gf_status`

Enable Comet support Option1: in domain.xml GlassFish

FOR JAVAONE 2009 MACHINE PROVIDED LABS: Comet is already enabled on the machines. Glassfish will start automatically with comet enabled. You do not need to follow the steps below to enable comet.

To use Comet with GlassFish, add the bold **red** line to the GlassFish `domain.xml` file in the `glassfish-v3_install_dir\glassfish\domains\domain1\config` directory:

Code Sample from: `domain.xml`

```
<http-listener port="8080" id="http-listener-1" address="0.0.0.0" default-virtual-server="server" server-na
    <property name="cometSupport" value="true"/>
</http-listener>
```

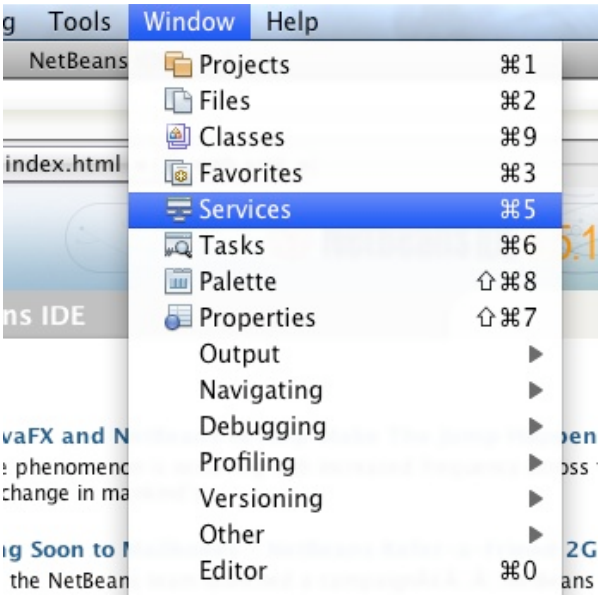
Or you can also enable comet support in NetBeans IDE.

Enable Comet support Option2: in NetBeans and GlassFish

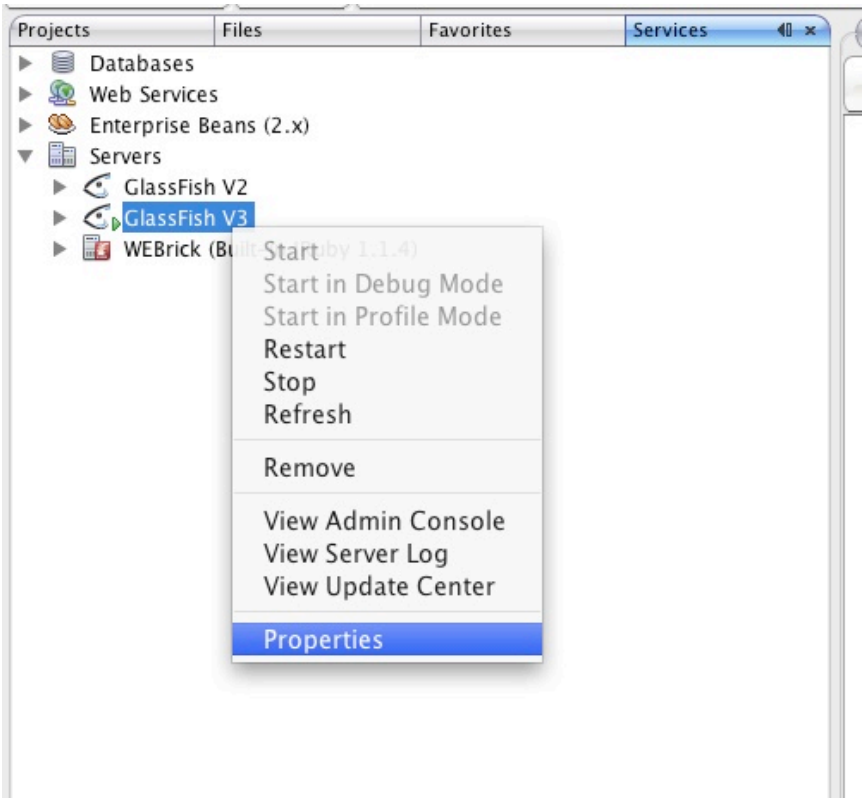
FOR JAVAONE 2009 MACHINE PROVIDED LABS: Comet is already enabled on the machines. Glassfish will start automatically with comet enabled. You do not need to follow the steps below to enable comet.

In addition to enable the comet support in domain.xml as above, this can also be accomplished easily with Netbeans and Glassfish.

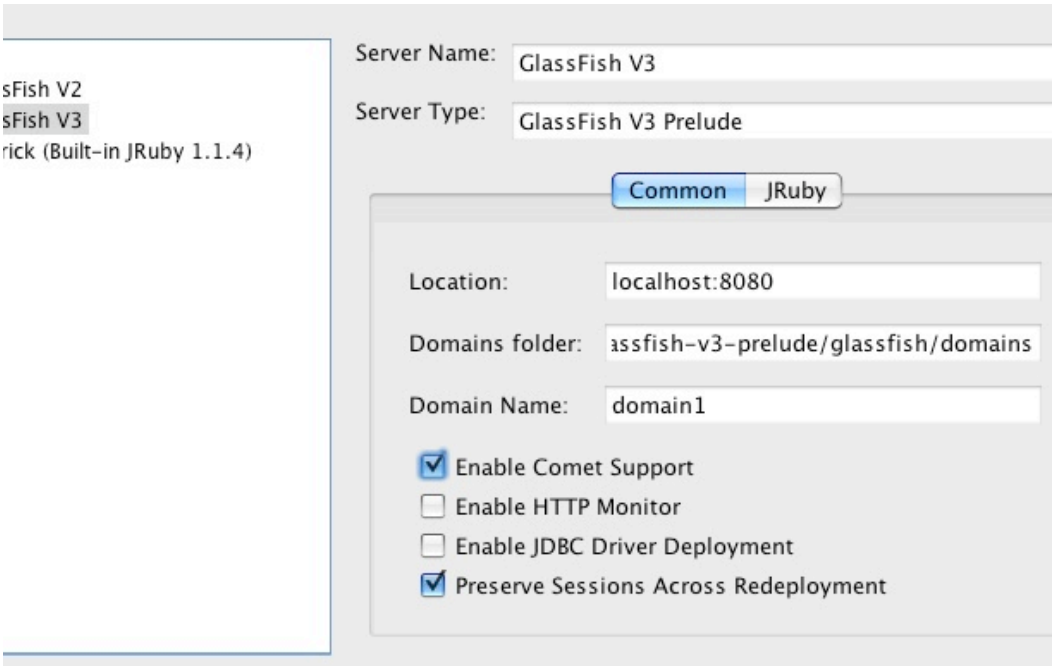
1. Switch to the Services window in Netbeans. If you can't find it, go to the Windows menu and select Services.



2. Right click your Glassfish server and select Properties.



3. In the Server's Common properties window you should see a box labeled **Enable Comet Support**. Check this box and select OK.



Setup Two Firefox Profiles

For this lab we will simulate multiple users in the exercises. To do this we'll need to setup a second Firefox profile so that we can simulate two independent browser sessions. This can be accomplished quite easily using the Mozilla Profile Manager. Here are instructions for the various operating systems, if you need more information or you have trouble you can check the Mozilla KB.

FOR JAVAONE 2009 MACHINE PROVIDED LABS: Firefox is already setup with multiple profiles. When you first login to the machine you will see a prompt to select a Firefox profile. You should select *User 1* the first time. The next time you want to launch a new browser, open a terminal window and type the command `ff`. This will launch Firefox and allow you to select *User 2*. You do not need to follow the instructions below to setup multiple profiles.

Steps To Follow

1. Close the application completely and make sure that it is not running in the background.

   OpenSolaris/Linux

   Open a terminal and execute:

   ```
   firefox -profilemanager
   ```

   If firefox is not in your path, you need to `cd (firefox program directory)` and then execute:

   ```
   ./firefox -profilemanager
   ```

   MacOS

   Assuming the program is installed in the "Applications" folder, launch Terminal ("Applications -> Utilities -> Terminal") and enter the command starting with / after the prompt in Terminal:

   ```
   /Applications/Firefox.app/Contents/MacOS/firefox -profilemanager
   ```

   Windows

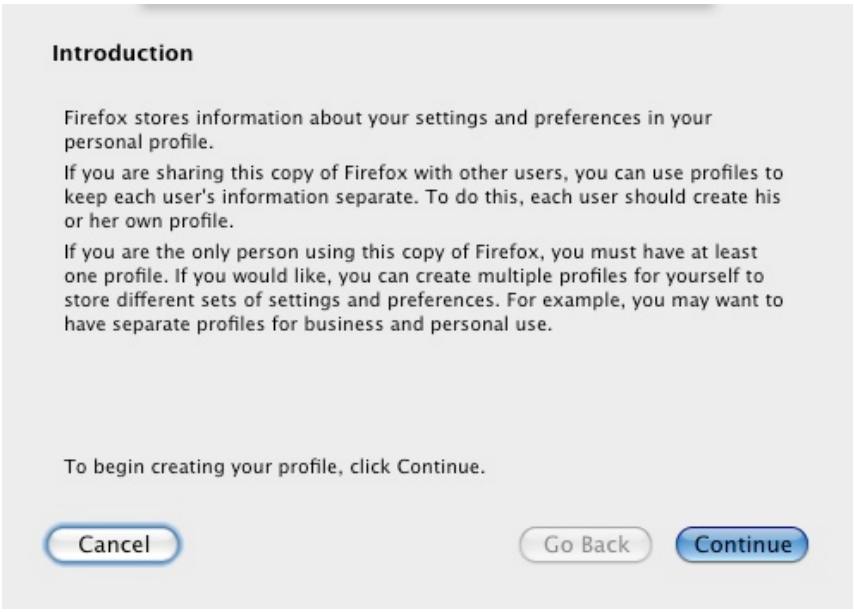
   Open the Windows "Start" menu, select "Run" (on Windows Vista, use "Start Search") then type and enter one of the following:

   ```
   firefox.exe -profilemanager
   ```

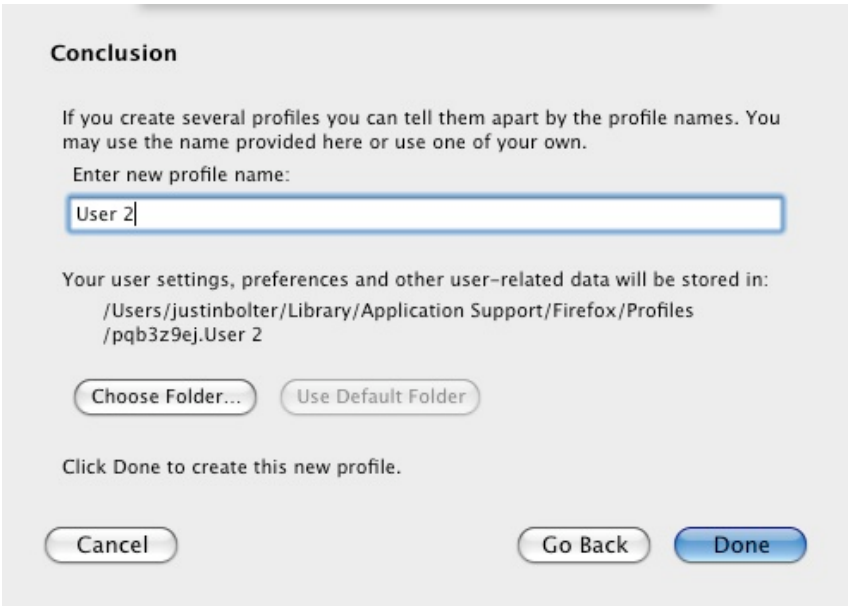
   if the above instructions do not work, include the full path to the executable surrounded by quotation marks in the "Run" (or Vista "Start Search") box, as in this Firefox example:

```
"C:\Program Files\Mozilla Firefox\firefox.exe" -profilemanager
```

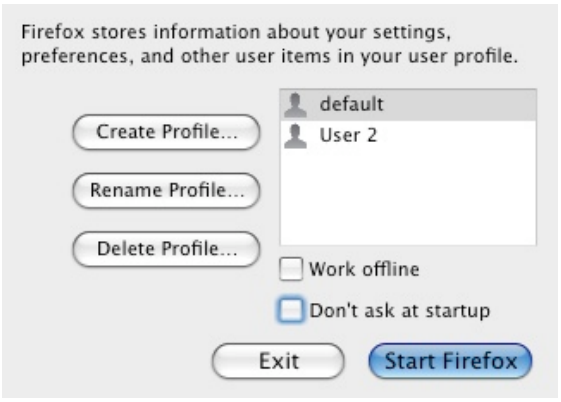
2. Select Create Profile
3. Click Continue



4. Enter a new profile name and click Done.



5. Finally, uncheck the box that says **Don't ask at startup**



Now when you start Firefox you will be able to choose which profile you want to use. This will allow you to use Firefox to simulate two browser sessions by choosing different profiles for each user. The first time you should select your default profile. Now launch the profile manager again from your terminal window and select your second user and choose Start Firefox. If the profile manager does not launch try running this command and then you should be able to select your second profile (details):

```
firefox -P -no-remote
```

Exercise 1: Build a Comet Chat Application (20 minutes)

In this lesson we are going to build a chat applicaton using comet.

Background Information

The term Ajax was invented to describe the technologies that provide background request-response between a web server and client browser. With Ajax, web applications can retrieve data from the server in the background, essentially preloading information that is revealed in response to a user gesture and giving the impression of a more responsive web page. Although the data appears to be downloaded asynchronously, the server responds to client requests — typically through the use of the `XMLHttpRequest` object.

Web protocols require that servers respond to browser requests. Ajax falls short for multi-user, interactive pages because one user won't see changes other users make until the browser sends a request. In order for a server to push data to a browser asynchronously, some design workarounds are required. These design strategies are broadly described by the term **Reverse Ajax**.

One strategy is to use browser polling, in which the browser makes a request of the server every few seconds. Another strategy, known as piggybacking, delays a server's page update until the next request is received from the browser, at which time the server sends the pending update along with the response.

A third strategy, and the one used in the example application presented in this article, is to design with long-lived HTTP connections between the browser and server. With this strategy, a browser does not poll the server for updates; instead, the server has an open line of communication it can use to send data to the browser asynchronously. Such connections reduce the latency with which messages are passed to the server. This technique is known as Comet — a play on words that can be appreciated by users of household cleansers.

The example application in this exercise provides a chat feature that enables users to publishmessages and let subscribed users see the messages updated on their browser.

The example application incorporates the following technologies. All are available for download.

- Ajax – Techniques that provide background updating of web pages using a request-response model.
- Comet – Techniques that enable a server to push data to client browsers through an HTTP open line of communication.
- Dojo – An open-source DHTML toolkit written in JavaScript. The Dojo Toolkit includes many utilities that go beyond Ajax. For example, the `dojox.comet` module simplifies programming Comet applications.
- Bayeux – A protocol for routing JSON-encoded events between clients and servers in a publish-subscribe model.
- GlassFish – Open-source Java EE-compliant application server.
- Grizzly – Open-source framework designed to help developers take advantage of the Java New I/O API (Java NIO API). Grizzly comes bundled with GlassFish application server, but it can be used separately.

More About the Technologies

Comet is a term coined by Alex Russell to describe applications where the server pushes data to the client over a long-lived HTTP connection. Comet uses the persistent connection feature in HTTP/1.1. This feature keeps alive the TCP connection between the server and the browser until an explicit 'close connection' message is sent by one or the other, or until a timeout or network error occurs. The technique enables the server to send a message to the client when an event occurs without waiting for the client to send a request. Cometd is a scalable HTTP-based event routing bus that uses a push technology pattern known as Comet.

Grizzly is an HTTP framework that uses the Java NIO API to provide fast HTTP processing. Grizzly provides long-lived streaming HTTP connections that can be used by Comet, with support built on top of Grizzly's Asynchronous Request Processing (ARP). With Grizzly ARP, each Comet request doesn't monopolize a thread, and the availability of threads permits scalability.

Bayeux is a protocol for routing JSON-encoded events between clients and servers in a publish-subscribe model. Grizzly provides an implementation of Bayeux, which makes it easy to build Comet applications with `dojox.comet`. With the support for Cometd and Bayeux, an application can be developed using only Dojo (or Ajax component), and the application can be deployed in any server that supports the Bayeux protocol. To build your application, configure GlassFish for Comet and configure your web application's `web.xml` file for the Grizzly Bayeux servlet. You can then use the `dojox cometd` publish and subscribe methods to send and receive Comet events. These techniques are described in more detail in the description of the application.

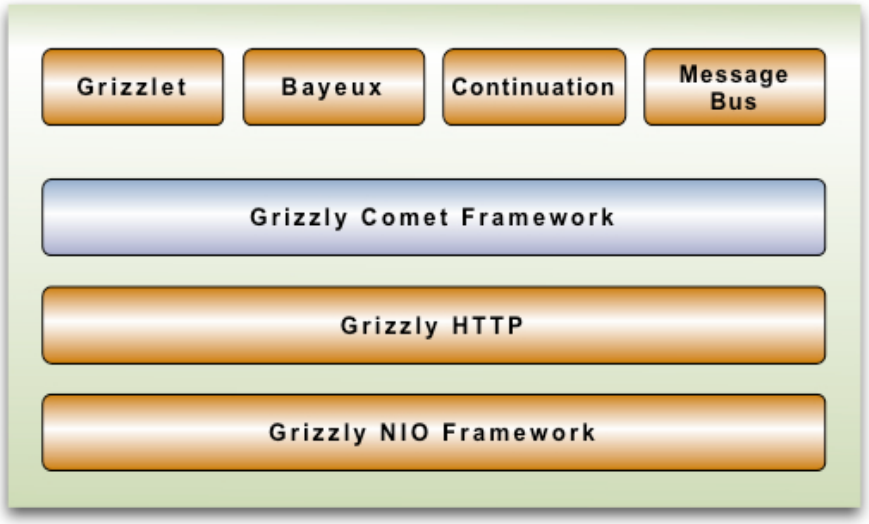


Figure 1: *Grizzly Architecture*

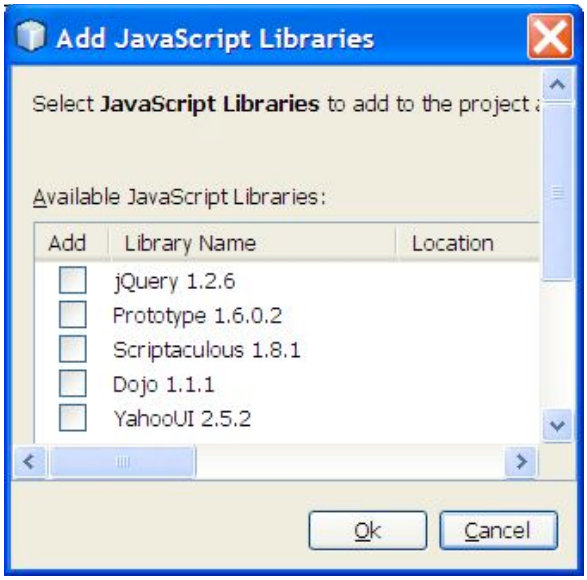
Steps to Follow

Step 1: create a new Web Application project

1. If NetBeans is not already running, start it.
2. Choose File > New Project (Ctrl-Shift-N) from the main menu. Under Categories, select Java Web. Under Projects, select Web Application and click Next.
3. Type **chat** in the Project Name field. Note that the Context Path becomes /chat.
4. Specify the Project Location to the exercise1 sub directory of this lab on your computer.
5. Under Server, select GlassFish v3. GlassFish is a Java EE5-certified application server and is bundled with the Web and Java EE insallation of NetBeans IDE.
6. Leave the Set as Main Project option selected and click Finish. The IDE creates the chat project folder. The project folder contains all of your sources and project metadata, such as the project's Ant build script. The chat project opens in the IDE. The welcome page, index.jsp, opens in the Source Editor in the main window. You can view the project's file structure in the Files window (Ctrl-2), and its logical structure in the Projects window (Ctrl-1).)

Step 2: Installing and Using Dojo With NetBeans IDE

There are three main ways to install Dojo, which you can read about in the book of Dojo. Netbeans 6.5 has JavaScript libraries bundled with it, which makes it easy to add the dojo libraries to your project.



In this lab we are not using the current bundled version because it does not contain dojox, which we need for the dojox.grid widget and dojox.cometd. If you want to use a different version of dojo than the bundled one (like in this HOL), another easy way to use Dojo with the NetBeans IDE is to download and extract the dojo JavaScript library, and then copy it into your Netbeans project web directory: `.../web`. The dojo JavaScript library has already been downloaded and extracted for this HOL and is in the **js** folder in the hol\resources\dojo directory.

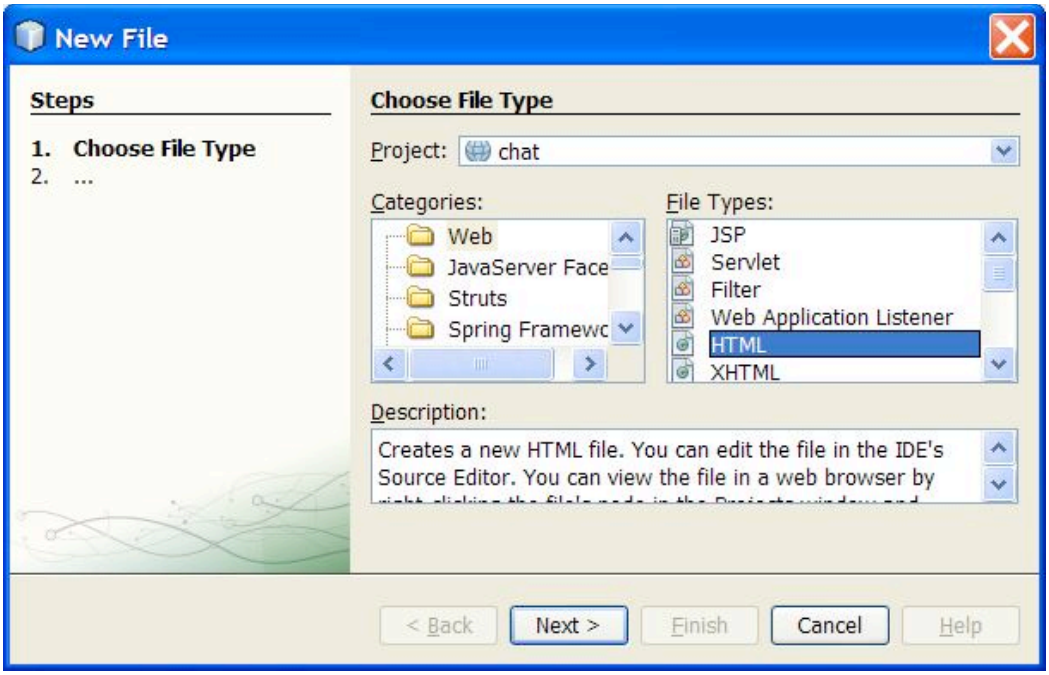
1. In your file system copy the **js** folder from the <lab_root>\exercises\exercise1\dojo directory to your chat project's web directory \exercises \exercise1\chat\web. The **js** folder contains the dojo javascript libraries.

