

## Introduction


In December 2007, BlazeDS was initially released in beta, making it the first freely available product to support AMF3 and RTMP. These two technologies, available to all developers with access to a Tomcat server, open up the doors to not only new ways of collaborating, but also exciting new ways of interacting with data. Like many, I rushed to download the release, and the possibilities presented in the TestDrive excited me. Then I searched in vain for documentation describing how to create my own Java-powered backend.

I intend to bridge the knowledge gap between Flex developer and Java developer. As it stands, the only resources for learning Java fall into two categories: those that expend many chapters explaining basic concepts, and those that assume five years of Java experience. Neither of these help the Flex developer (who is) endeavoring to learn basic Java in order to create advanced websites. Novice material does not cover deploying to Tomcat, nor do they explain POJOs in Tomcat references/resources/tutorials. Hopefully this text can help bridge that gap.

For the purposes of this paper/article, I'm going to assume that you are relatively proficient with Flex/AS3. While Flex proficiency is not required to complete this exercise, I will be skipping many explanations on the Flex side of things as I explain how to create a very simple application. I will be skipping database connectivity, but there will be a follow-up that explains how to use Hibernate with this setup to provide quick and easy data access.



Let's start it up: navigate to the bin directory and double-click startup.bat. We'll need to load the BlazeDS application into this new installation of Tomcat, so copy blazeds.war and ds-console.war from the BlazeDS directory into the Tomcat6\webapps directory. Tomcat will automatically deploy the applications. Now double-check this in the Manager application by visiting our server at <http://localhost:8080/>, clicking on 'Tomcat Manager' on the left, and using the username/password: admin/admin. You'll see blazeds and ds-console listed.

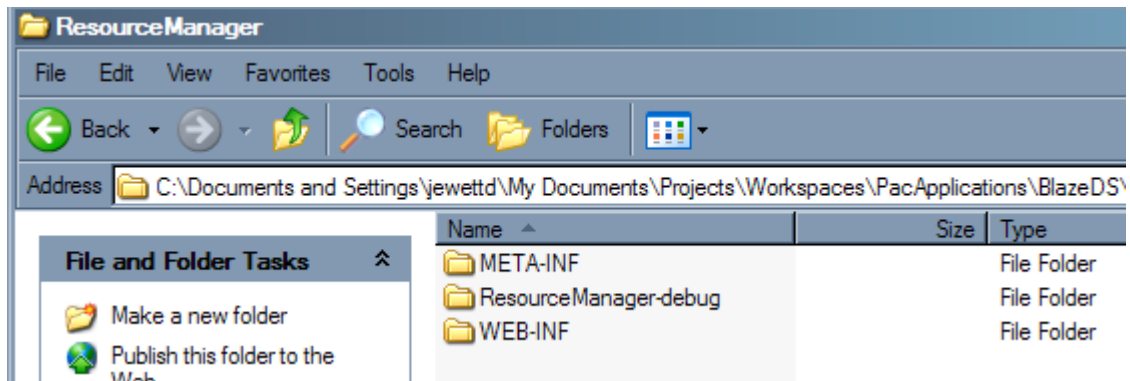


The Apache Software Foundation logo is shown at the top left, and the Tomcat Web Application Manager interface is shown at the top right. The interface includes a navigation menu with 'List Applications', 'HTML Manager Help', 'Manager Help', and 'Server Status'. Below the menu is a table of applications.

Path	Display Name	Running	Sessions	Commands
/	Welcome to Tomcat	true	0	Start Stop Reload Undeploy Expire sessions with idle > 30 minutes
/blazeds	BlazeDS	true	0	Start Stop Reload Undeploy Expire sessions with idle > 30 minutes
/docs	Tomcat Documentation	true	0	Start Stop Reload Undeploy Expire sessions with idle > 30 minutes
/ds-console	Console	true	0	Start Stop Reload Undeploy Expire sessions with idle > 30 minutes
/examples	Servlet and JSP Examples	true	0	Start Stop Reload Undeploy Expire sessions with idle > 30 minutes
/host-manager	Tomcat Manager Application	true	0	Start Stop Reload Undeploy Expire sessions with idle > 30 minutes
/manager	Tomcat Manager Application	true	0	Start Stop Reload Undeploy Expire sessions with idle > 30 minutes