2. You should now see the js folder in your chat project as below:

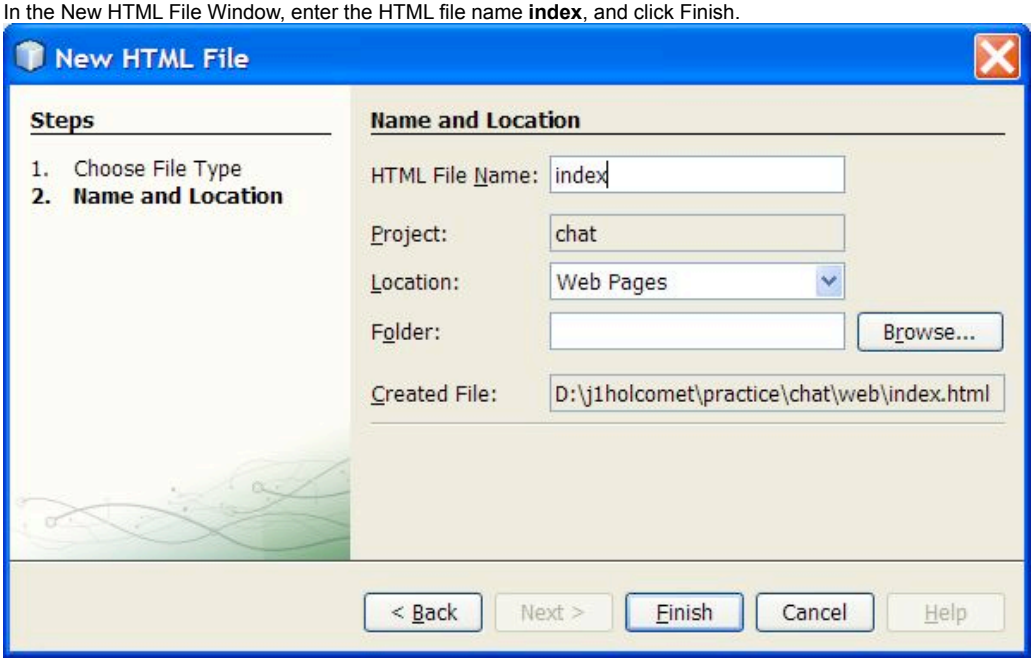
Step 3:

1. In the Projects window right click the `chat` node (from exercise 1), and select New > Other....

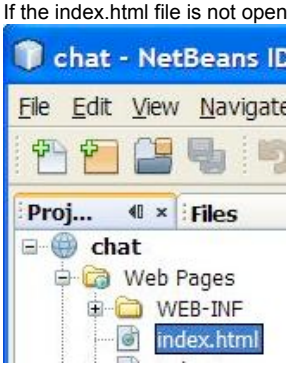
2. In the New File window select Web HTML and click Next



3. In the New HTML File Window, enter the HTML file name **index**, and click Finish.



4. If the index.html file is not open , then Double click on the **index.html** file to open it in the editor.

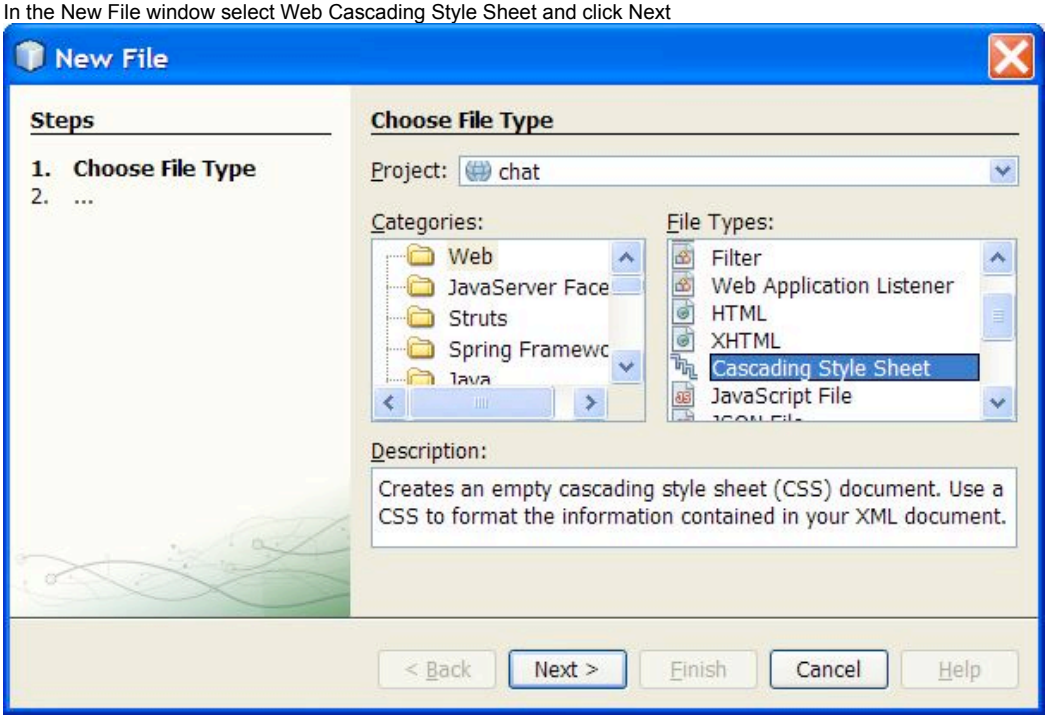


5. Copy all of the html contents below and paste them to replace the contents of the index.html file in the editor.

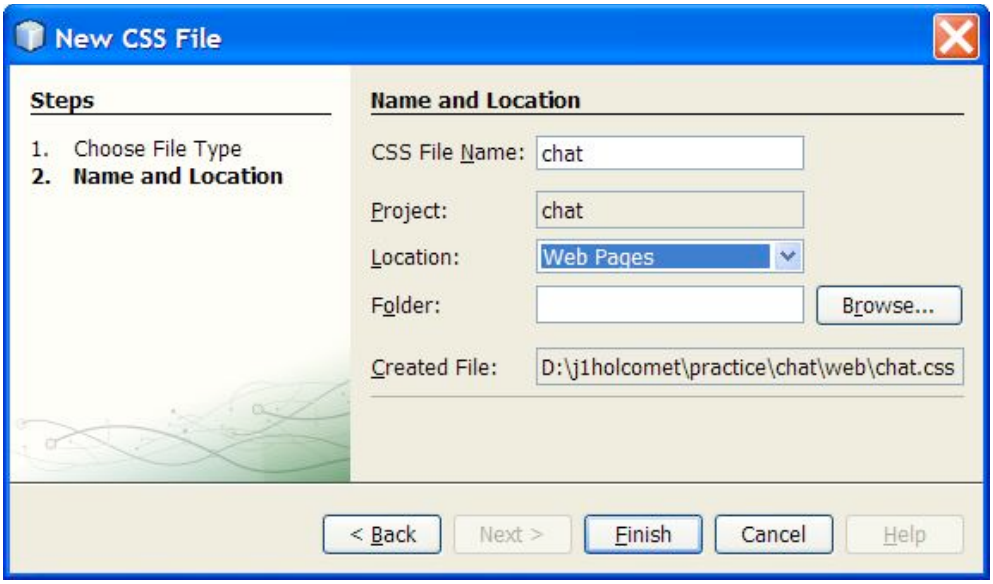
```
<html>
<head>
   <title>chat</title>
   <style type="text/css">
      @import "js/dijit/themes/tundra/tundra.css";
      @import "js/dojo/resources/dojo.css";

   </style>

   <script type="text/javascript" src="js/dojo/dojo.js"
         djConfig="parseOnLoad: true"></script>
   <script type="text/javascript" src="chat.js"></script>
   <link rel="stylesheet" type="text/css" href="chat.css">
</head>
<body class="tundra">

   <h1>Chat</h1>
   <br>
   <div id="chatroom">
      <div id="join" >
         <label for="sendName" style="float: left; width: 100px; padding: 3px;">Name:</label>
         <input id="sendName" type="text" dojoType="dijit.form.TextBox">
         <button id="joinB" dojoType="dijit.form.Button">Join</button>
      </div>
      <div id="joined" class="hidden">
         <label for="sendText" style="float: left; width: 100px; padding: 3px;">Message:</label>
         <input id="sendText" type="text" dojoType="dijit.form.TextBox">
         <button id="sendB" dojoType="dijit.form.Button">Send</button>
         <button id="leaveB" dojoType="dijit.form.Button">Leave</button>
      </div> <br>
      <div id="chat">
         <div id="messageLog">Messages:</div>
      </div>
   </div>
</body>
</html>
```

6. In the Projects window right click the `chat` node (from exercise 1), and select New > Other....

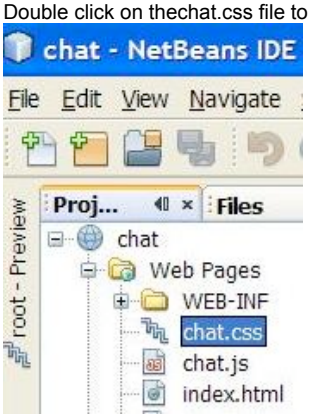
7. In the New File window select Web Cascading Style Sheet and click Next



8. In the New CSS File Window, enter the CSS file name **chat**, and click Finish.



9. Double click on thechat.css file to open it in the editor.



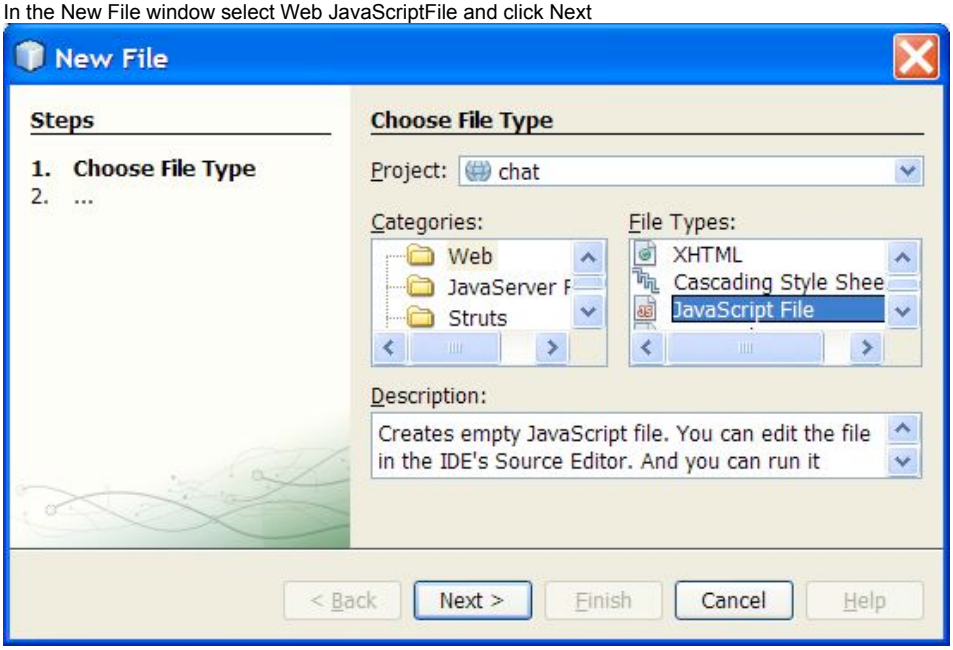
10. Copy all of the contents below and paste them to replace the contents of the chat.css file in the editor.

```
div
{
    border: 0px solid black;
}
div#chatroom
{
    background-color: #e0e0e0;
    border: 1px solid black;
    width: 45em;
}
div#chat
{
    background-color: #f0f0f0;
    padding: 4px;
    border: 0px solid black;
}
div#input
{
    clear: both;
    padding: 4px;
    border: 0px solid black;
    border-top: 1px solid black;
}
input#sendText
{
    width:28em;
    background-color: #e0f0f0;
```

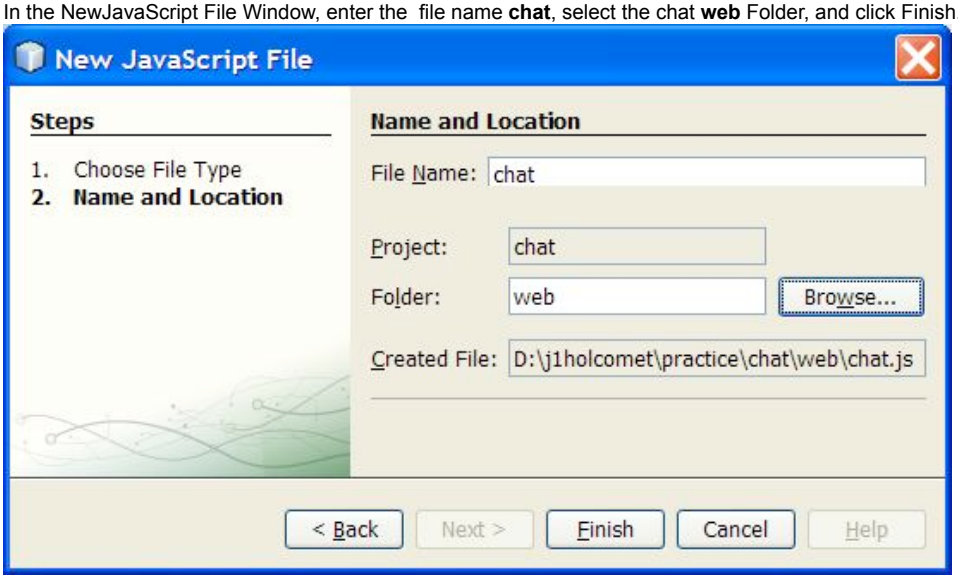
```
}
input#sendName
{
    width:14em;
    background-color: #e0f0f0;
}
div.hidden
{
    display: none;
}
span.alert
{
    font-style: italic;
}
```

11. In the Projects window right click the `chat` node (from exercise 1), and select New > Other....

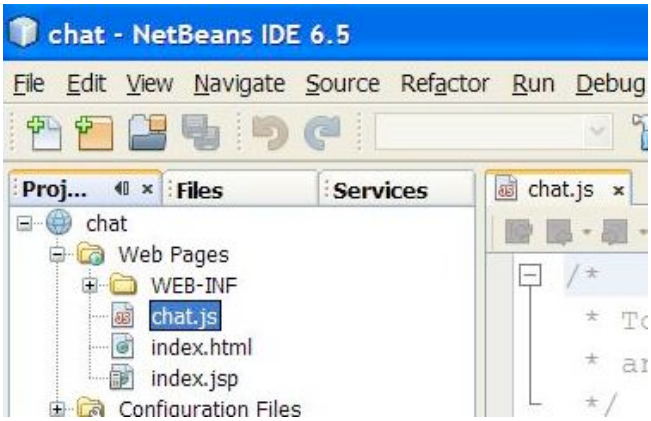
12. In the New File window select Web JavaScriptFile and click Next



13. In the NewJavaScript File Window, enter the file name **chat**, select the chat **web** Folder, and click Finish.



14. Double click on the chat.js file to open it in the editor.



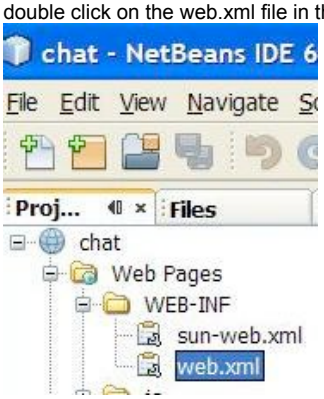
15. Copy the JavaScript contents below and paste them into the chat.js file in the editor.

```
dojo.require("dojox.cometd");

var room = {
    username: null,
    channel: "/chat/demo",

    join: function(name){
        dojox.cometd.init("cometd");
        room.username=name;
        dojo.byId('join').className='hidden';
        dojo.byId('joined').className='';
        dojox.cometd.subscribe(room.channel, room, "chatCallback");
        dojox.cometd.publish(room.channel, {
            user: room.username,
            text: " has joined"
        });
    },
    leave: function(){
        if (room.username==null)
            return;
        dojox.cometd.unsubscribe(room.channel, room, "chatCallback");
        dojox.cometd.publish(room.channel, {
            user: room.username,
            text: " has left"
        });
        dojox.cometd.disconnect();
        // switch the input form
        dojo.byId('join').className='';
        dojo.byId('joined').className='hidden';
        room.username=null;
    },
    chat: function(text){
        dojox.cometd.publish(room.channel, {
            user: room.username,
            text: text
        });
    },
    chatCallback: function(message){
        if(!message.data){
            alert("bad message format "+message);
            return;
        }
        var messageLog=dojo.byId('messageLog');
        var from=message.data.user;
        var text=message.data.text;
        from+=":";
        // display the  chat text
        messageLog.innerHTML += "<br/><span class=\"from\">"+from+"  </span><span class=\"text\">"+text+"</span>";
    },
```

```
    init: function(){
        dojo.connect(dojo.byId("joinB"),"onclick",function(){
            if (!dojo.byId("sendName").value.length) {
                alert("Please enter Name");
                return;
            }
            room.join(dojo.byId('sendName').value);
        });
        dojo.connect(dojo.byId("sendB"),"onclick",function(){
            if (!dojo.byId("sendText").value.length) {
                alert("Please enter Text");
                return;
            }
            room.chat(dojo.byId("sendText").value);
        });
        dojo.connect(dojo.byId("leaveB"),"onclick",function(){
            room.leave();
        });
    }
};

dojo.addOnLoad(room, "init");
dojo.addOnUnload(room,"leave");
```

Edit the web.xml file to change the welcome page:

1. double click on the web.xml file in the chat WEB-INF directory



2. Click on the Pages tab. Change the Welcome File from index.jsp to **index.html .**

Step 4: Configuring for Comet

The goal of this part of exercise 1 is to configure Glassfish for comet and your web project for the Grizzly Cometd servlet.

1. **Enabling Bayeux in GlassFish**
   To enable Bayeux on GlassFish, add the lines shown in red below to your Web application's `web.xml` file:

   Code Sample from: `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" ...>

  <servlet>
    <servlet-name>Grizzly Cometd Servlet</servlet-name>
    <servlet-class>
      com.sun.grizzly.cometd.servlet.CometdServlet
    </servlet-class>
    <init-param>
     <param-name>expirationDelay</param-name>
     <param-value>-1</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
   <servlet-name>Grizzly Cometd Servlet</servlet-name>
   <url-pattern>/cometd/*</url-pattern>
  </servlet-mapping>

  <session-config>
      <session-timeout>
            30
      </session-timeout>
  </session-config>
  <welcome-file-list>
        <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

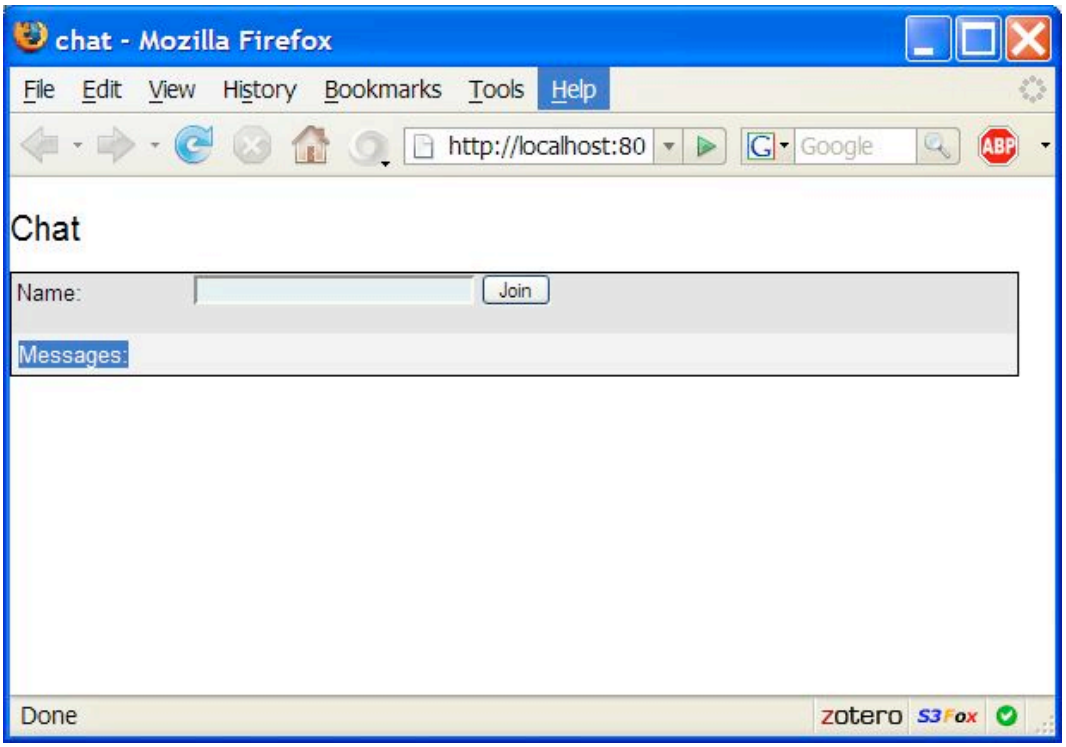
Every request sent to your `war` file's *`context-path`*`/cometd/` will be serviced by the Grizzly Bayeux runtime.

2. **Enable Comet support in Glassfish** if you haven't done so in exercise 0.
   See exercise 0 for detail on how to enable comet support in Glassfish if you haven't done so.

Step 5: Run the Project

1. Right click the chat node in the Projects window, and select Run.
2. When you run the project, your browser should display the opening page of the dojo comet Sample Application (at http://localhost:8080/chat/).
3. Open another browser, preferably with different profile or different brand of browser, and type http://localhost:8080/chat/ to access the application. Enter name to join the chat and then enter some message.

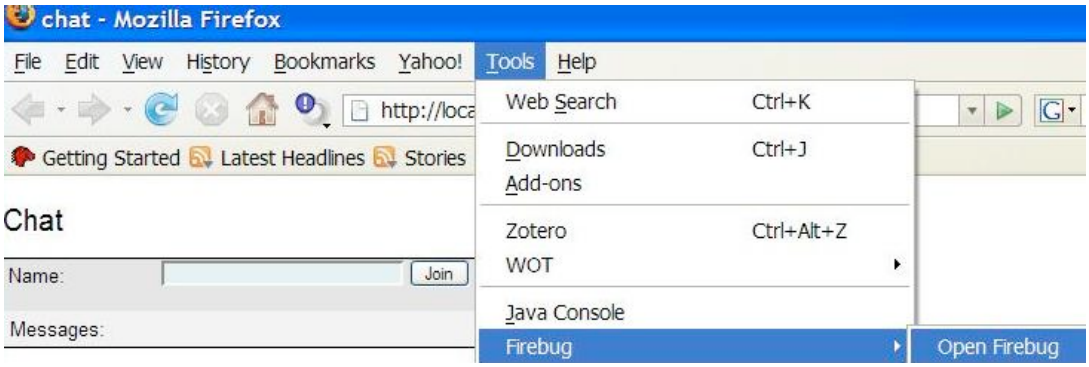
   Comet chat page, which allows the users to chat at the same time.