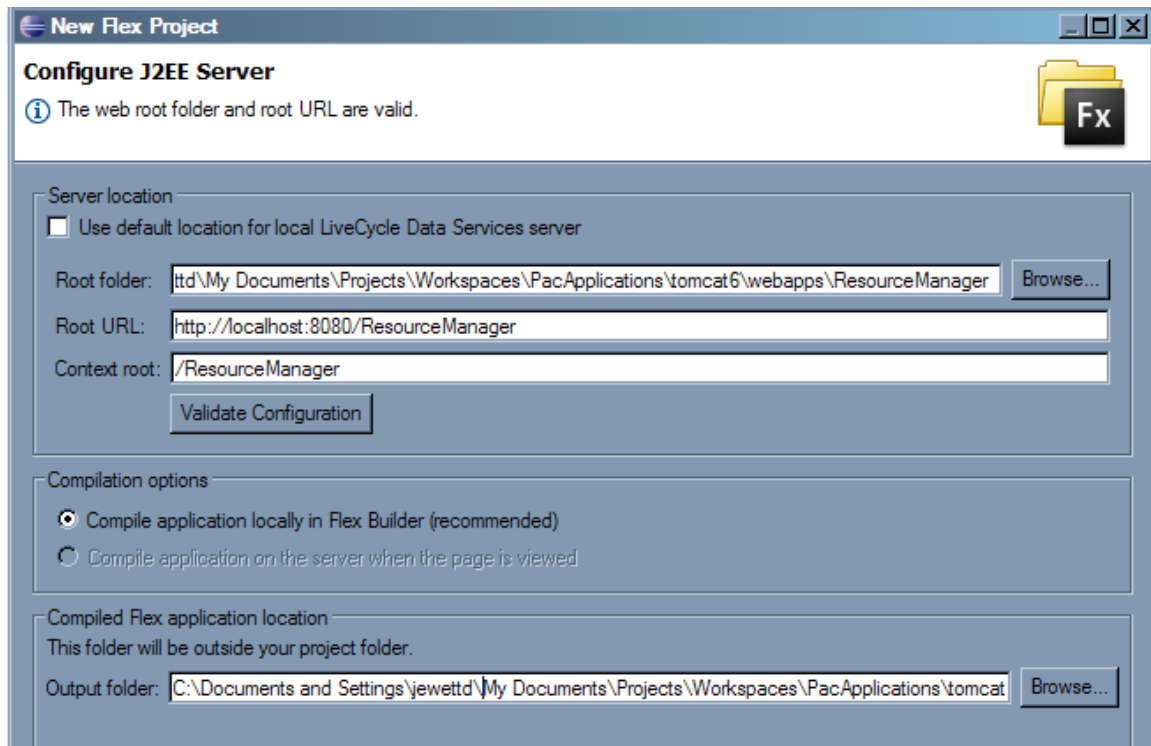
## Filesystem

Because FlexBuilder requires existing files on the server before it can setup a Remoting project, we have to copy some of the sample files. To do this, find your BlazeDS/tomcat/webapps directory, and copy the blazeds folder to a new folder (I call mine ResourceManager, which probably isn't such an awesome idea, as there is already a ResourceManager flex component). This folder is basically a template with the required files and folders included. We'll look closer at these folders later, but these folders contain enough information to let FlexBuilder create a Remoting project. (Note for image below: your folder won't have the ResourceManager-debug folder)



## Eclipse

Now that BlazeDS is running and our webapp folder is set up, we can start our project in Eclipse. WebORB provides good instructions with exactly how to setup FlexBuilder [here](#). When you get to the J2EE Server configuration, your setup will look