Different Browsers Sharing Photos in Comet chat

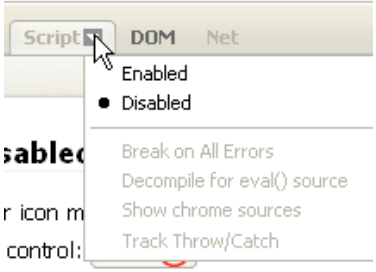
Step 6: Use Firebug to see the JSON Messages

Firebug is Firefox plugin which allows you to debug, and monitor CSS, HTML, and JavaScript live in any web page. In this lab you can use it to see the HTTP Request and Response content in order to see the JSON messages exchanged during the chat session. In Firefox , Select Tools > Firebug > Open Firebug.



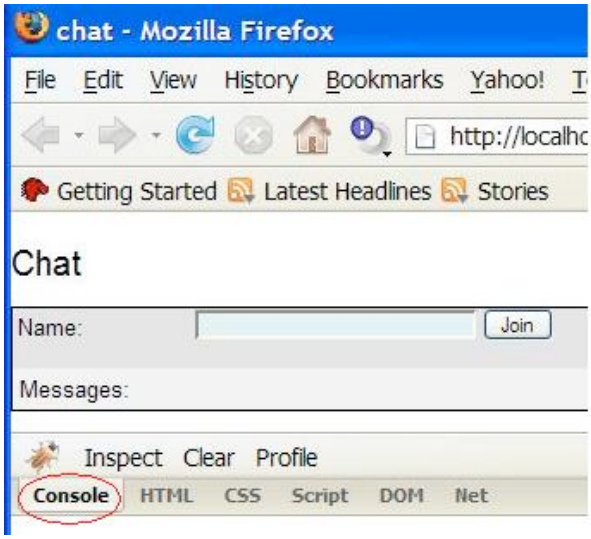
Enabling Firbug Panels

To work on Javascript code or to study the network action on a site, you need to enable one or more Firebug panels. If you select the Console, Script, or Net panels, you will see the tab is grey and the panel says "disabled". Each panel tab has a small menu control for enabling the panel. All the panels can be enabled or disabled using the context menu (right click) on the Firebug status bar icon.

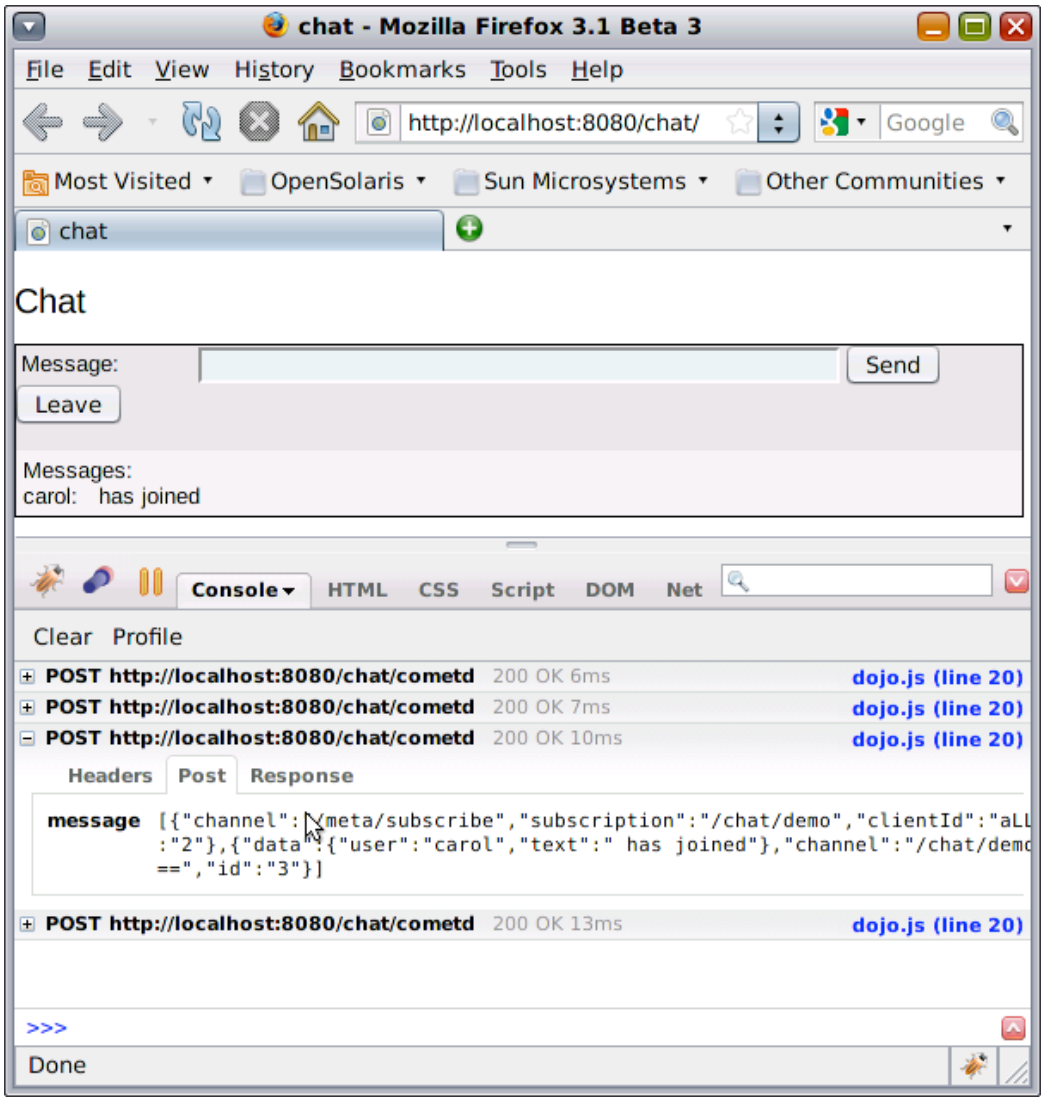


See Enabling Firebug for more information.

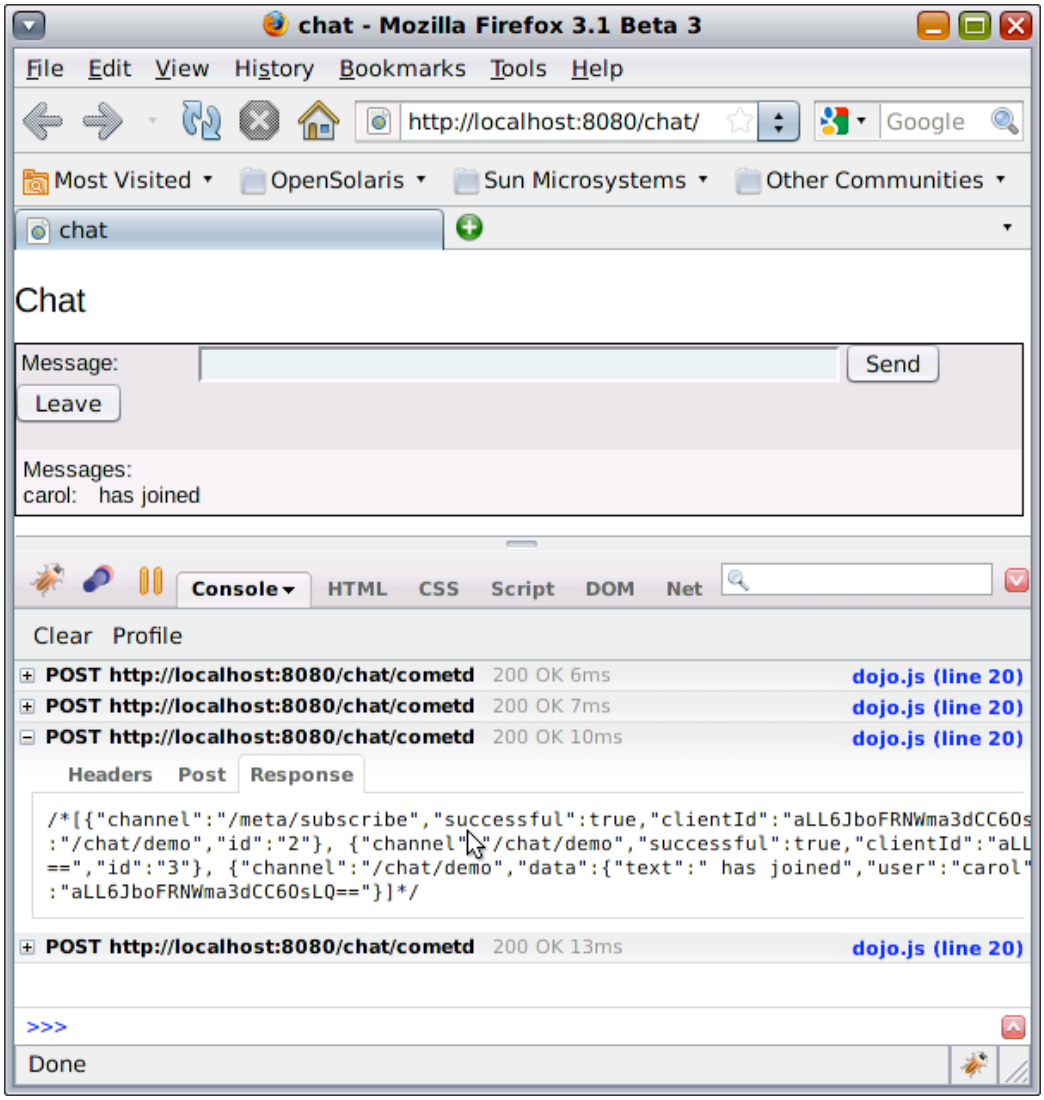
Click on the **Console panel and select Enabled.**



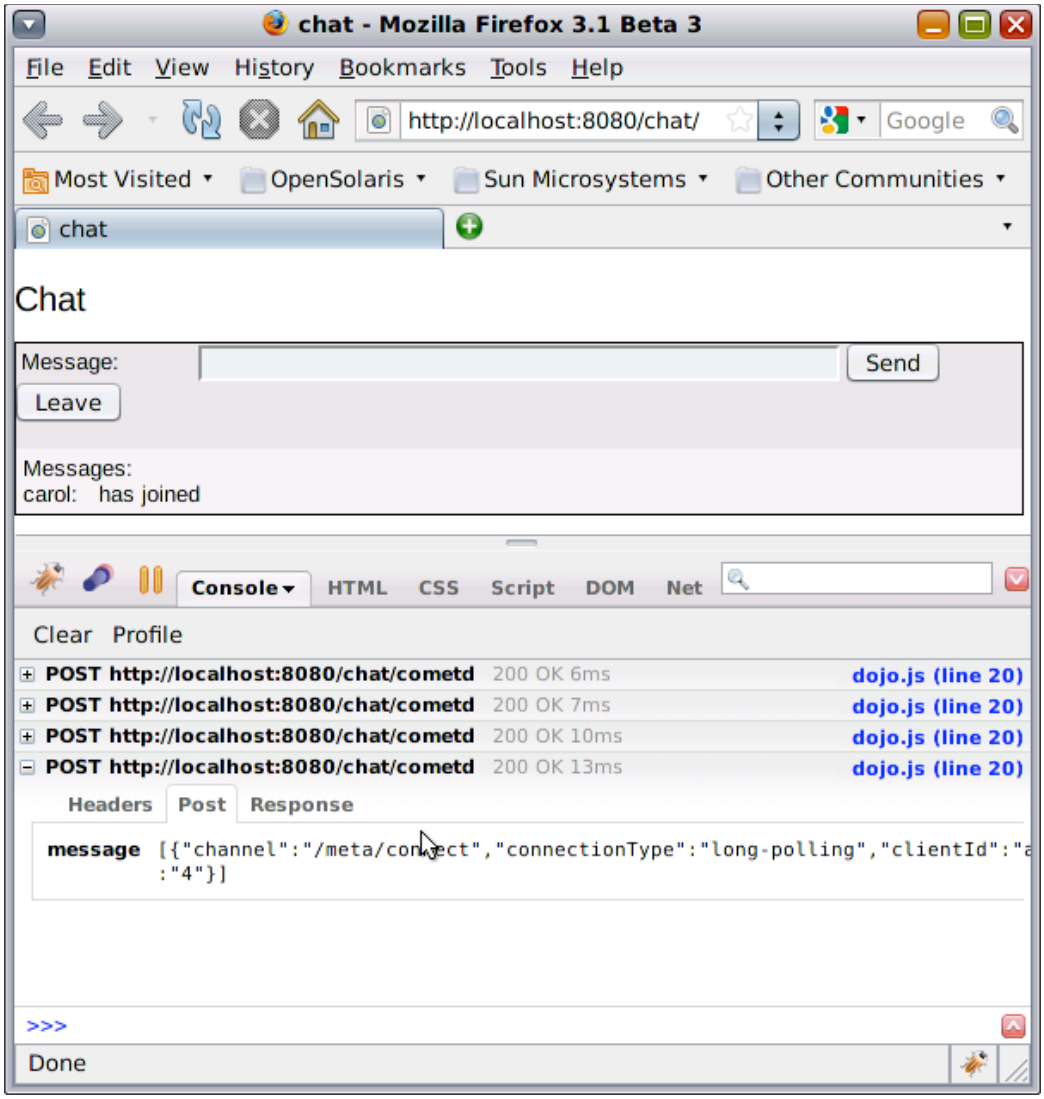
In the Chat Application type in a name and click Join . You should see the HTTP Posts as shown below. Click to expand a **Post Request** , then select the **Post tab ,** you should see the Bayeux JSON message sent as shown below. When you join a the chat session multiple json messages are sent between the browser client and the glassfish servlet to perform a handshake. This is explained more in understanding the code below. The image below shows the JSON message that user carol has joined.



Click to expand a **Post Request** , then select the **Response tab ,** you should see the Bayeux JSON message received as shown below.



Click to expand the last **Post Request** , then select the **Post tab ,** you should see the Bayeux JSON message as shown below. This is the long-polling HTTP "keep alive" Request which remains blocked until a message in the chat session is sent (or until it times out).



Understanding the code

Using dojox.cometd

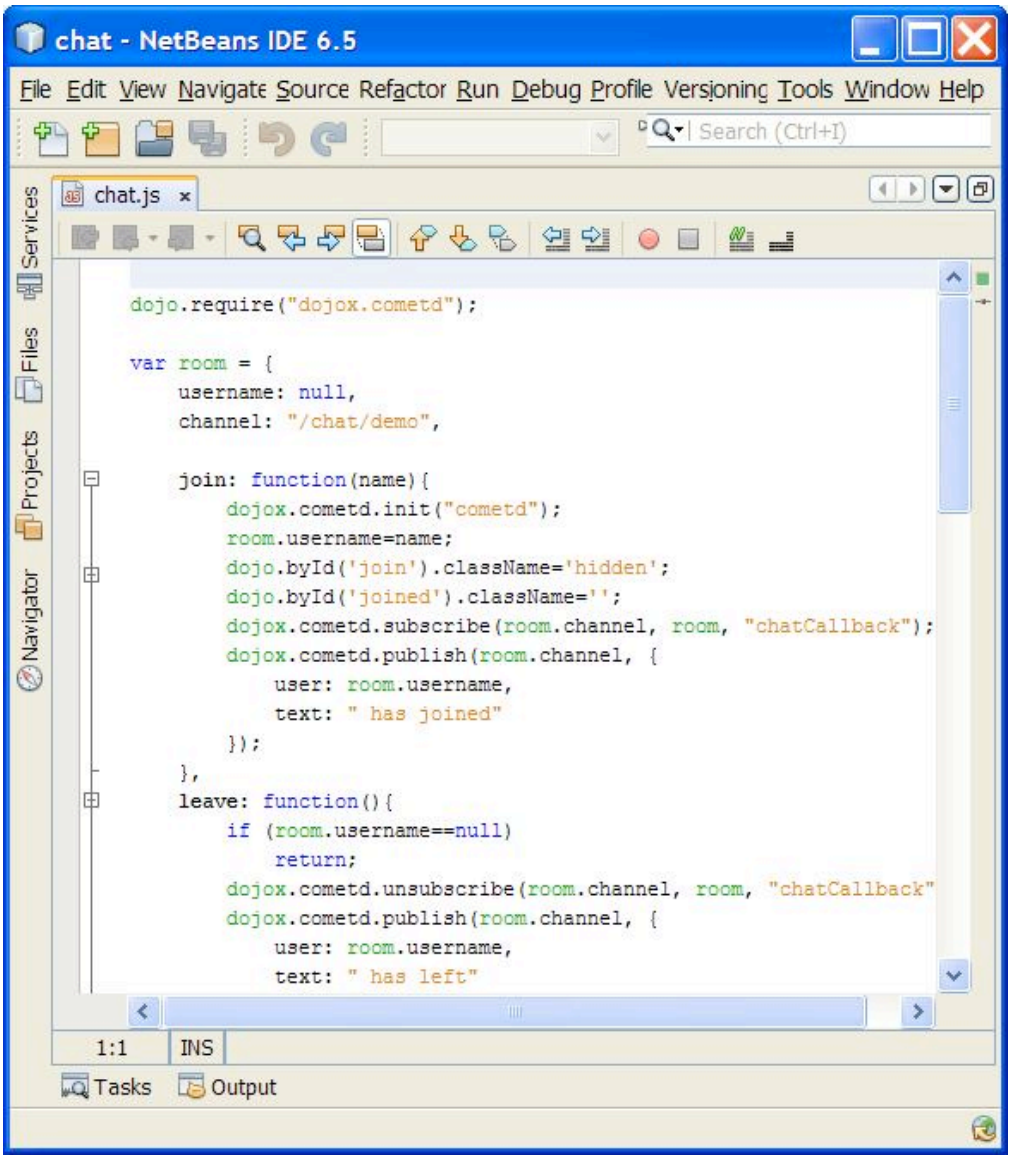
Loading Base Dojo and Required Modules Into the Application

This script element: src="src/dojo/dojo.js" loads the base dojo script that gives you access to the Dojo functionality, other dojo modules are loaded as required with the the `dojo.require` function explained later.

Code Sample from: `index.html`

```
<script type="text/javascript" src="js/dojo/dojo.js"></script>
<script type="text/javascript" src="chat.js"></script>
```

The rest of the JavaScript code for this application is in the file `chat.js`. Note that NetBeans IDE 6.5 has a very capable JavaScript editor, shown below editing the `chat.js` file:



In `chat.js`, the application uses the `dojo.require` function (similar to the `import` directive in Java) to specify which Dojo modules to load.

Code Sample from: `chat.js`

```
dojo.require("dojox.cometd");
```

Dojo is organized into three major layers: Dojo Core, Dijit, and DojoX. DojoX builds on Dojo Core and provides newer extensions to the Dojo Toolkit. DojoX cometd implements a Bayeux protocol client for use with a Bayeux server.

Initializing a Connection Between the Dojo Client and the Grizzly Bayeux Servlet

Upon loading the chat application, a user can enter a username and join a chat session, as shown in Figure 3.

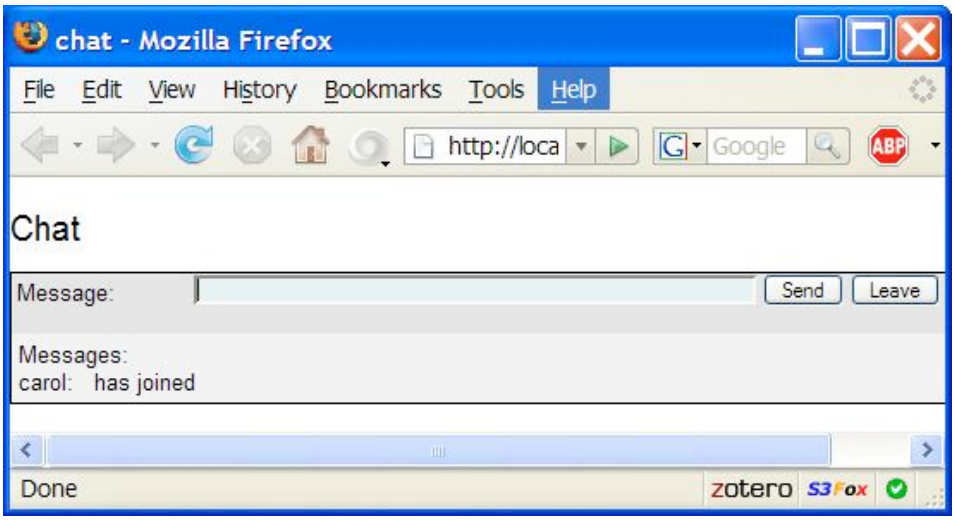


Figure 3: *User Login Page*

When a user clicks the Join button, the `join` JavaScript function is called. In the `join` function, the call to `dojox.cometd.init` initializes a connection to the given Comet server — in this case with the GlassFish Grizzly Bayeux servlet. Note that **"cometd"** is the URL-pattern for the Grizzly Cometd servlet configured in the `web.xml` file for the application.

Code Sample from: `chat.js`

```
var room = {
...
   join: function(name){

     dojox.cometd.init("cometd");
     dojox.cometd.subscribe("/chat/demo", room, "chatCallback");
     dojox.cometd.publish("/chat/demo",
       {  user: room.username,
          text: " has joined"});
   }
```

The `dojox.cometd.subscribe` line subscribes the `chatCallback` callback function to the `/chat/demo` channel. Whenever a message is sent to the `/chat/demo` channel, the `chatCallback` function is called. The `dojox.cometd.publish` line publishes a message that the user whose name was entered with the Join button has joined the `/chat/demo` channel. All subscribers to the `/chat/demo` channel receive the message.

Using Firebug you can see that **dojox.cometd.init("cometd")** sends the following with a POST to the server:

```
message=
[ {
    "version":"1.0","minimumVersion":"0.9",
   "channel":"/meta /handshake",
    "id":"0",
    "supportedConnectionTypes":["long-polling", "callback-polling"]
 } ]
```

A Bayeux client initiates a connection negotiation by sending a message to the "/meta/handshake" channel. A Bayeux server must respond to a handshake request with a handshake response message in the body content of the response. The response looks like this:

```
[ {
  "channel":"/meta/handshake",
    "version":"1.0",
   "supportedConnectionTypes":["long-polling","callback-polling"],
   "minimumVersion":"0.9","id":"0",
```

```
   "clientId":"xtY0HrhJ/YkXetV1CWaRSw==",
   "successful":true,
   "advice":{"reconnect" :"retry", "interval":0,"multiple-clients":false},
   "authSuccessful":true
}]
```

When you join a the chat session multiple json messages are sent between the browser client and the glassfish servlet to perform a handshake, which is specified in the Bayeux protocol

Using Firebug you can see that **dojox.cometd.publish("/chat/demo",{ user: room.username,text: " has joined"})** sends the following with a POST to the server:

```
message=[
 {  "channel":"/meta/subscribe",
    "subscription":"/chat/demo",
    "clientId":"xtY0HrhJ/YkXetV1CWaRSw==",
     "id":"2" },
 {  "data":{"user":"carol","text":" has joined"},
     "channel":"/chat/demo",
     "clientId":"xtY0HrhJ/YkXetV1CWaRSw==",
     "id":"3"}
]
```

The response looks like this:

```
[
  {"channel":"/meta/subscribe",
  "successful":true,
  "clientId":"xtY0HrhJ/YkXetV1CWaRSw==",
  "subscription":"/chat/demo",
   "id":"2"},
   {"channel":"/chat/demo",
   "successful":true,
   "clientId":"xtY0HrhJ/YkXetV1CWaRSw==",
    "id":"3"},
 { "channel":"/chat/demo",
    "data":{"text":" has joined","user":"carol"},
    "id":"3",
    "clientId":"xtY0HrhJ/YkXetV1CWaRSw=="}
]
```

Publishing the Next Slide for the Comet chat

When the user clicks the Send button, a JavaScript function is called that publishes the message, as shown in Figure 4.

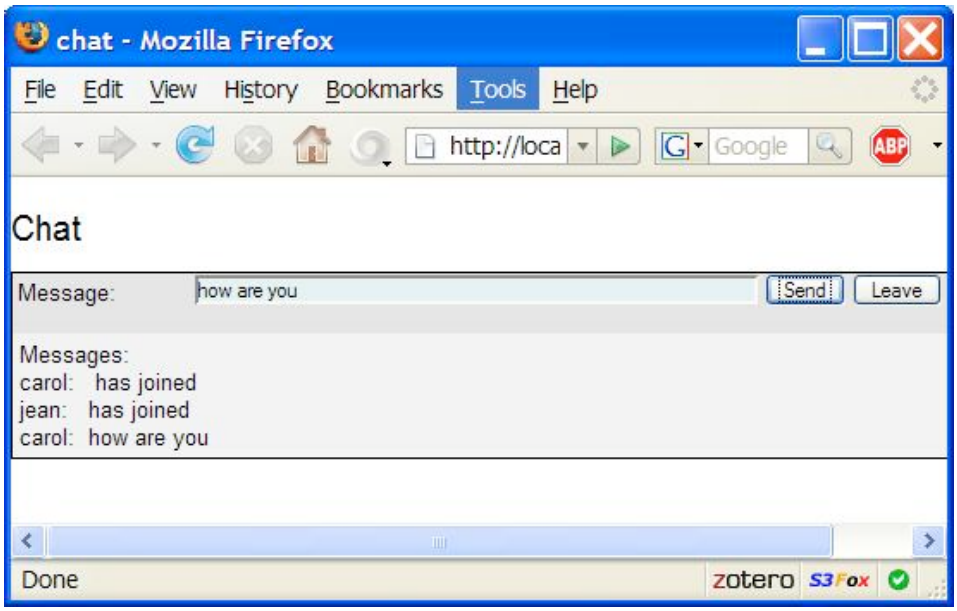


Figure 4: *Viewing the Next Slide*

Code Sample from: `index.html`

```
<button id="sendB" dojoType="dijit.form.Button">Send</button>
```

The JavaScript function that is called when the user clicks the Send button is shown in the following code sample. This function calls `room.chat`, which publishes the message.

Code Sample from: `chat.js`

```
var room = {

 ..
 dojo.connect(dojo.byId("sendB"),"onclick",function(){
    if (!dojo.byId("sendText").value.length) {
        alert("Please enter Text");
        return;
    }
    room.chat(dojo.byId("sendText").value);
 });
```

The function `room.chat`, shown in the following code sample, calls `dojox.cometd.publish` to publish the message text(which it receives as an input argument) to the `/chat/demo` channel. All subscribers to the `/chat/demo` channel receive this message.

Code Sample from: `chat.js`

```
var room = {
 channel: "/chat/demo",
 ...

 chat: function(text){
   dojox.cometd.publish(room.channel, {
       user: room.username,
       text: text
   });
 },
 ...
```

```
}
```

When a message is published to a Bayeux channel on the server, it is delivered to all clients subscribed to that channel — in this case, to the `/chat/demo` channel.

In the `room.join` function described earlier, `dojox.cometd.subscribe("/chat/demo", room, "chatCallback")` was called to subscribe the `chatCallback` callback function to the `/chat/demo` channel. The `chatCallback` function, shown in the following code sample, is called with the published message as an input argument. The `chatCallback` function updates the browser page by setting the messageLog dom element `innerHTML` to the text from the published message . This updates the browser page with the message text that was published.

Code Sample from: `chat.js`

```
var room = {
  ...
  chatCallback: function(message){
    if(!message.data){
        alert("bad message format "+message);
        return;
    }
    var messageLog=dojo.byId('messageLog');
    var from=message.data.user;
    var text=message.data.text;
    from+=":";
    // display the  chat text
    messageLog.innerHTML += from+"  "+text;
  },
  ...
}
```

Summary

The example application demonstrates the use of Dojo, Comet, Bayeux, running on the GlassFish application server.

See Also

For more information see the following resources:

- The Comet chat code used in the example application was adapted from an example originally written by Greg Wilkins.
- Cometd framework and its Bayeux protocol support in Grizzly – Jean-Francois Arcand's blog
- Using the Grizzly Comet API – Sun document on developing web applications
- Dojo Toolkit – Dojo home page
- dojo.cometd – Cometd.org page at the Dojo Foundation
- Grizzly – Project Grizzly home page
- GlassFish Application Server – home page for the GlassFish Community
- The Aquarium – news from the GlassFish Community
- Enterprise Comet: Awaken the Grizzly! – article
- Asynchronous HTTP and Comet Architectures – article
- A Comparison of Push vs Pull Ajax – article
- Comet and Reverse Ajax: The Next Generation Ajax 2.0 – book

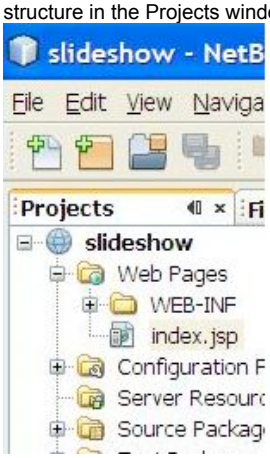
Exercise 2: Build a Comet Slideshow Application (20 minutes)

The example application in this exercise presents a photo slideshow that can be controlled by multiple users. Actions of one user affect the pages seen by other users. The application provides a chat feature that enables users to comment on photos and let other users see the comments.

Steps to Follow

Step 1: create a new Web Application project

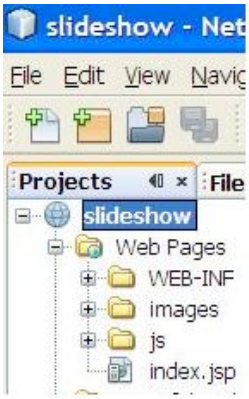
1. If NetBeans is not already running, start it.
2. Choose File > New Project (Ctrl-Shift-N) from the main menu. Under Categories, select Java Web. Under Projects, select Web Application and click Next.
3. Type `slideshow` in the Project Name field. Note that the Context Path becomes `/slideshow`.
4. Specify the Project Location to the exercise2 sub directory of this lab on your computer.
5. Under Server, select GlassFish v3. GlassFish is a Java EE5-certified application server and is bundled with the Web and Java EE insallation of NetBeans IDE.
6. Leave the Set as Main Project option selected and click Finish. The IDE creates the slideshow project folder. The project folder contains all of your sources and project metadata, such as the project's Ant build script. The slideshow project opens in the IDE. The welcome page, index.jsp, opens in the Source Editor in the main window. You can view the project's file structure in the Files window (Ctrl-2), and its logical structure in the Projects window (Ctrl-1).)



Step 2: Installing and Using Dojo With NetBeans IDE

In this lab we are not using the version of dojo bundled with Netbeans because it does not contain dojox, which we need for the dojox.grid widget and dojox.cometd. If you want to use a different version of dojo than the bundled one (like in this HOL), another easy way to use Dojo with the NetBeans IDE is to download and extract the dojo JavaScript library, and then copy it into your Netbeans project web directory: `.../web`. The dojo JavaScript library has already been downloaded and extracted for this HOL and is in the **js** folder in the hol\resources\dojo directory.

1. In your file system copy the **js** folder from the <lab_root>\exercises\exercise1\dojo directory to your slideshow web directory <lab_root>\exercises\exercise2\slideshow\web. The **js** folder contains the dojo javascript libraries.
2. You should now see the js folder in your slideshow project as below:



3. In your file system copy the **images** folder from the <lab_root>\exercises\exercise2\ directory to your slideshow web directory <lab_root>\exercises\exercise2\slideshow\web. The **images** folder contains the images for the slideshow.

Step 3:

1. In the Projects window right click the `slideshow` node (from exercise 1), and select New Other....
2. In the New File window select Web HTML and click Next
3. In the New HTML File Window, enter the HTML file name **index**, and click Finish.
4. Double click on the index.html file to open it in the editor.
5. Copy all of the html contents below and paste them to replace the contents of the index.html file in the editor.

```
<html>
<head>
   <title>Comet Slideshow</title>

   <script type="text/javascript" src="js/dojo/dojo.js"
         djConfig="parseOnLoad: true"></script>
   <script type="text/javascript" src="chat.js"></script>
   <link rel="stylesheet" type="text/css" href="chat.css">

</head>
<body>
   <h1>Comet Slideshow</h1>

   <div id="chatroom">
      <div id="chat">
          <div id="messageLog"></div>
      </div>
      <div id="slide"></div>

      <div id="input">
         <div id="join" >
            Name: <input id="sendName" type="text"/><input id="joinB" class="button" type="submit"
name="join" value="Join"/>
         </div>
         <div id="joined" class="hidden">
            <input id="previousB" class="button" type="submit" name="previous" value="Previous Slide"/>
            <input id="nextB" class="button" type="submit" name="next" value="Next Slide"/><br>
            Message: <input id="sendText" type="text"></input>
            <input id="sendB" class="button" type="submit" name="join" value="Send"/>
            <input id="leaveB" class="button" type="submit" name="join" value="Leave"/>
         </div>
      </div>
   </div>
   <p id=returnMsg></p>
</body>
</html>
```

6. In the Projects window right click the `slideshow` node (from exercise 1), and select New Other....
7. In the New File window select Web Cascading Style Sheet and click Next
8. In the New CSS File Window, enter the CSS file name **chat**, and click Finish.
9. Double click on the chat.css file to open it in the editor.
10. Copy all of the contents below and paste them to replace the contents of the chat.css file in the editor.

```
div
{
  border: 0px solid black;
}

div#chatroom
{
  background-color: #e0e0e0;
  border: 1px solid black;
  width: 45em;
}

div#chat
{
  height: 5ex;
  overflow: auto;
  background-color: #f0f0f0;
  padding: 4px;
  border: 0px solid black;
}

div#input
{
  clear: both;
  padding: 4px;
  border: 0px solid black;
  border-top: 1px solid black;
}

input#sendText
{
  width:28em;
  background-color: #e0f0f0;
}

input#sendName
{
  width:14em;
  background-color: #e0f0f0;
}

div.hidden
{
  display: none;
}

span.from
{
  font-weight: bold;
}

span.alert
{
  font-style: italic;
```

```
}
```

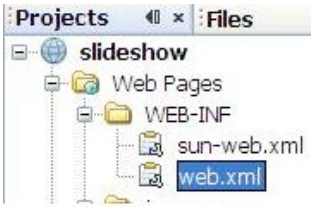
11. In the Projects window right click the `slideshow` node (from exercise 1), and select New Other....
12. In the New File window select Web JavaScriptFile and click Next
13. In the NewJavaScript File Window, enter the file name **chat**, select the slideshow **web** Folder, and click Finish.
14. Double click on the **chat.js** file to open it in the editor.
15. Copy the JavaScript contents below and paste them into the **chat.js** file in the editor.

```
dojo.require("dojox.cometd");

var room = {
    username: null,
    channel: "/chat/demo",

    join: function(name){
        dojox.cometd.init("cometd");
        room.username=name;
        dojo.byId('join').className='hidden';
        dojo.byId('joined').className='';
        dojo.byId('sendText').focus();
        dojox.cometd.subscribe(room.channel, room, "chatCallback");
        dojox.cometd.publish(room.channel, {
            user: room.username,
            text: " has joined"
        });
    },
    leave: function(){
        if (room.username==null)
            return;
        dojox.cometd.unsubscribe(room.channel, room, "chatCallback");
        dojox.cometd.publish(room.channel, {
            user: room.username,
            text: " has left"
        });
        dojox.cometd.disconnect();
        // switch the input form
        dojo.byId('join').className='';
        dojo.byId('joined').className='hidden';
        dojo.byId('sendName').focus();
        room.username=null;
    },
    chat: function(text){
        dojox.cometd.publish(room.channel, {
            user: room.username,
            text: text
        });
    },
    next: function(text){
        dojox.cometd.publish(room.channel, {
            slide: text
        });
    },
    chatCallback: function(message){
        if(!message.data){
            alert("bad message format "+message);
            return;
        }
        var chat=dojo.byId('chat');
        var slide=dojo.byId('slide');
        var messageLog=dojo.byId('messageLog');
        var from=message.data.user;
```

```
        var text=message.data.text;
        var slideUrl=message.data.slide;
        from+=":";
        // display the next image, or chat text
        if(slideUrl){
            slide.innerHTML ="<img src='" + slideUrl + "' style='width: 540px; height: 380px;' />";
        }else{
            messageLog.innerHTML += "<span class=\"from\">"+from+" </span><span class=\"text\">"+text+"</span><br/>";
        }
        chat.scrollTop = chat.scrollHeight - chat.clientHeight;
    },
    //replace to make request to the slides web service
    loadSlides: function (){
        room.slideUrls=[
        "/slideshow/images/image0.jpg",
        "/slideshow/images/image1.jpg",
        "/slideshow/images/image2.jpg",
        "/slideshow/images/image3.jpg",
        "/slideshow/images/image4.jpg",
        "/slideshow/images/image5.jpg"];
    },

    init: function(){
        room.loadSlides();
        var i=room.slideUrls.length;

        dojo.connect(dojo.byId("joinB"),"onclick",function(){
            if (!dojo.byId("sendName").value.length) {
                alert("Please enter Name");
                return;
            }
            room.join(dojo.byId('sendName').value);
        });
        dojo.connect(dojo.byId("sendB"),"onclick",function(){
            if (!dojo.byId("sendText").value.length) {
                alert("Please enter Text");
                return;
            }
            room.chat(dojo.byId("sendText").value);
        });
        dojo.connect(dojo.byId("nextB"),"onclick",function(){
            if (i>=room.slideUrls.length)
                i=0;
            else
                i++;
            room.next( room.slideUrls[i]);
        });
        dojo.connect(dojo.byId("previousB"),"onclick",function(){
            if (i<=0)
                i=room.slideUrls.length-1 ;
            else
                i--;
            room.next( room.slideUrls[i]);
        });
        dojo.connect(dojo.byId("leaveB"),"onclick",function(){
            room.leave();
        });
    }
};

dojo.addOnLoad(room, "init");
dojo.addOnUnload(room,"leave");
```

Edit the web.xml file to change the welcome page:

1. double click on the web.xml file in the slideshow WEB-INF directory



2. Click on the Pages tab. Change the Welcome File from index.jsp to **index.html .**

Step 4: Configuring for Comet

The goal of this part of exercise3 is to configure Glassfish for comet and your web project for the Grizzly Cometd servlet.

1. **Enabling Bayeux in GlassFish**
   To enable Bayeux on GlassFish, add the lines shown in red below to your Web application's `web.xml` file:

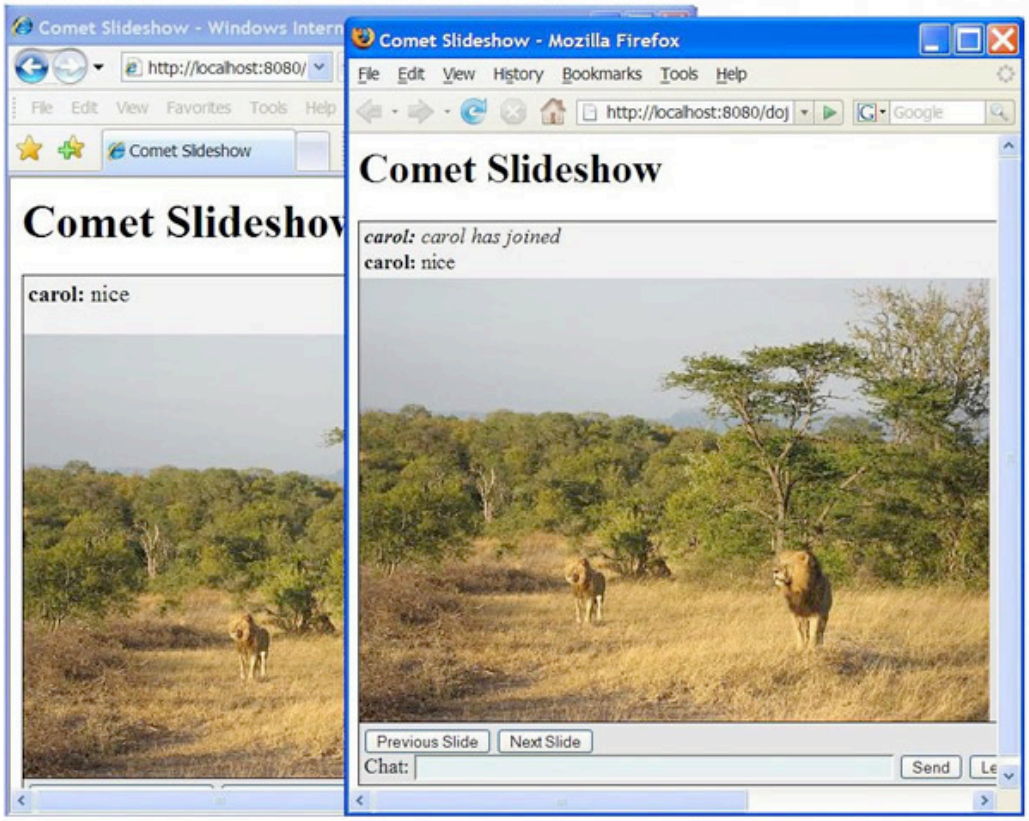
   Code Sample from: `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" ...>

 <servlet>
   <servlet-name>Grizzly Cometd Servlet</servlet-name>
   <servlet-class>
      com.sun.grizzly.cometd.servlet.CometdServlet
   </servlet-class>
   <init-param>
     <param-name>expirationDelay</param-name>
     <param-value>-1</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
   <servlet-name>Grizzly Cometd Servlet</servlet-name>
   <url-pattern>/cometd/*</url-pattern>
 </servlet-mapping>

  <session-config>
      <session-timeout>
            30
      </session-timeout>
  </session-config>
  <welcome-file-list>
        <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

   Every request sent to your `war` file's *`context-path`*`/cometd/` will be serviced by the Grizzly Bayeux runtime.

Step 5: Run the Project

1. Right click the slideshow node in the Projects window, and select Run.
2. When you run the project, your browser should display the opening page of the dojo comet Sample Application (at http://localhost:8080 /slideshow/).
3. Open another browser, preferably with different profile or different brand of browser, and type http://localhost:8080/slideshow/ to access the application.
   Comet slideshow page, which allows the users to slideshow at the same time.



Different Browsers Sharing Photos in Comet slideshow

Understanding the code

Using dojox.cometd

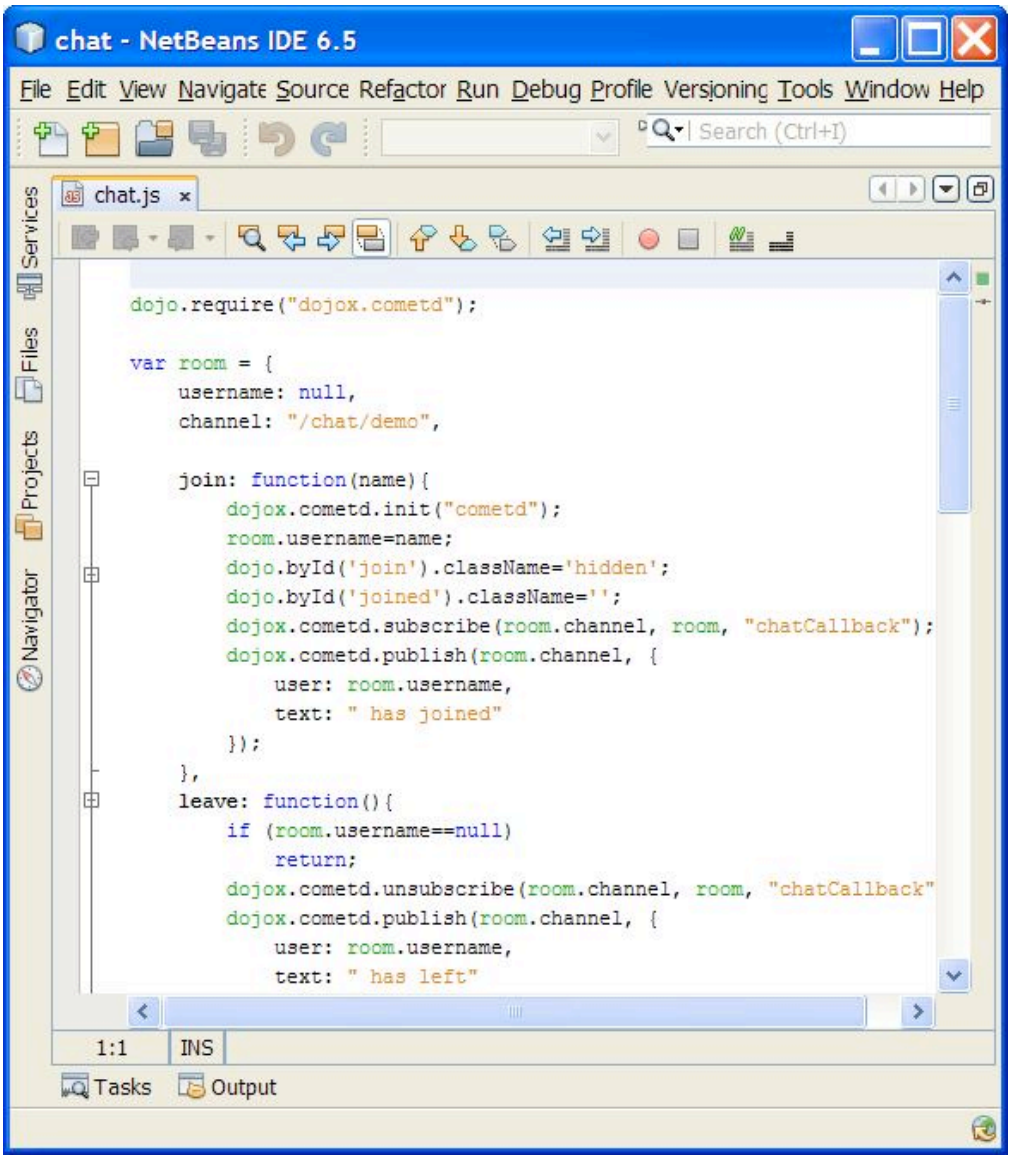
Loading Base Dojo and Required Modules Into the Application

This script element: src="src/dojo/dojo.js" loads the base dojo script that gives you access to the Dojo functionality, other dojo modules are loaded as required with the the `dojo.require` function explained later.