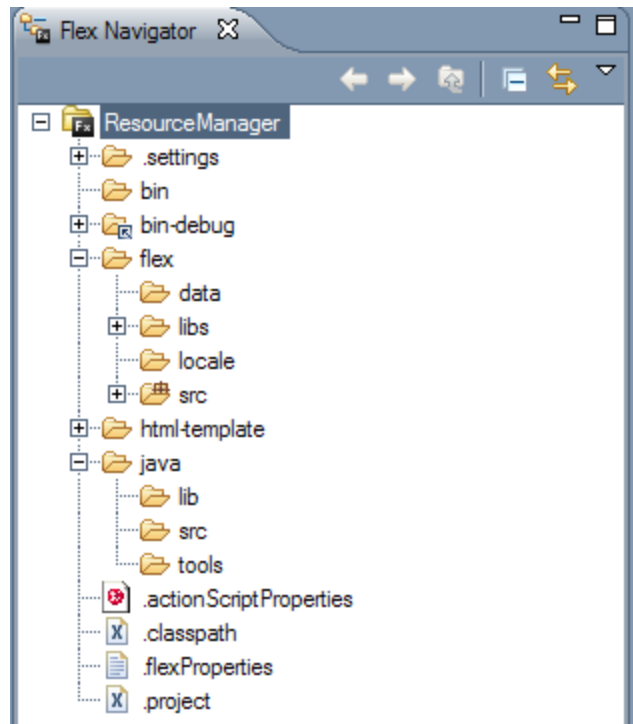


something like:

We'll set up the Flex Project a bit differently, since we will be storing both Java and Flex code in the same project. Set your main source folder to be `flex/src`, then press "Finish". We now have to edit our `.project` file to allow us to use JDT commands. Daniel Harfleet has already written instructions for that, so follow the directions [here](#). Mac users will have a hard time finding `.project`, use the command found [here](#) to show the file. Be sure to close then reopen your project after you make these changes.

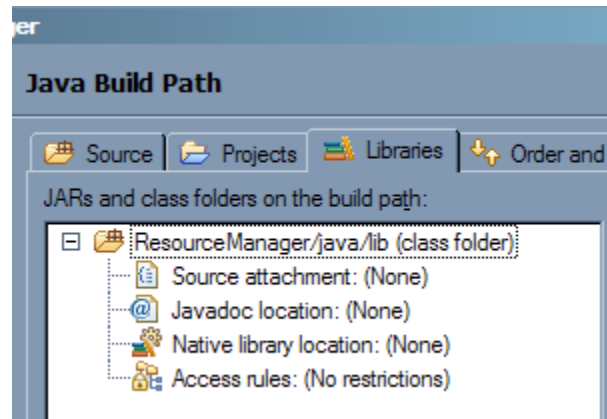
```
<buildSpec>
  <buildCommand>
    <name>com.adobe.flexbuilder.project.flexbuilder</name>
    <arguments>
    </arguments>
  </buildCommand>
  <buildCommand>
    <name>org.eclipse.jdt.core.javabuilder</name>
    <arguments>
    </arguments>
  </buildCommand>
</buildSpec>
<natures>
  <nature>com.adobe.flexbuilder.project.flexnature</nature>
  <nature>com.adobe.flexbuilder.project.actionscriptnature</nature>
  <nature>org.eclipse.jdt.core.javanature</nature>
</natures>
```

Alright, we've got a project. Let's make some folders in our main project folder:



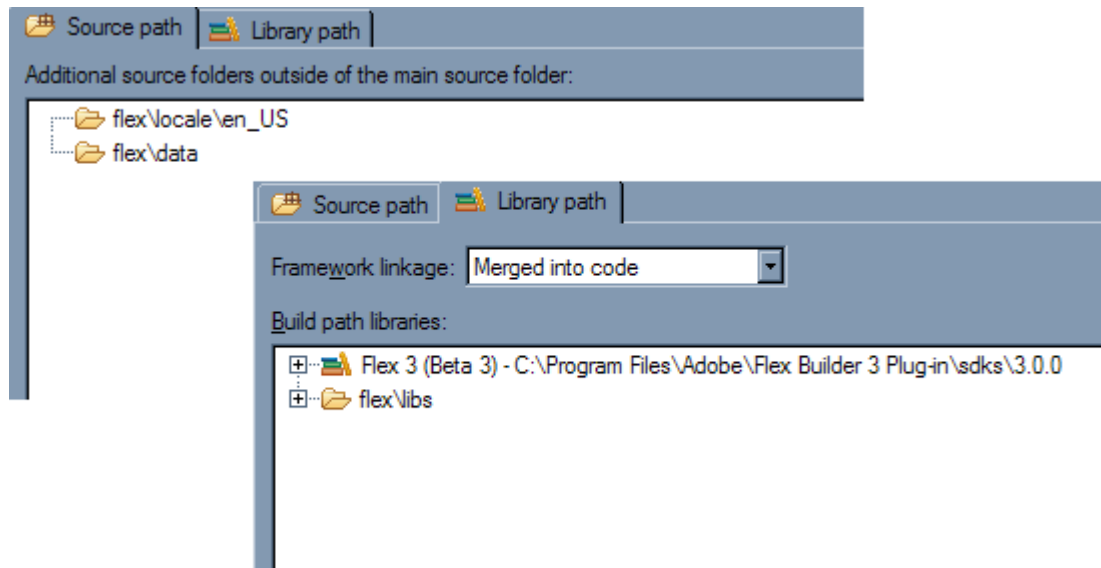
## Eclipse Java

Now we have to setup the Java side of the project. For that we'll switch the perspective to Java. When you switch, Eclipse will automatically search for probable source folders. It'll be wrong, so you'll have to fix it. Right click on your project, click the "Java Build Path" section. We're going to remove the default build path, and point to a new one: `ResourceManager/java/src`.



We're almost done setting up, but we need our ILOG components. We'll download their package from their website: <http://www.ilog.com/products/elixir/> and install. Find the sample source files here: `C:\Program Files\ILOG\ILOG Elixir 1.0 Beta 2\samples\humanresources` and copy the folders into the `ResourceManager/flex/` folder. There are two swc's that you'll have to copy into the `flex/libs` folder. One is in `ILOG Elixir 1.0 Beta 2\frameworks\libs`,

while the other is in `ILOG Elixir 1.0 Beta 2\frameworks\locale\en_US`. A couple of changes to the project build path, switch to Flex perspective, and we'll be able to compile!



You should be able to run your program, and it will pop up a browser pointing to your Tomcat server at port 8400. If not, visit the problems pane and figure those out. (If you named your project ResourceManager, you'll have to rename or delete ResourceManager.mxml)

## Flex Development

The ideal way to connect BlazeDS and our charting is by using POJOs, Plain Old Java Objects. ~~Well, the way we deal with Java Remoting is using POJOs, Plain Old Java Objects.~~ It took me a while to get the hang of how simple it is, thanks to the Java world assuming that **everyone** knows what a POJO is, but I found that it's one of the easiest things to do in the data-transfer world.

Before we write any POJOs, let's look at the Flex side of our application. On line 128 in `humanresources.mxml`, we see some generic objects being created from XML. These two objects (person and absence) are really simple, so let's create a separate type for each of them:

```
package vo
{
    public class HRPerson
    {
        public var id:String;
```

```

public var firstname:String;
public var name:String;
public var location:String;
public var genre:String;
}
}

```

```

package vo
{
public class HRAbsence
{
public var resourceId:String
public var reason:String
public var startTime:String
public var endTime:String
}
}

```

Replace the generic object assignments with your shiny new objects:

```

var persons:Array = [];
for each (var person:XML in data.person) {
var p:HRPerson = new HRPerson();
p.id = person.@id.toString();
p.firstname = person.@firstname.toString();
p.name = person.@name.toString();
p.location = person.@location.toString();
p.genre = person.@genre.toString();
persons.push(p);
}
//Read the absences
var absences:Array = [];
for each (var absence:XML in data.absence) {
var a:HRAbsence = new HRAbsence();
a.resourceId = absence.@resourceId.toString();
a.reason = absence.@reason.toString();
a.startTime = absence.@startTime.toString();
a.endTime = absence.@endTime.toString();
absences.push(a);
}

```

Compile, and make sure your code still runs. Until we get our backend running, we're done in Flex.

## Java Configuration

It's time to get back to those `META-INF` and `WEB-INF` folders. To be honest, I have no clue what `META-INF` does, but it's included in the template, so we'll leave it alone. `WEB-INF`, on the other hand, is pretty much the heart of your application. First off, we have to get these files into our project, so make a directory called `war` in the root of your project folder (note: `war` stand for Web ARchive and it allows developers to distribute their application as one file), then copy the `META-INF` and `WEB-INF` folder into it.