Code Sample from: `index.html`

```
<script type="text/javascript" src="js/dojo/dojo.js"></script>
<script type="text/javascript" src="chat.js"></script>
```

The rest of the JavaScript code for this application is in the file **chat.js**. Note that NetBeans IDE 6.5 has a very capable JavaScript editor, shown below editing the `chat.js` file:



In `chat.js`, the application uses the `dojo.require` function (similar to the `import` directive in Java) to specify which Dojo modules to load.

Code Sample from: `chat.js`

```
dojo.require("dojox.cometd");
```

Dojo is organized into three major layers: Dojo Core, Dijit, and DojoX. DojoX builds on Dojo Core and provides newer extensions to the Dojo Toolkit. DojoX cometd implements a Bayeux protocol client for use with a Bayeux server.

Initializing a Connection Between the Dojo Client and the Grizzly Bayeux Servlet

Upon loading the slideshow application, a user can enter a username and join a slideshow session, as shown in Figure 3.

Figure 3: *User Login Page*

When a user clicks the Join button, the `join` JavaScript function is called. In the `join` function, the call to `dojox.cometd.init` initializes a connection to the given Comet server — in this case with the GlassFish Grizzly Bayeux servlet. Note that **"cometd"** is the URL-pattern for the Grizzly Cometd servlet configured in the `web.xml` file for the application.

Code Sample from: `chat.js`

```
var room = {
...
   join: function(name){

     dojox.cometd.init("cometd");
     dojox.cometd.subscribe("/chat/demo", room, "chatCallback");
     dojox.cometd.publish("/chat/demo",
       {  user: room.username,
          text: " has joined"});
   }
```

The `dojox.cometd.subscribe` line subscribes the `chatCallback` callback function to the `/chat/demo` channel. Whenever a message is sent to the `/chat/demo` channel, the `chatCallback` function is called. The `dojox.cometd.publish` line publishes a message that the user whose name was entered with the Join button has joined the `/chat/demo` channel. All subscribers to the `/chat/demo` channel receive the message.

Publishing the Next Slide for the Comet chat

When the user clicks the Next Slide button, a JavaScript function is called that publishes the URL for the next slide, as shown in Figure 4.

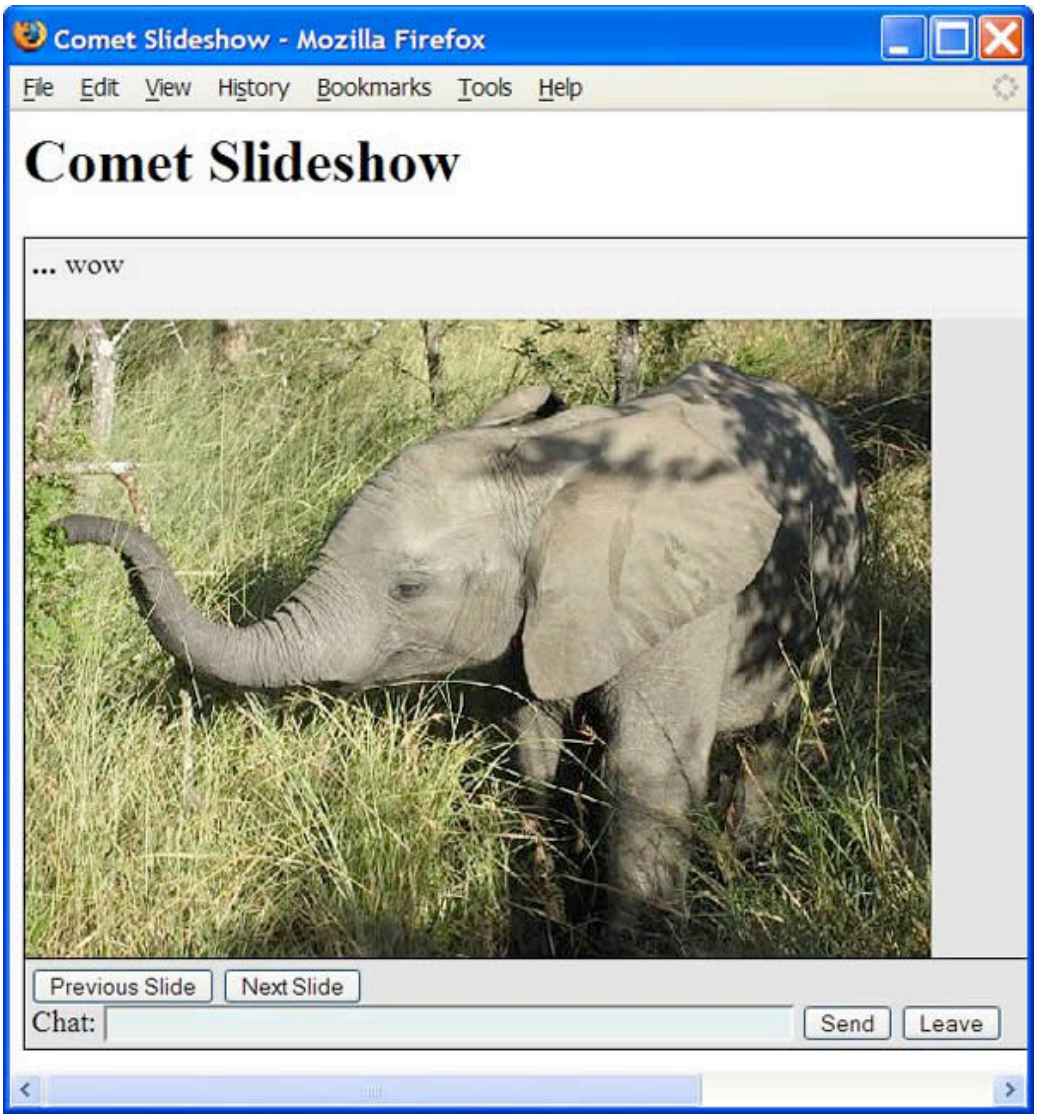


Figure 4: *Viewing the Next Slide*

Code Sample from: `index.html`

```
<input id="previousB" class="button" type="submit" name="previous" value="Previous Slide"/>
<input id="nextB" class="button" type="submit" name="next" value="Next Slide"/><br>
```

The JavaScript function that is called when the user clicks the Next Slide button is shown in the following code sample. This function calls `room.next`, which passes the URL for the next slide. The function then increments the index for the next slide. The URLs for the slides are stored in the `slideUrls` array, which is loaded at initialization in the `room.loadSlides()` function.

Code Sample from: `chat.js`

```
var room = {
 ..
dojo.connect(dojo.byId("nextB"),"onclick",function(){
  if (i>=room.slideUrls.length)
      i=0;
  else
      i++;
  room.next( room.slideUrls[i]);
});
```

The function `room.next`, shown in the following code sample, calls `dojox.cometd.publish` to publish the next slide URL (which it receives as an input argument) to the `/chat/demo` channel. All subscribers to the `/chat/demo` channel receive this message.

Code Sample from: `chat.js`

```
var room = {
   channel: "/chat/demo",
   ...
   next: function(text){
        dojox.cometd.publish(room.channel, {
            slide: text
        });
   },
  ...
}
```

When a message is published to a Bayeux channel on the server, it is delivered to all clients subscribed to that channel — in this case, to the `/chat/demo` channel.

In the `room.join` function described earlier, `dojox.cometd.subscribe("/chat/demo", room, "chatCallback")` was called to subscribe the `chatCallback` callback function to the `/chat/demo` channel. The `chatCallback` function, shown in the following code sample, is called with the published message as an input argument. The `chatCallback` function updates the browser page by setting the `slide` dom element `innerHTML` to an HTML `img` tag with the slide URL from the published message `"<img src='" + slideUrl + "'/>"`. This message updates the browser page with the image that corresponds to the slide URL that was published.

Code Sample from: `chat.js`

```
 chatCallback: function(message){
  if(!message.data){
      alert("bad message format "+message);
      return;
  }
  var chat=dojo.byId('chat');
  var slide=dojo.byId('slide');
  var messageLog=dojo.byId('messageLog');
  var from=message.data.user;
  var text=message.data.text;
  var slideUrl=message.data.slide;
  from+=":";
  // display the next image, or chat text
  if(slideUrl){
      slide.innerHTML ="<img src='" + slideUrl + "' style='width: 540px; height: 380px;' />";
  }else{
      messageLog.innerHTML += "<span class=\"from\">"+from+" </span><span class=\"text\">"+text+"</sp
  }
  chat.scrollTop = chat.scrollHeight - chat.clientHeight;
 },
```

Loading the Slide URLs

The `dojo.addOnLoad` function enables you to call a function after a page has loaded and after Dojo has finished its initialization. This application uses `dojo.addOnLoad` to call the `init` function. The `init` function calls the `loadSlides` function, which initializes the `slideUrls` object, as shown in the following code sample.

Code Sample from: `chat.js`

```
var room = {

    slideUrls: null,
    ...

    loadSlides: function (){
        room.slideUrls=[
        "/slideshow/images/image0.jpg",
        "/slideshow/images/image1.jpg",
        "/slideshow/images/image2.jpg",
        "/slideshow/images/image3.jpg",
        "/slideshow/images/image4.jpg",
        "/slideshow/images/image5.jpg"];
    },

   init: function(){
     room.loadSlides();
    ...
  }
  ...
};

dojo.addOnLoad(room, "init");
```

Summary

The example application demonstrates the use of Dojo, Comet, Bayeux, running on the GlassFish application server.

See Also

For more information see the following resources:

- The Comet chat code used in the example application was adapted from an example originally written by Greg Wilkins.
- Cometd framework and its Bayeux protocol support in Grizzly – Jean-Francois Arcand's blog
- Using the Grizzly Comet API – Sun document on developing web applications
- Dojo Toolkit – Dojo home page
- dojo.cometd – Cometd.org page at the Dojo Foundation
- Grizzly – Project Grizzly home page
- GlassFish Application Server – home page for the GlassFish Community
- The Aquarium – news from the GlassFish Community
- Enterprise Comet: Awaken the Grizzly! – article
- Asynchronous HTTP and Comet Architectures – article
- A Comparison of Push vs Pull Ajax – article
- Comet and Reverse Ajax: The Next Generation Ajax 2.0 – book

Exercise 3: Server Interact with Chat Application (20 minutes)

In this exercise, you will learn how to implement a servlet to interact with the Chat application by using Grizzly comet framework API.

Background Information

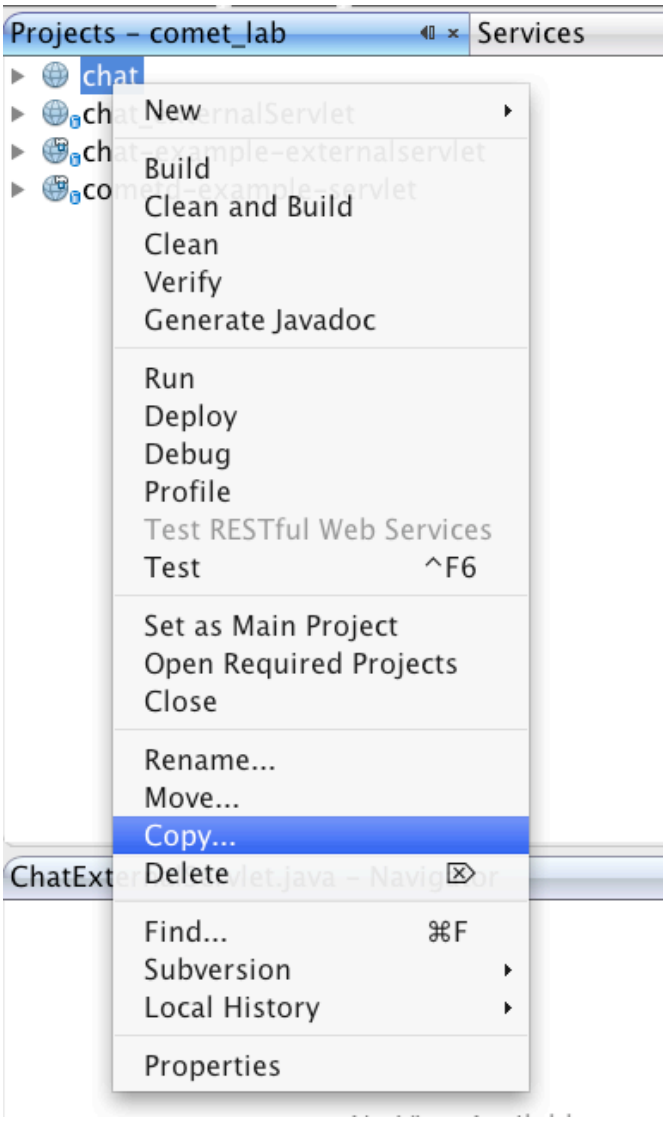
For the Chat application developed in exercise 1, we only need to implement the application in the client with Dojo or other Ajax components. It is very easy to develop as it is server agnostic if the server supports Bayeux and cometd. However, if we want to have some control on the server side like receiving some messages from the server, or interacting with server in certain way, we will need to use Grizzly comet framework to implement the interaction between server and client.

Steps to Follow

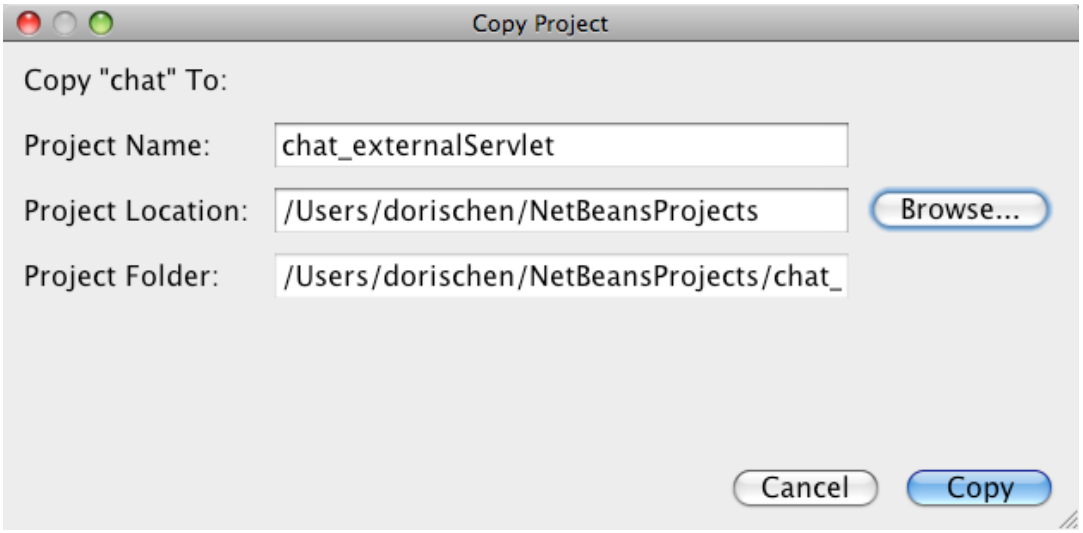
Step 1: Continue on chat Web Application project

We will continue on the chat application created in exercise 1. This will all run on our familiar Glassfish server and many of the steps you will be familiar with from previous exercises.

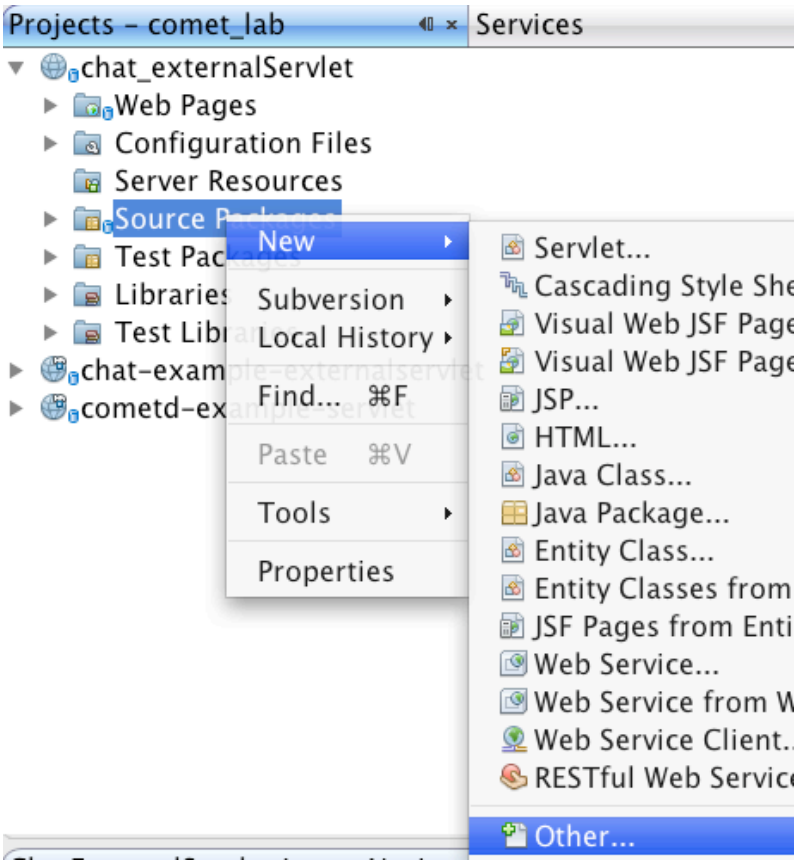
1. If NetBeans is not already running, start it.
2. Let's make a copy of the chat application. Right click on chat application and select copy from the list as illustrated in the following figure.



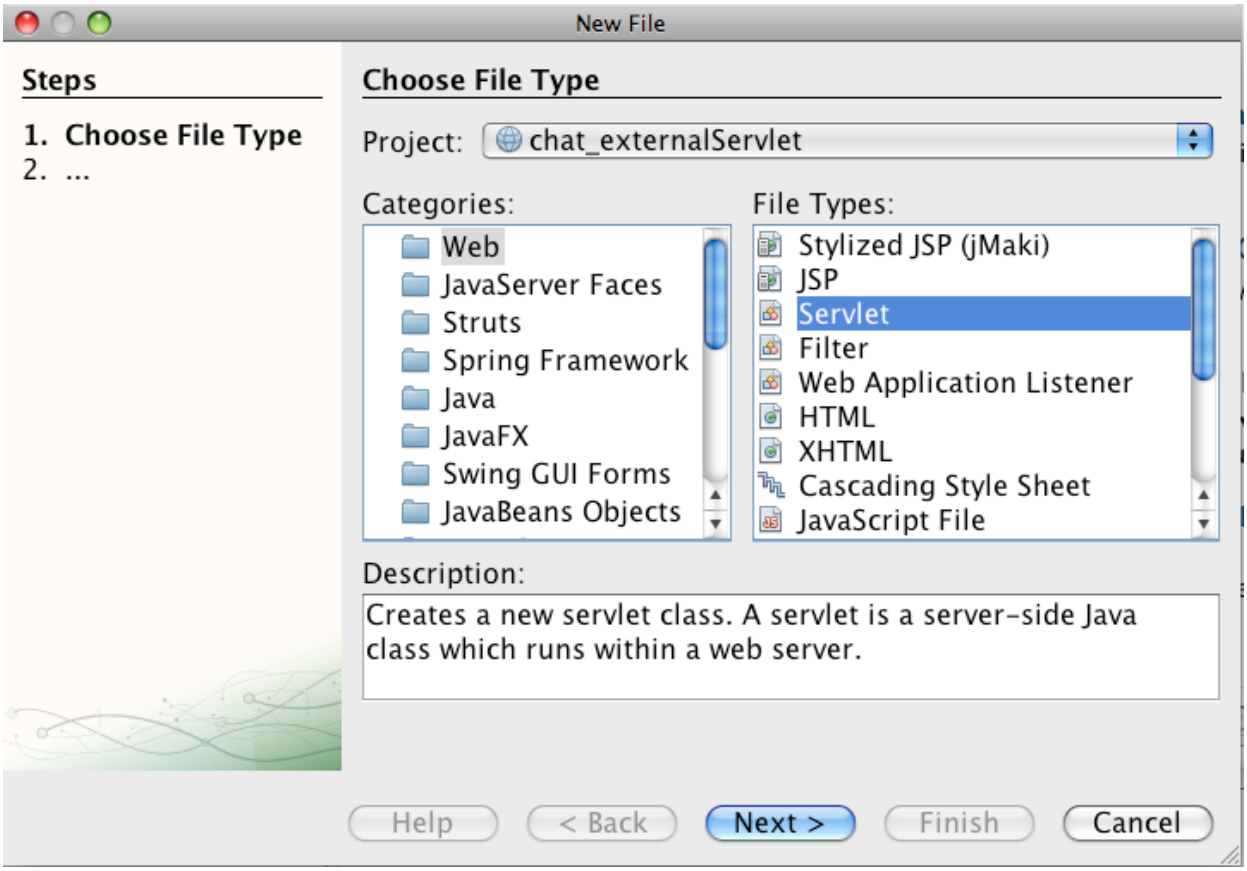
3. In copy project panel, type chat_externalServlet as the project name and choose location where you want to put this project and then select copy to execute copy project.



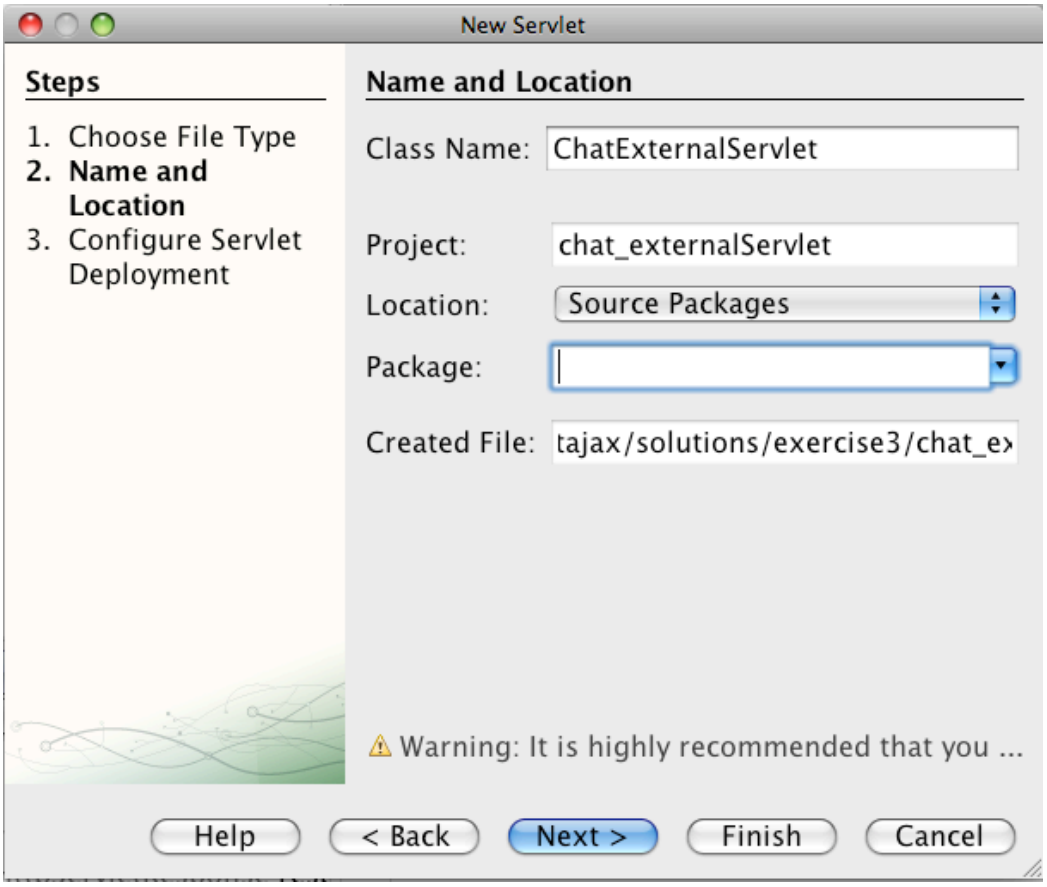
4. Now create a new servlet in the project. Right click Source Packages and choose New from the list and then select Other as shown below.



5. In New File panel , select Web as the Categories and Servlet as the File Types and then select Next to continue



6. Now configure the New Servlet. Type ChatExternalServlet as the Class Name, choose <default package> as the Package, and then select **Finish** to create the servlet.



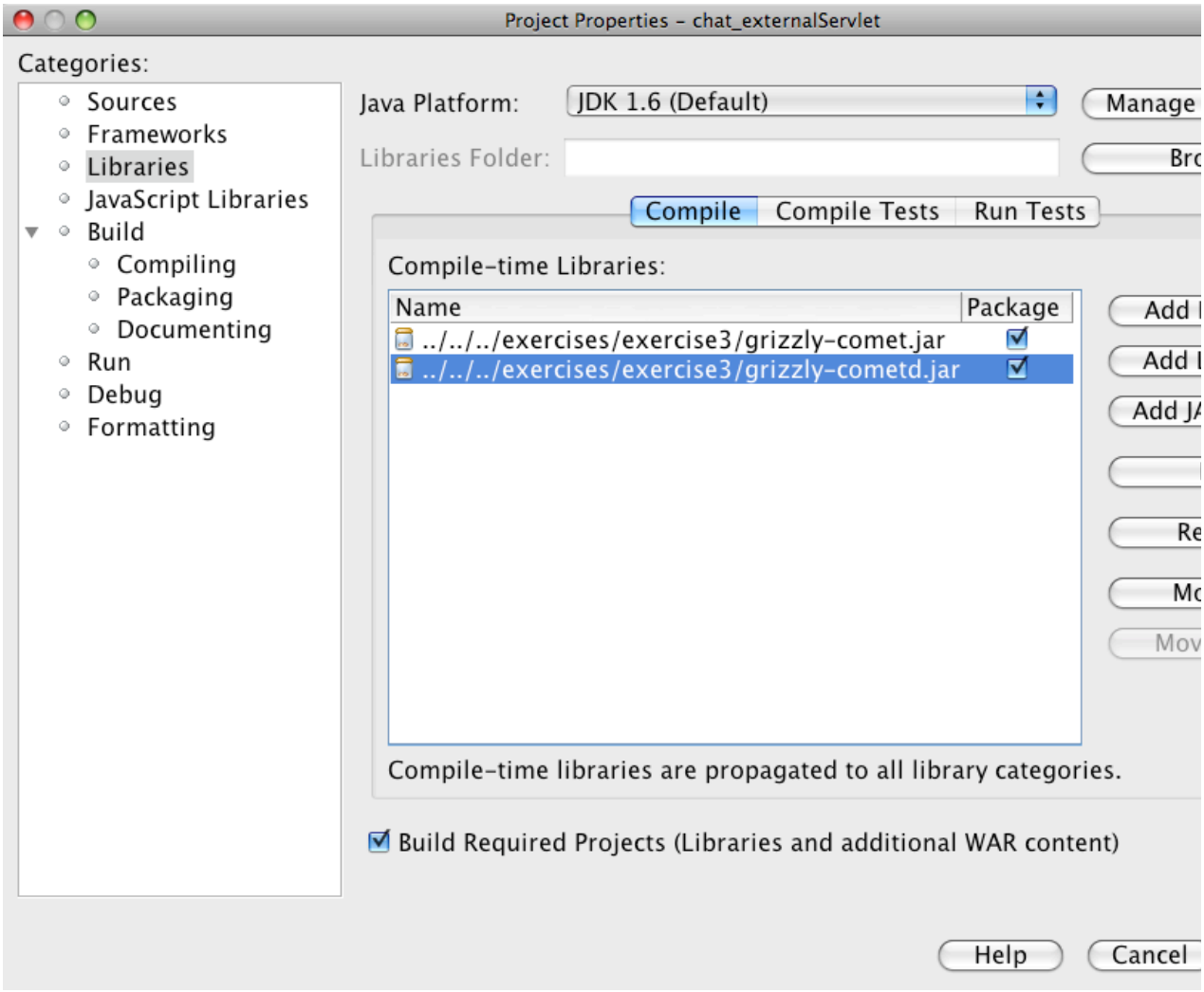
7. Now we need to add the Comet libraries to our project.

    1. Right click the `chat_externalServlet` node and select Properties from the list.

    2. Find the project's Libraries in the left hand side of the window and select it.

    3. Click Add JAR/Folder button on the right side. Select the following two JAR's from the <lab_root>/exercises/exercise3. Click OK to finish Library selection.

        - grizzly-comet.jar
        - grizzly-cometd.jar



Step 2: Develop A Simple Servlet

We will develop a simple servlet that will interact with the chat application via a simple HTTP request.

1. Open ChatExternalServlet.java under Sources Package in NetBeans.

2. The following is a simple servlet to start with with. Copy and paste the code below into ChatExternalServlet.java in NetBeans. Pay particular attention to the code in bold.

```
import com.sun.grizzly.comet.CometContext;
import com.sun.grizzly.comet.CometEngine;
import com.sun.grizzly.cometd.bayeux.Data;
import com.sun.grizzly.cometd.bayeux.DeliverResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ScheduledThreadPoolExecutor;
```

```
import java.util.concurrent.TimeUnit;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Simple Servlet that update the chat application via a simple Http request
 *
 * <code>
 * http://localhost:port/cometd/ChatExternalServlet?user=me&message=hello
 * </code>
 *
 * @author Doris Chen
 */
public class ChatExternalServlet extends HttpServlet {


 /**
 * All request to that channel will be considered as cometd enabled.
 */
 private String channel ="/chat/demo";


 /**
 * Initialize the Servlet.
 */
 @Override
 public void init(ServletConfig config) throws ServletException {
 super.init(config);

 if (config.getInitParameter("channel") != null){
 channel = config.getInitParameter("channel");

 }
 }



 @Override
 public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, I
 doPost(request, response);
 }

 /**
 * See
 * @param request
 * @param response
 * @throws javax.servlet.ServletException
 * @throws java.io.IOException
 */
 @Override
 public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
 // message and user are part of the simple HTTP request will invoke late.
 String message = request.getParameter("message");
 String user = request.getParameter("user");
 ServletOutputStream out = response.getOutputStream();

 //create the comet context with the specified channel.
 CometEngine engine = CometEngine.getEngine();
 CometContext context = engine.getCometContext(channel);

 if (context != null && message != null) {
 Map<String, Object> map = new HashMap<String, Object>();
 // text and user are defined entry in the previous chat application.
 map.put("text", message);
 map.put("user", user);

 //Construct a Bayeux response message using Data and DeliverResponse.
 Data data = new Data();
 data.setMapData(map);

 DeliverResponse deliverResponse = new DeliverResponse();
 deliverResponse.setChannel(channel);
 deliverResponse.setData(data);
 deliverResponse.setLast(true);
 deliverResponse.setFollow(true);
 deliverResponse.setFinished(true);
```

```
 //send the Bayeux message out by invoking notify().
 context.notify(deliverResponse);

 out.println("Data is sent.");

 } else {
 out.println("No data is sent.");
 }
 }
}
```

3. Understanding the code. In the example above, we will generate a cometd Bayeux message in a servlet ChatExternalServlet.java.
   1. First, we import all the Grizzly Comet Framework packages at the beginning:

      ```
      import com.sun.grizzly.comet.CometContext;
      import com.sun.grizzly.comet.CometEngine;
      import com.sun.grizzly.cometd.bayeux.Data;
      import com.sun.grizzly.cometd.bayeux.DeliverResponse;
      ```

   2. In doPost() implementation, one can get a CometContext as follows:

      ```
      CometEngine engine = CometEngine.getEngine();
      CometContext context = engine.getCometContext(channel);
      ```

      In our case, the channel is "/chat/demo" which is defined in exercise 1 chat application.

   3. In doPost(), we construct a Bayeux response message by using classes in package com.sun.grizzly.cometd.bayeux. The classes that we need to use are DeliverResponse and Data. It is constructed as follows:

      ```
      Map<String, Object> map = new HashMap<String, Object>();
      map.put("text", message);
      map.put("user", user);
      Data data = new Data();
      data.setMapData(map);

      DeliverResponse deliverResponse = new DeliverResponse();
      deliverResponse.setChannel(channel);
      deliverResponse.setData(data);
      deliverResponse.setLast(true);
      deliverResponse.setFollow(true);
      deliverResponse.setFinished(true);
      ```

      Note that

      - text and user headers are defined in exercise 1 chat application.
      - deliverResponse.setLast(true) indicates that this is the last Bayeux message in this Http response.
      - deliverResponse.setFollow(true) indicates that this is not the first Bayeux message in this Http response.
      - deliverResponse.setFinished(true) indicates that the underlying connection needs to be resumed.

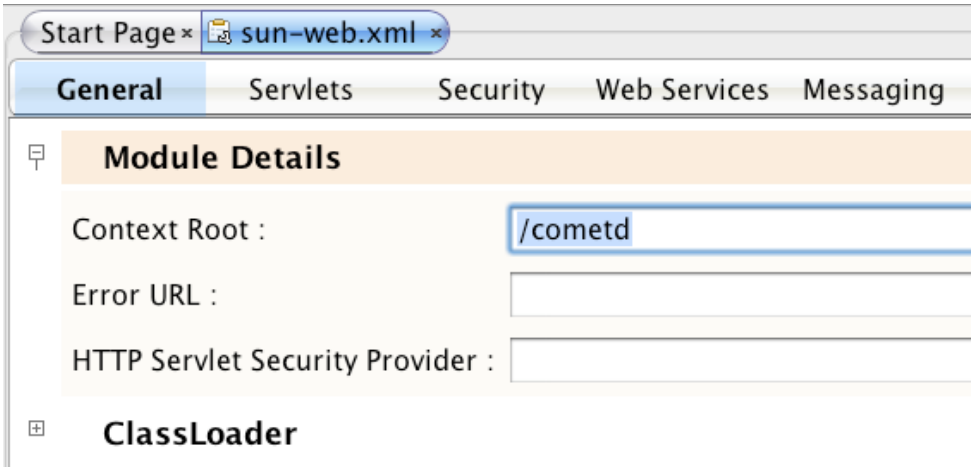
      Then we can send the Bayeux message as follows:

      ```
      context.notify(deliverResponse);
      ```

Step 3: Run the project

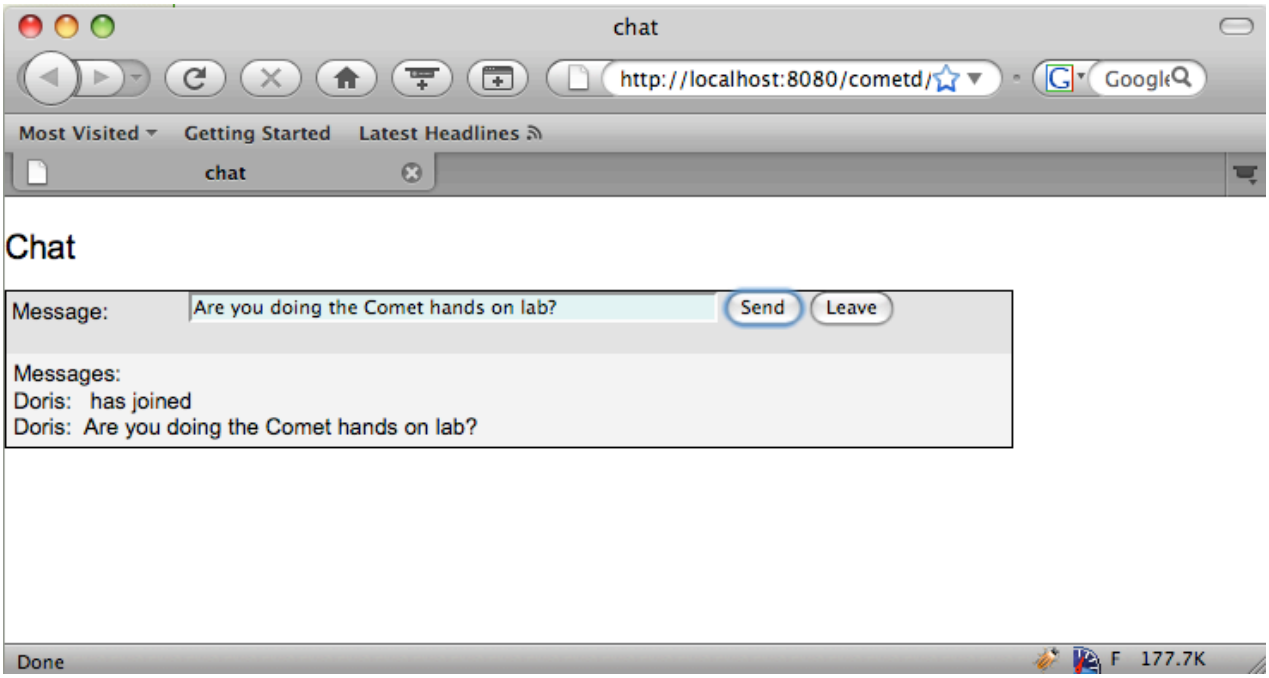
With the NetBeans IDE

1. Open sun-web.xml in NetBeans. The sun-web.xml is located in chat_ExternalServlet project under /Web Pages/WEB-INF. In Context Root area, type /cometd so it will enable comet servlet once the project is invoked.



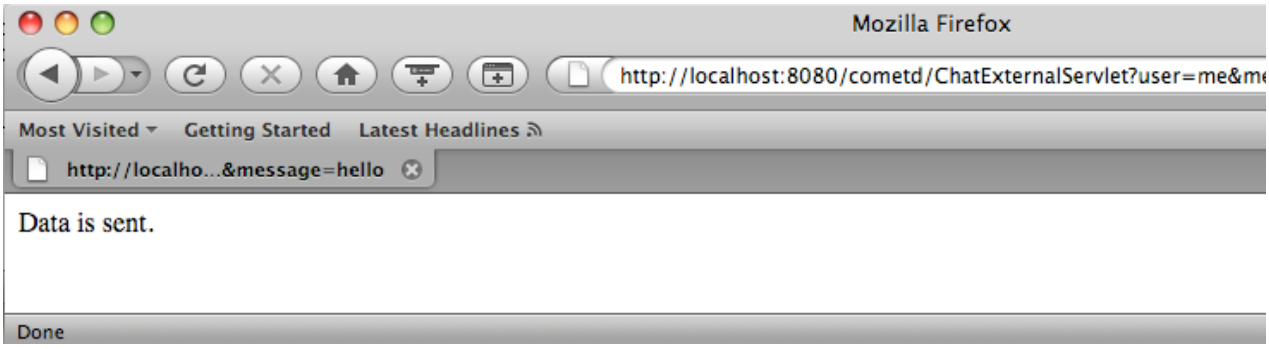
2. Make sure the comet is already enabled in GlassFish and GlassFish is already started.
   See exercise 0 for detail on how to enable comet support in Glassfish if you haven't done son.

3. In the Projects tab, right click and select Run. Run option will compile, build and invoke the application in the default browser, let's call it browser A1 as shown below. Enter name to join the chat and then enter some message. You will see the chat application running as you have seen in exercise 1.

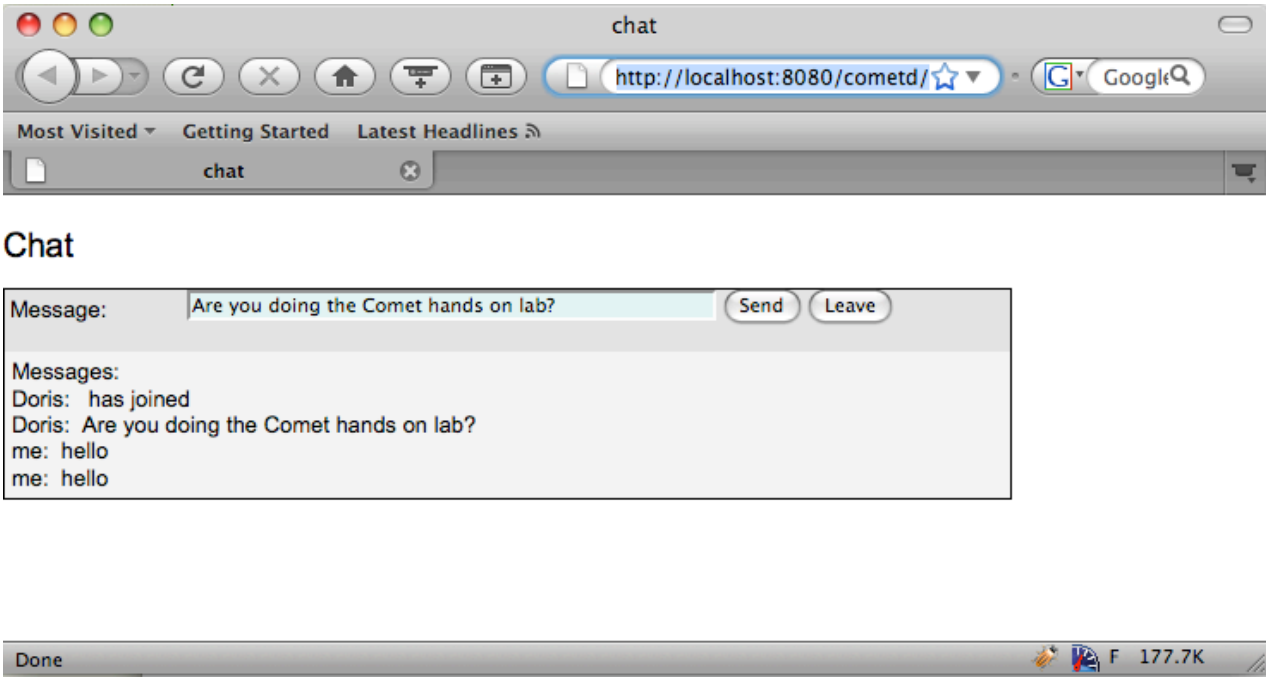


4. Open another browser A2, preferably with different profile or different brand of browser, and type http://localhost:8080/cometd to access the application. Enter name to join the chat and then enter some message.

5. Open another browser B and type http://localhost:8080/cometd/ChatExternalServlet?user=me&message=hello and hit return. It will invoke the servlet ChatExternalServlet. Notice user=me&message=hello are in the HTTP request. The following shows the response in browser B.



The following is the response in browser A1: **me: hello** message is sent by the ChatExternalServlet and received in client chat browser. Browser A2 will have the same response.

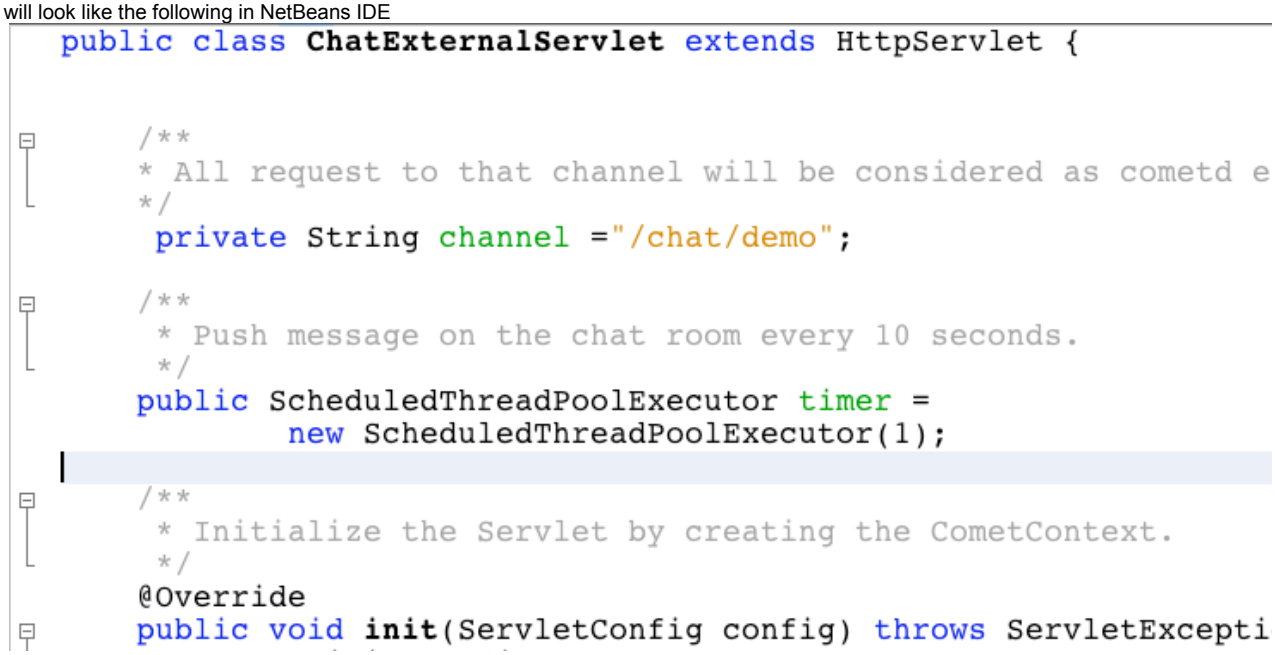


Step 4: Implement Periodic Message in Servlet ChatExternalServlet

1. We will add a timer service to generate some periodic maintenance message to the chat client. In ChatExternalServlet.java, first add the following code to define the timer.

```
/**
 * Push message on the chat room every 10 seconds.
 */
public ScheduledThreadPoolExecutor timer =
new ScheduledThreadPoolExecutor(1);
```

It will look like the following in NetBeans IDE



2. In servlet init() method, add a timer service routine like the following code template. You will need to complete the code based on the instruction in red. This wake up message is scheduled to push to the chat client every 10 seconds.

```
timer.scheduleAtFixedRate(new Runnable(){
```

```
public void run(){
CometEngine engine = CometEngine.getEngine();
CometContext context = engine.getCometContext(channel);

if (context != null) {
Map<String, Object> map = new HashMap<String, Object>();
map.put("text", "Wake up call form the chatroom :-)");
map.put("user", "ChatMonitor");
//Construct Bayeux Data by creating the Data and add the map into it.

DeliverResponse deliverResponse = new DeliverResponse();
//Construct DeliveryResponse by setChannel, setData, setLast, setFollow, and setFinished.

try{
context.notify(deliverResponse);

} catch(IOException ex){
ex.printStackTrace();
}
}
}
}, 10, 10, TimeUnit.SECONDS);
```

After you have completed the code in inti() method, it will look like the following in NetBeans.

```
@Override
public void init(ServletConfig config) throws ServletExcepti
    super.init(config);

    if (config.getInitParameter("channel") != null){
         channel = config.getInitParameter("channel");

    }

    timer.scheduleAtFixedRate(new Runnable(){
            public void run(){
                CometEngine engine = CometEngine.getEngine()
                CometContext context = engine.getCometContex

                if (context != null) {
                    Map<String, Object> map = new HashMap<St
                    map.put("text", "Wake up call form the c
                    map.put("user", "ChatMonitor");
                    Data data = new Data();
                    data.setMapData(map);

                    DeliverResponse deliverResponse = new De
                    deliverResponse.setChannel(channel);
```

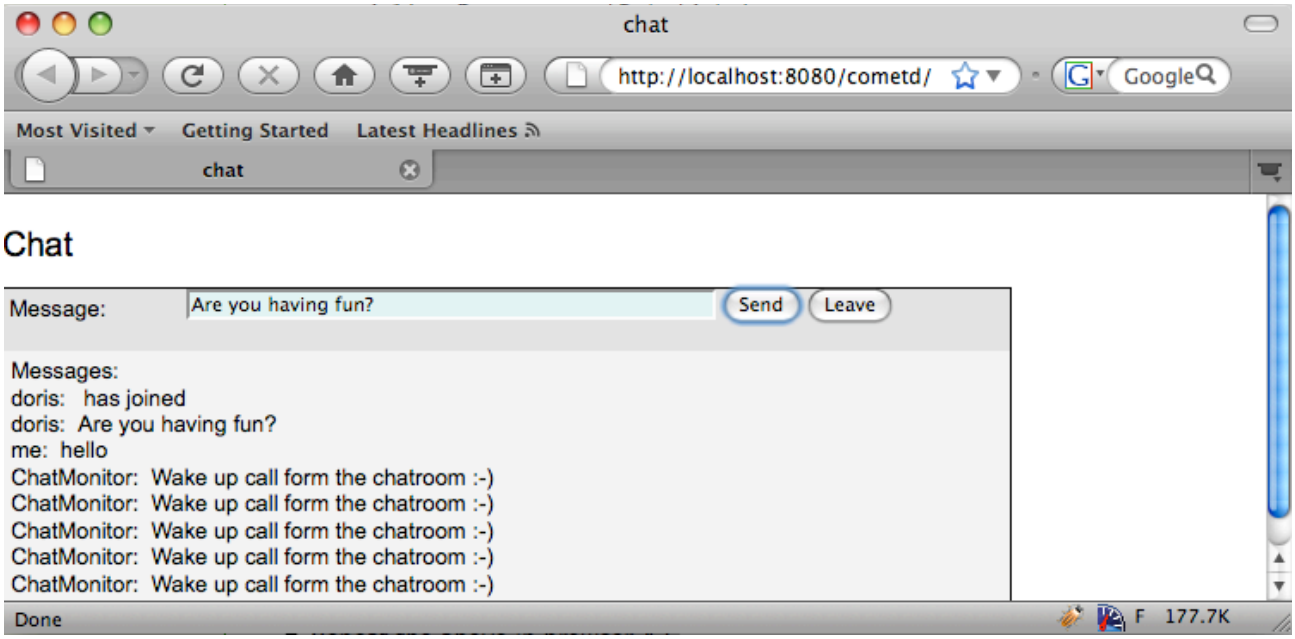
3. One last step is to clean up the resources of the timer. Add the following destroy() method to the servlet.

```
@Override
public void destroy(){
    timer.shutdown();
}
```

Step 5: Run the Project

With the NetBeans IDE

Now repeat the steps in Step 3 from 2 to 5. Now you will see the timer is generating all the message to the chat client as shown below.



Summary

In this exercise, you have learned how to easily develop a servlet to interact with the chat clients by using Grizzly Comet Framework APIs.

Exercise 4: Build a Two Way Tic-tac-toe Game (40 minutes)

Using AJAX and Comet, you will learn how to create a simple tic-tac-toe game in which two people play while other people can watch the game via their browser. This exercise involves building the servlet that implements the Comet event handling and then building the client side html code to play the game. You will learn how you can customize the code on the server side to handle your specific application requirements, and how you can use Comet technology to build back-end capibilities enabling a two-player distributed game environment.

Background Information

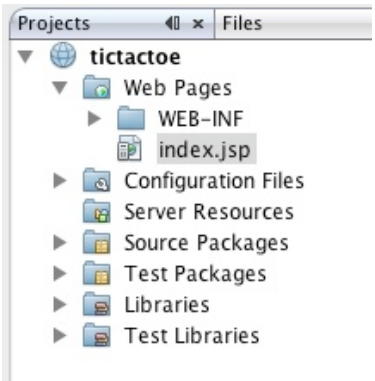
This exercise is going to use the *long polling* method to develop our Comet-enabled web application. The client side of our application, our tic-tac-toe board, is going to use some embedded JavaScript to allow us to connect to the server. The server side of our application consists of a servlet that listens for updates from clients (our players), processes the player's moves, validates them, checks if there is a winner of the game, and finally writes JavaScript code to the clients that updates the board.

Steps to Follow

Step 1: create a new Web Application project

In this portion of the exercise we will create the web application that will contain our server side Comet application and our client side tic-tac-toe game. This will all run on our familiar Glassfish server and many of the steps you will be familiar with from previous exercises.

1. If NetBeans is not already running, start it.
2. Choose File > New Project (Ctrl-Shift-N) from the main menu. Under Categories, select Java Web. Under Projects, select Web Application and click Next.
3. Type `tictactoe` in the Project Name field. Note that the Context Path becomes `/tictactoe`.
4. You may be prompted to enter the Glassfish username and password. If so they are:
   - Username: admin
   - Password: adminadmin
5. Specify the Project Location to the exercise4 sub directory of this lab on your computer.
6. Under Server, select GlassFish v3. GlassFish is a Java EE5-certified application server and is bundled with the Web and Java EE insallation of NetBeans IDE.
7. Leave the Set as Main Project option selected and click Finish. The IDE creates the tictactoe project folder. The project folder contains all of your sources and project metadata, such as the project's Ant build script. The tictactoe project opens in the IDE. The welcome page, index.jsp, opens in the Source Editor in the main window. You can view the project's file structure in the Files window (Ctrl-2), and its logical structure in the Projects window (Ctrl-1).)



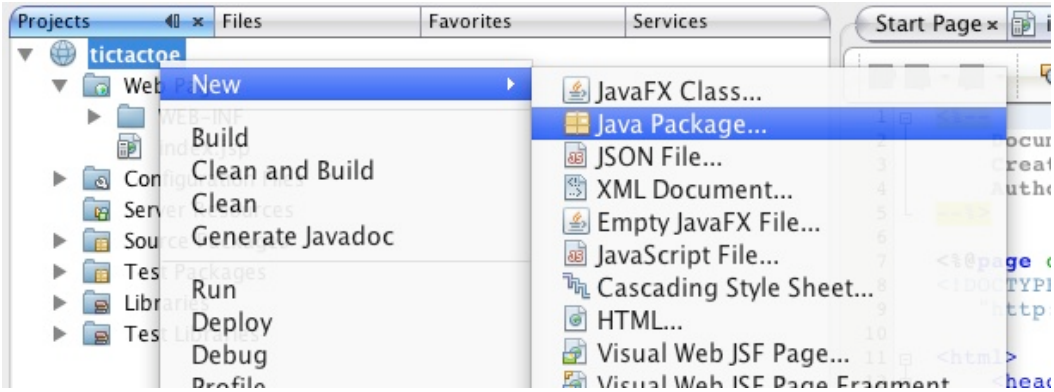
Step 2: Creating the server side Web Component

Now we will begin building the servlet that will handle our Comet requests. This will start out as a standard servlet but we will customize the `doGet` and `doPost` methods with our Comet code. Here are the general steps you will take to build your web component.

- Create a web component to support Comet requests.
- Register the component with the Comet engine.
- Define a Comet handler that updates the client

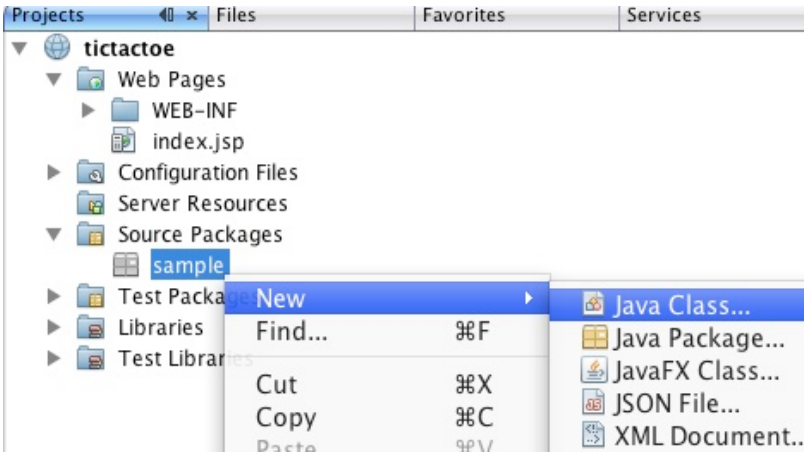
- Add the Comet handler to the Comet context
- Notify the Comet handler of an event using the Comet context

1. In the Projects window right click the Source Packages in your `tictactoe` node (from Step 1) and select New Java Package...



2. In the New Java Package Window, enter the Package name **sample**, and click Finish.

3. In the Projects window right click the **sample** package and select New Java Class...



4. In the New Java Class Window, enter the Class Name **TTTComet**, and click Finish.

5. Double click on the TTTComet.java file to open it in the editor.

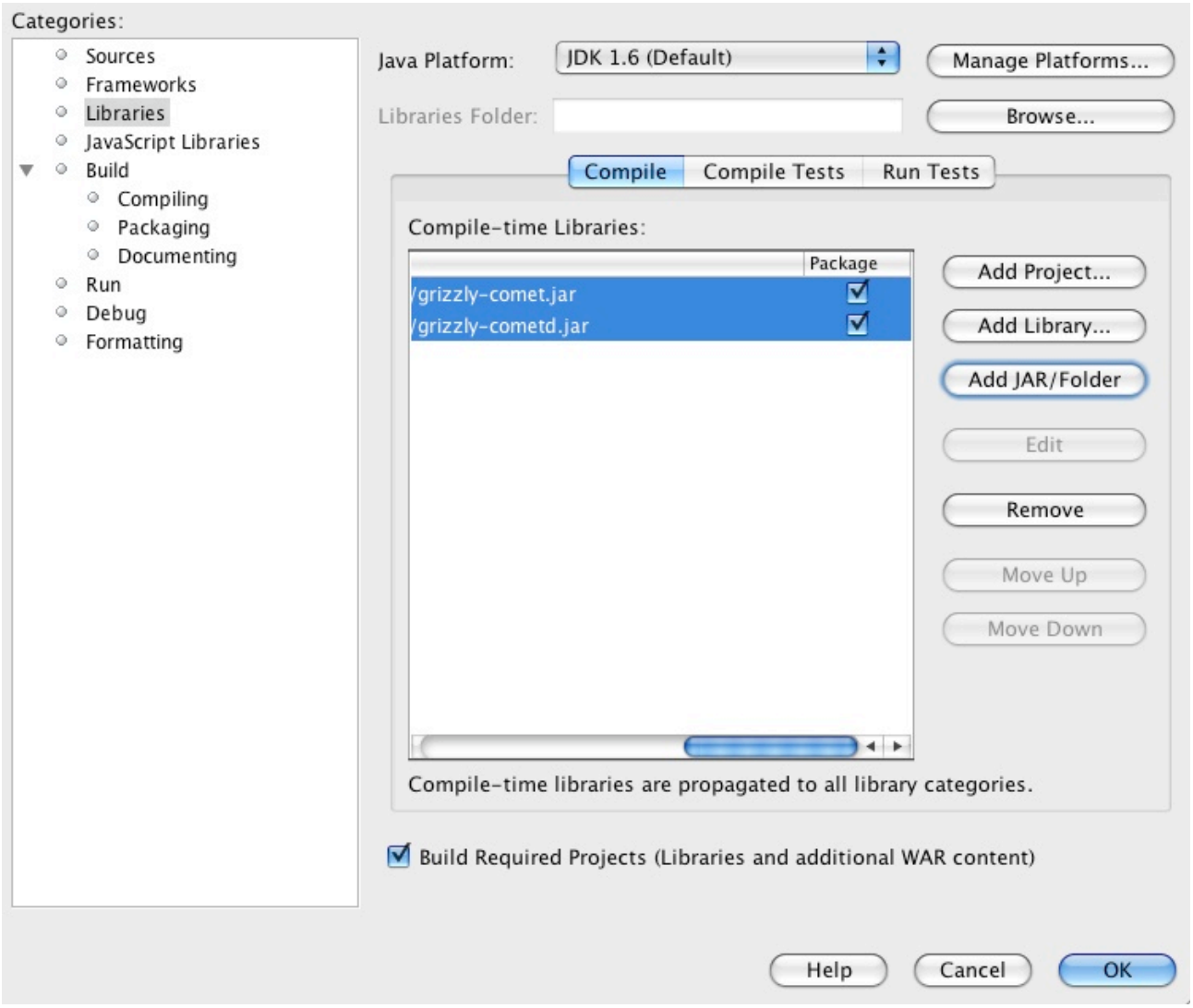
6. Copy all of the code below and paste it to replace the contents of the TTTComet.java file in the editor. This will be your web component that supports Comet requests.

```
package sample;


import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;


public class TTTComet extends HttpServlet {
       private String contextPath = null;
 private static TTTGame game = new TTTGame();

       @Override
       public void init(ServletConfig config) throws ServletException {
       }

       @Override
       protected void doGet(HttpServletRequest request,
       HttpServletResponse response)
       throws ServletException, IOException {
       }
```

```
        @Override
        protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        }
}
```

7. Now we need to add the Comet libraries to our project. Right click the `tictactoe` node and select Properties.
8. Find the project's Libraries in the left hand side of the window and select it.
9. Click the add JAR/Folder button on the right side. Select the following two JAR's from the `<lab_root>/exercises/exercise3` folder:
   - grizzly-comet.jar
   - grizzly-cometd.jar

   Click OK to finish Library selection.



10. Add the following imports to your TTTComet.java file.

```
import com.sun.grizzly.comet.CometContext;
import com.sun.grizzly.comet.CometEngine;
import com.sun.grizzly.comet.CometEvent;
import com.sun.grizzly.comet.CometHandler;
```

11. Now that we have our servlet class we need to add our Comet functionality. Start by copying and pasting the following code into the `init` method. This allows us to get the component's context path.

```
ServletContext context = config.getServletContext();
contextPath = context.getContextPath() + "/TTTComet";
```

12. Add this line to `init` get an instance of the Comet engine.

```
CometEngine engine = CometEngine.getEngine();
```

13. Add these lines to the `init` method. This registers the component with the Comet engine.

```
CometContext cometContext = engine.register(contextPath);
cometContext.setExpirationDelay(120 * 1000);
```

14. Next we need to create a private class that implements CometHandler in the servlet class. Copy and paste this code inside your TTTComet class. This is our Comet Handler that updates the client when events happen.

```
private class TTTHandler implements CometHandler<HttpServletResponse> {

        private HttpServletResponse response;

}
```

15. Now copy and paste the following methods into your new TTTHandler private class. We need to provide the implementations of all these methods since our class implements `CometHandler`. The `onInterrupt` and `onTerminate` methods execute when certain changes occur in the status of the underlying TCP communication. The `onInterrupt` method executes when communication is resumed. The `onTerminate` method executes when communication is closed. Both methods call `removeThisFromContext`, which removes the `CometHandler` object from the `CometContext` object.

```
public void onInitialize(CometEvent event) throws IOException {
}

public void onInterrupt(CometEvent event) throws IOException {
    removeThisFromContext();
}

public void onTerminate(CometEvent event) throws IOException {
    removeThisFromContext();
}

public void attach(HttpServletResponse attachment) {
    this.response = attachment;
}

private void removeThisFromContext() throws IOException {
    response.getWriter().close();
    CometContext context =
            CometEngine.getEngine().getCometContext(contextPath);
    context.removeCometHandler(this);
}
```

16. The final method we need in our TTTHandler class is the `onEvent` method. This method defines what happens when an event occurs. This method first checks if the event type is NOTIFY, which means that the web component is notifying the CometHandler object that a client has made a move. If the event type is NOTIFY, the `onEvent` method writes out JavaScript to the client with the updated board information. The JavaScript includes a call to the `chImg` function, which will update the board on the clients' pages.

    The last line resumes the Comet request and removes it from the list of active CometHandler objects. By this line, you can tell that this application uses the long-polling technique.

    Copy and paste this code into your TTTHandler class.

```
public void onEvent(CometEvent event) throws IOException {
    if (CometEvent.NOTIFY == event.getType()) {
        PrintWriter writer = response.getWriter();
        writer.write("<script type='text/javascript'>parent.chImg(" + game.getJSON() + ")</script>\n");
        writer.flush();
        event.getCometContext().resumeCometHandler(this);
    }
}
```

17. Now we need to get an instance of the TTT`Handler` and attach the response to it. We then get the Comet context and add the handler to it. Copy and paste the following lines into your `doGet` method.

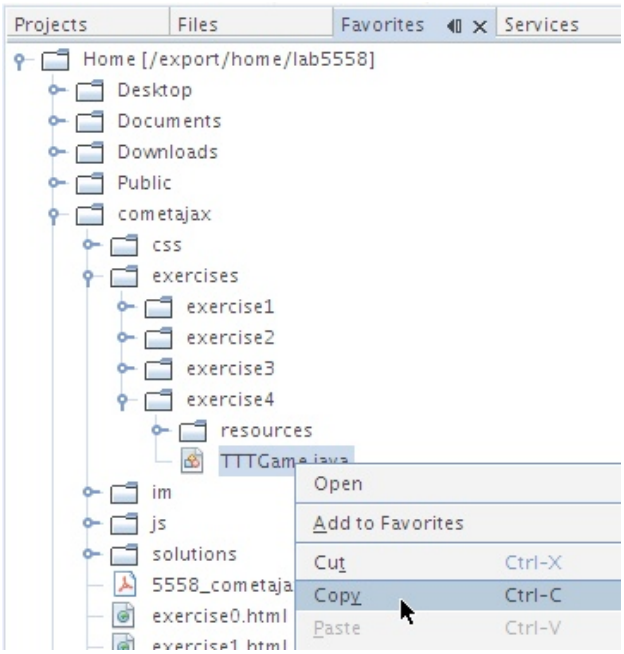
```
TTTHandler handler = new TTTHandler();
handler.attach(response);

CometEngine engine = CometEngine.getEngine();
CometContext context = engine.getCometContext(contextPath);

context.addCometHandler(handler);
```

18. When a user clicks the button, the `doPost` method is called. The doPost method checks that the player has made a valid move. It then obtains the current CometContext object and calls its notify method. By calling `context.notify`, the `doPost` method triggers the `onEvent` method you created in the step before last. After `onEvent` executes, `doPost` checks if there was a winner in the game. Copy and paste the following code into your `doPost` method.

```
int cell = -1;
String cellStr = request.getParameter("cell");
PrintWriter writer = response.getWriter();
writer.println("cell is '" + cellStr + "'");
if (cellStr == null) {
    writer.println("error - cell not set");
    return;
}
try {
    cell = Integer.parseInt(cellStr);
} catch (NumberFormatException nfe) {
    writer.println("error - cellStr not an int: " + cellStr);
    return;
}
if (!game.turn(cell)) {
    writer.println("warning - invalid move");
}
writer.println(game.getJSON());

CometEngine engine = CometEngine.getEngine();
CometContext context = engine.getCometContext(contextPath);
context.notify(null);

// Hideous hack, not threadsafe
if (game.win() != -1) {
    try {
    Thread.sleep(2000); // sleep 2 sec
    } catch (InterruptedException ie) {}
    game = new TTTGame();
}
```

19. The final thing that we ned to add to our server side web application is a new class for our game. Copy the TTT`Game.java` file from `<lab_root>/exercises/exercise4/` to your **tictactoe** project's **sample** package.

    The easiest way to do this is to open the Favorites window (Windows -> Favorites). This should display your home directory, if not, Right click and choose Add to Favorites and select the `<lab_root>`. Now you can navigate to the `<lab_root>/exercises/exercise4/` folder and right click the TTT`Game.java` file. Choose Copy and go back to the Projects window. Right click the **sample** package and choose Paste.



20. If you look back at our `onEvent` code you will see that we call a function in our game class called `getJSON`. This is the code that you will need to implement for this section of the exercise. Let's take a brief walkthrough of our `TTTGame` class.

    We start off with our class variables. The important thing to note here is the representation of our board. It is an array with one element for each position on the tic-tac-toe board. We also have an array with all the possible winning combinations represented.

    The first function that we have is called `turn` and checks whether a move is valid or not. It also checks whose turn it is.

    Next we have the `whoseTurn` function, which does exactly what it sounds like, checking whose turn it is.

    Our next important function is `win`. This function checks if there is a winner, no winner or a tie game.

    And finally we have the function that you need to implement, the `getJSON` function. This is called by onEvent which then sends this JSON representation of the game board to the clients. You need to implement a function that cycles through the elements in the board array and prints out a JSON representation of the board in the format described in the comments. Here it is again as an example as to how your JSON string should be outputted.

    Hint: You should use the win() function in your implementation. If you get stuck, check out the solutions folder for the answer.

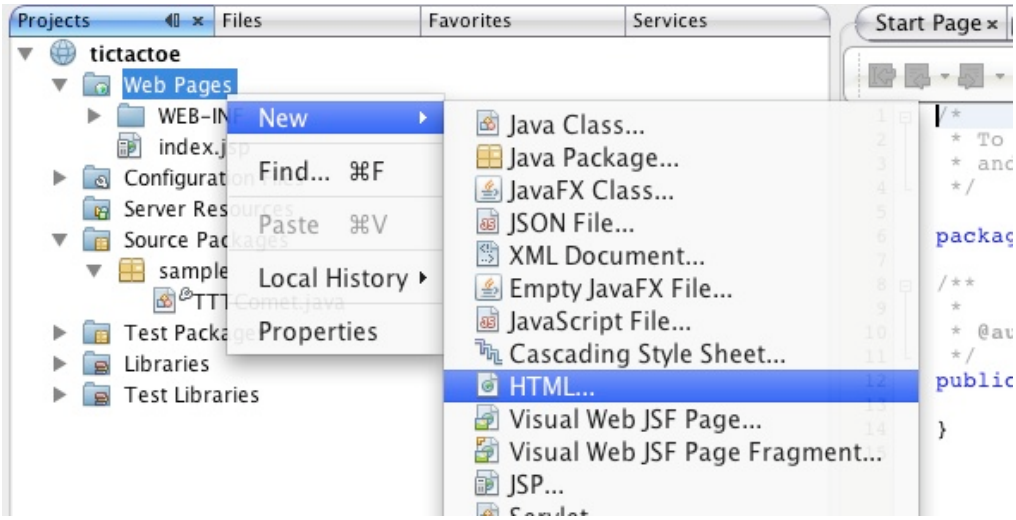
```
/**
 * Create a JSON representation of the game state.  It will look something like this:
 *  * { "win": "-1", "board": ["0","0","0","0","0","0","0","0","0"] } *
 * @return json
 */
synchronized public String getJSON() {

// CODE THIS FUNCTION

}
```

Step 3: Creating the client pages for the Tic-Tac-Toe game

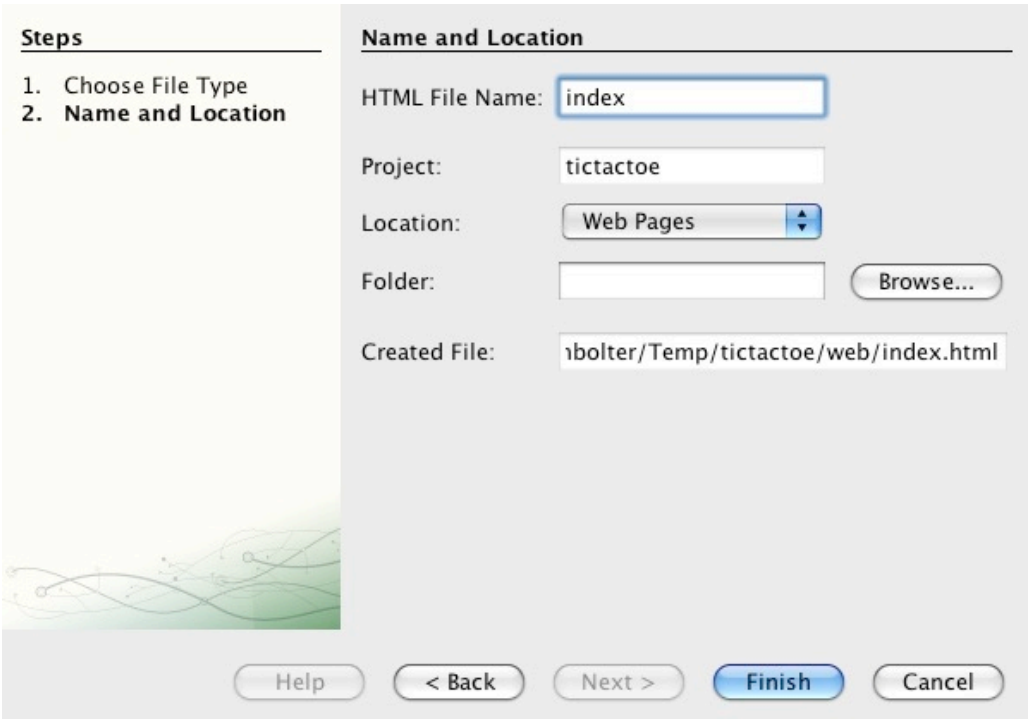
The goal of this part of Exercise 4 is to create the client pages that will represent our game. When a user clicks on a square a new POST request will be created and sent with information as to which cell was clicked. This will be caught and the `doPost` method on the server will run and update the game board and then notify all the clients of the new board.

1. The first thing that we need to do is create a new HTML page that will display our game board. In the Projects window right click the **Web Pages** node of the **tictactoe** project and select New HTML File...

   If you don't see New HTML File, select New -> Other. Change the Categories type to Other and select HTML File.



2. In the New HTML File Window, enter the name **index**, and click Finish.



3. Double click on the index.html file to open it in the editor.

4. Copy and paste the following code into your new index.html file between the opening <body> tag and the closing </body> tag.

```
<iframe name="hidden" src="TTTComet" frameborder="0" height="0" width="100%"
        onload="restartPoll()" onerror="restartPoll()" ></iframe>

<h1>Tic Tac Toe</h1>
<table>
    <tr>
        <td id="cell0"><img id="img0" src="resources/0.gif"
                            onclick=postMe("0")></td>
        <td id="cell1"><img id="img1" src="resources/0.gif"
                            onclick=postMe("1")></td>
        <td id="cell2"><img id="img2" src="resources/0.gif"
                            onclick=postMe("2")></td>
    </tr>
    <tr>
        <td id="cell3"><img id="img3" src="resources/0.gif"
                            onclick=postMe("3")></td>
        <td id="cell4"><img id="img4" src="resources/0.gif"
```

```
                            onclick=postMe("4")></td>
        <td id="cell5"><img id="img5" src="resources/0.gif"
                            onclick=postMe("5")></td>
    </tr>
    <tr>
        <td id="cell6"><img id="img6" src="resources/0.gif"
                            onclick=postMe("6")></td>
        <td id="cell7"><img id="img7" src="resources/0.gif"
                            onclick=postMe("7")></td>
        <td id="cell8"><img id="img8" src="resources/0.gif"
                            onclick=postMe("8")></td>
    </tr>
</table>
<h2 id="gstatus">Starting to watch the game.</h2>
```

This code contains a few important things. The first is the <iframe> tag. This IFrame contains the JavaScript code to make a connection to the servlet. When the page loads the connection is initialized and the restartPoll() function is run to put the client back into a waiting for more data mode. The next important thing to note is the HTML table. This is our tic-tac-toe board. Whenever a cell is clicked the postMe function is run with the associated cell number passed as an argument to the server.

5. Inside the opening <head> and closing </head> tags paste the following code.

```
<link rel="stylesheet" type="text/css" href="resources/tictactoe.css">
<script type="text/javascript">

</script>
```

6. Now let's add our JavaScript functions. Paste this code inside the <script></script> tags that you added in the last step.

```
var url = "TTTComet";
function postMe(arg) {
    function createXMLHttpRequest() {
        try { return new XMLHttpRequest(); } catch(e) {}
        try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
        try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
        alert("Sorry, you're not running a supported browser - XMLHttpRequest not supported");
        return null;
    };
    var xhReq = new createXMLHttpRequest();
    xhReq.open("POST", url, false);
    xhReq.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    xhReq.send("cell="+arg);
};
```

This function runs whenever a cell in the game board is clicked. It creates a POST request and sends it to the Comet servlet with the cell number that was clicked as an argument.

7. This is the next important JavaScript function. The chImg function updates the board with the JSON data that is received from the Comet servlet. It also alerts the clients of the status of the game and if there is a winner. Finally it sets the IFrame url to the servlet url which puts the client back into the waiting for more data mode. Copy and paste this code inside the <script></script> tags after the last JavaScript function we inserted.

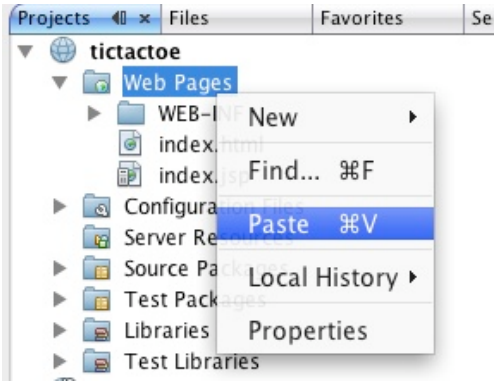
```
function chImg(args) {
    var data  = eval(args);

    for (i = 0; i < 9; i++) {
        // 0 is blank, 10 is x, 1 is o
        document.getElementById("img"+i).src="resources/"+data.board[i]+".gif";
    }

    var statusMsg;
    // -1 is unfinished, 0 is tie, 1 is X win, 2 is Y win
    if (data.win == 0) {
        statusMsg = "It's a tie!";
    } else if (data.win == 1) {
        statusMsg = "X wins!";
    } else if (data.win == 2) {
        statusMsg = "O wins!";
    } else if (data.win == -1 && data.turn == 10) {
```

```
        statusMsg = "It's X's Turn";
    } else if (data.win == -1 && data.turn == 1) {
        statusMsg = "It's O's Turn";
    } else {
        statusMsg = "That's odd, it shouldn't get here";
    }

    if (data.win != -1) {
        statusMsg = statusMsg + '<br><a href="index.html">Restart the game</a>';
    }

    // And write the status message out here -
    document.getElementById("gstatus").innerHTML = statusMsg;

    hidden.location = url;  // restart the poll
    retries = 0; // reset retry number
}
```

8. The final JavaScript that we need to make this game run is the `restartPoll` function. This runs when the IFrame loads and verifies that there is a connection to the servlet. Copy and paste this code inside the `<script></script>` tags after the last JavaScript function we inserted.

```
var retries = 0;
function restartPoll() {
    if (retries++ > 10) {  // stop the client from indefinate spin
        alert("The connection has errored out too many times - hit reload to retry");
    } else {
        hidden.location = url;    // restart the poll
    }
}
```

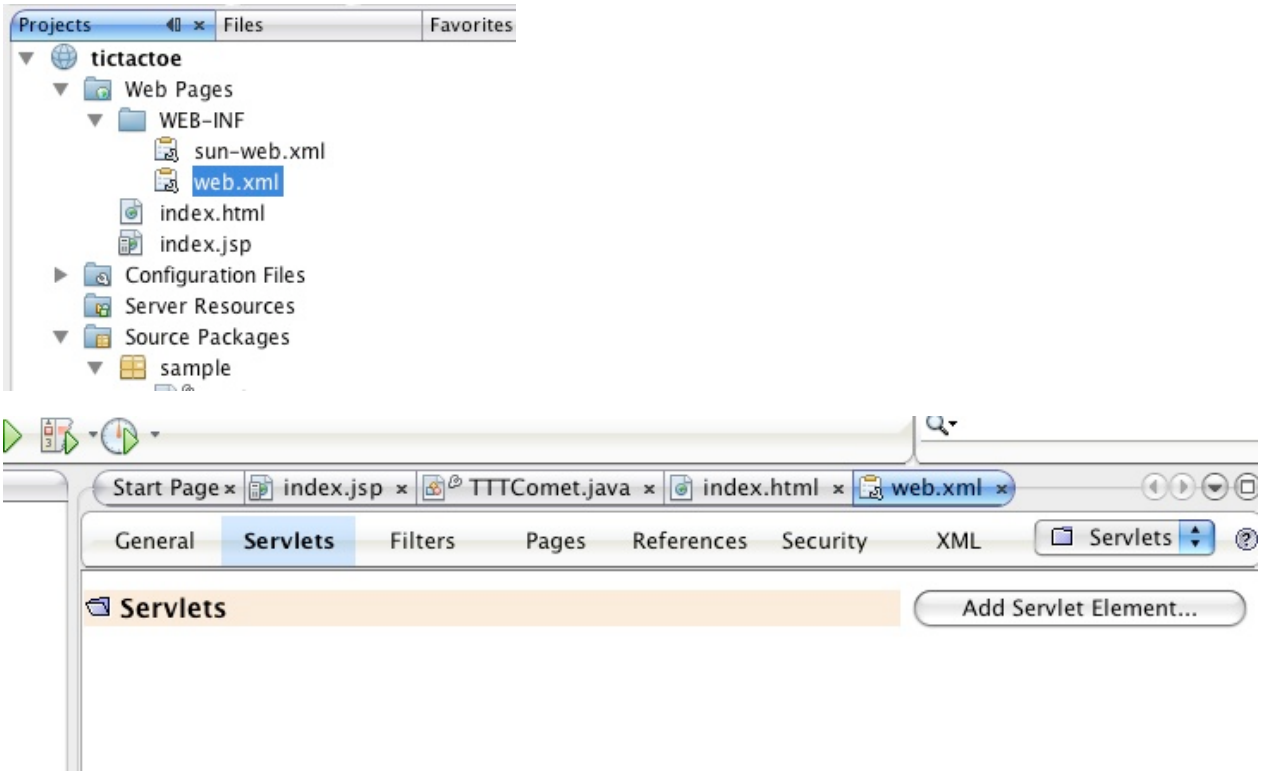
9. Finally we need to copy the images for our game into our project. Copy the `resources` folder from `<lab_root>/exercises/exercise4/` to your **tictactoe** project's Web Pages directory.



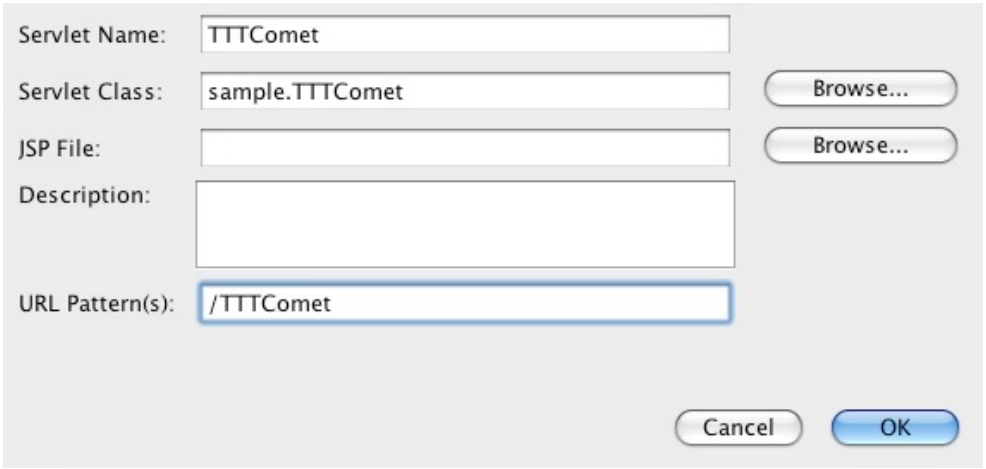
Step 4: Modifying the deployment descriptor.

Now we need to do is modify our deployment descriptor so that it knows about our servlet.

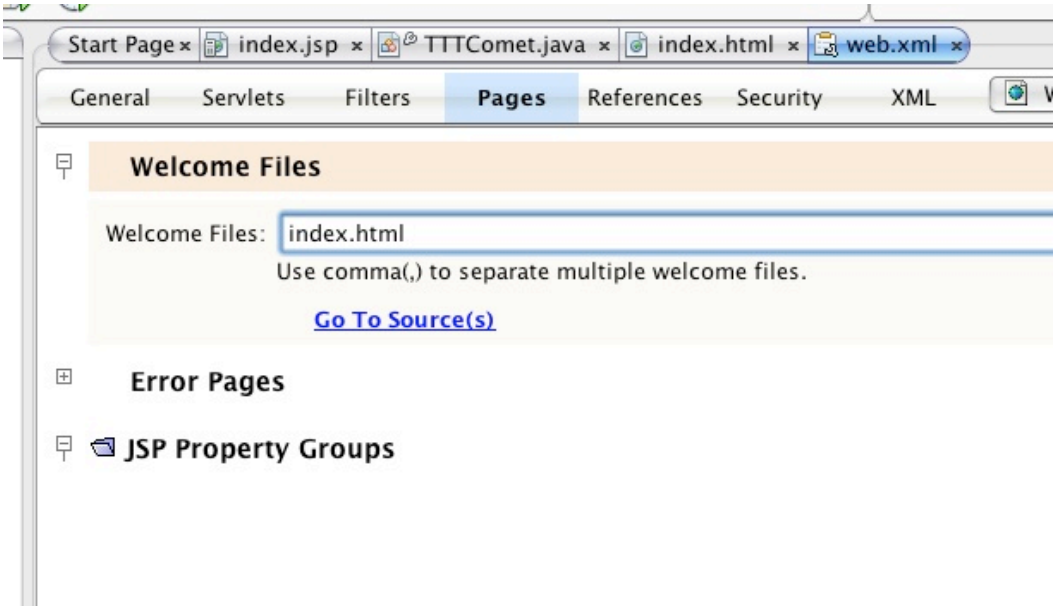
1. Open `web.xml` which you can find in your **tictactoe** project under `Web Pages -> WEB-INF` folder. Once you have the file open change to the Servlets view by clicking the Servlets button in Netbeans.



2. Click the Add Servlet Element button. Enter the following information for your servlet:
   - Servlet Name: **TTTComet**
   - Servlet Class: Click the Browse button and find your **sample.TTTComet** class.
   - JSP Class: *LEAVE BLANK*
   - Description: *LEAVE BLANK*
   - URL Pattern: **/TTTComet**



3. Now select the Pages button to move out of the Servlets section of the `web.xml` file.



4. Change the *Welcome Files* from index.jsp to index.html.
5. Save the changes you've made to your `web.xml` file.

Step 5: Enable Comet support in Glassfish (SKIP THIS STEP FOR JAVAONE 2009 MACHINE PROVIDED LAB).

See exercise 0 for detail on how to enable comet support in Glassfish if you haven't done so.

Step 6: Run the project.

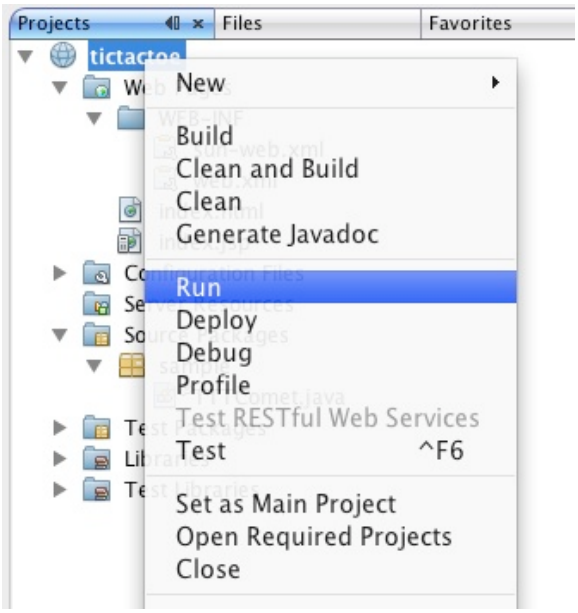
We have now built our Tic Tac Toe application. Go ahead and run the project and we will take a closer look at some of the code in a minute.

1. In the Projects window, right click your **tictactoe** project. Select Run. Wait for the Glassfish server to start. Once the browser window opens to your index.html page you can start playing the game. However, to see Comet in action, you need to open another browser and copy and paste the URL to open your tictactoe game (http://localhost:8080/TTTComet). This second browser can be another instance of Firefox. Make sure you've setup mutiple profiles as described in Exercise 0 and then when you start Firefox make sure to select your second profile. Note that if you are using OpenSolaris, to bring up the Profile Manager when Firefox is already running execute this command in a terminal and then select your second profile:

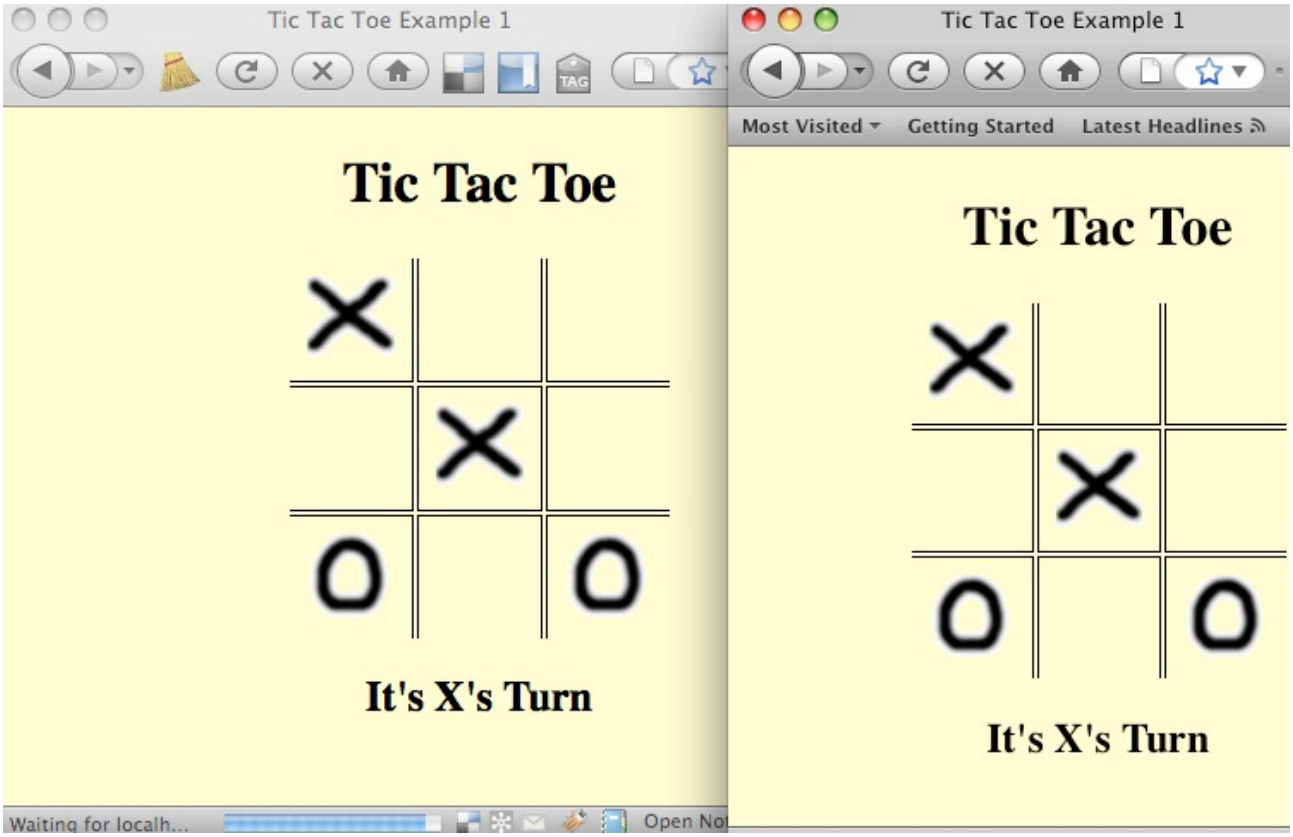
   ```
   firefox -P -no-remote
   ```

   FOR JAVAONE 2009 MACHINE PROVIDED LABS:
   You may type the shortcut `ff` into a terminal to start Firfox with the profile manager open.

   Now try clicking on a box and you can see each browser is updated as you click the different boxes. Once there is a winner each browser will be notified as well.



Here is what the game should look like when it is running:



Summary

The example application demonstrates the use of Comet's long-polling technique. You learned how to build the servlet that implements CometHandler to handle comet events. You also learned how the client side IFrame technique is used to interact with the Comet server to create an interactive two-player game.

See Also

For more information see the following resources:

- The Comet chat code used in the example application was adapted from an example originally written by Greg Wilkins.
- Cometd framework and its Bayeux protocol support in Grizzly – Jean-Francois Arcand's blog
- Using the Grizzly Comet API – Sun document on developing web applications
- Dojo Toolkit – Dojo home page
- dojo.cometd – Cometd.org page at the Dojo Foundation
- Grizzly – Project Grizzly home page
- GlassFish Application Server – home page for the GlassFish Community
- The Aquarium – news from the GlassFish Community
- Enterprise Comet: Awaken the Grizzly! – article
- Asynchronous HTTP and Comet Architectures – article
- A Comparison of Push vs Pull Ajax – article
- Comet and Reverse Ajax: The Next Generation Ajax 2.0 – book

Congratulations! You have successfully completed LAB-5558: Developing Real-Time Revolutionary Web Applications, Using Comet and Ajax Hands-on lab.

Where to send questions or feedbacks on this lab and public discussion forums:

- You can send technical questions via email to the authors of this Hands-on lab (and experts on the subject) or you can post the questions to the web.
  Please post questions that are relevant only to this hands-on lab.
  - 5558_cometajx at JavaONE_HOL_Forum

- You can send your other questions to the public users alias:
  - users@grizzly.dev.java.net