The first file we'll look at is `WEB-INF/web.xml`. This file controls how the application gets loaded into Tomcat, pointing at the necessary classes to listen for requests and points to the `services-config.xml` file. It also controls the application name and description, which you should change right now.

Following the file trail, we'll load up `WEB-INF/flex/services-config.xml` (as found on line 20 of `web.xml`). This file declares all of the possible channels, the logging functions, and a list of files that trigger redeployment when changed.

We'll keep exploring files as we find them, so next up is `WEB-INF/flex/remoting-config.xml`. This file controls all of the Remoting aspects, setting up destinations and pointing those destinations at specific class files. We're going to add a destination on line 12, right after `</default-channels>`

```
<destination id="hrManagement">
  <properties>
    <source>com.hat6.hrManagement.controller.HRService</source>
  </properties>
</destination>
```

This will tell BlazeDS to use our soon-to-be-written `HRService` class to respond to any requests labeled as "hrManagement". Pretty self-explanatory.

The next file in our list is `WEB-INF/flex/proxy-config.xml`. We won't be using proxies in this demo, but this will allow you to make cross-domain requests, even if there isn't a `crossdomain.xml` file on the host.

And finally, let's take a look at messaging-config.xml. We'll be using messaging to publish updates to our application after it's already running in the browser. Messaging allows real-time communication between the server and the swf. Add in a new destination on line 13, right after </default-channels>:

```
<destination id="updates">
  <channels>
    <channel ref="my-streaming-amf" />
    <channel ref="my-polling-amf" />
  </channels>
</destination>
```

The two channel references tell Flex that if you can't get a streaming connection, try a polling connection. You can use any of the channels setup in services-config.xml, and there is a lot more information out on the webs about the differences between channels.

We've setup our configuration files, so let's take a look at the rest of the files and folders in WEB-INF.

- WEB-INF/classes contains all of the java classes that we will be writing. Inside you'll see a commons-logging.properties, which sets up the logging for our app. Move this to your java/src folder.
- WEB-INF/flex folder contains the config files, but also a folder for hotfixes and for logs.
- WEB-INF/lib contains the java libraries needed for your java application. We can delete the HSQLDB.jar file, as we know we won't be using that, but leave the rest.

This war folder will end up being the base of the folders that we copy over into BlazeDS when we run our application. Our ant script will take care of all of the dirty work for us, so as long as we aren't coding, let's take a look at the build.xml file we will be using.

## Ant Configuration

Ant is a great tool for automating build processes, like compiling and copying and archiving, etc. You should really have a good grasp on how ant works before continuing, so read up on it over [here](#). The build.xml file was pilfered from [Sam](#) over at [ziazoo.co.uk](#), and slightly modified to fit our needs.

So, a quick scan through the file, with some brief explanations:

```
<project name="Resource Manager Master" basedir="." default="usage">
  <property file="build.properties" />

  <property name="src.dir" value="java/src" />
  <property name="web.dir" value="war" />
  <property name="flex.src" value="flex/src" />
  <property name="flex.dir" value="flex/bin" />
  <property name="flex.home"
    value="C:/Program Files/Adobe/Flex Builder 3 Plug-in/sdks/3.0.0/" />
  <property name="FLEX_HOME" value="${flex.home}" />
  <property name="build.dir" value="${web.dir}/WEB-INF/classes" />
  <property name="name" value="ResourceManager" />
  <property name="test.dir" value="test" />
```

Here we setup our properties, pointing to different folders. You'll want to double-check that flex.home is setup properly for your system. Pretty self explanatory. This code references a build.properties file, which contains this:

```
# Ant properties for building the ResourceManager

appserver.home=C:/Documents and Settings/jewettd/My
Documents/Projects/Workspaces/PacApplications/tomcat6
# for Tomcat 5 use ${appserver.home}/server/lib
# for Tomcat 6 use ${appserver.home}/lib
appserver.lib=${appserver.home}/lib
deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=admin
tomcat.manager.password=admin
```

The build.properties file helps when dealing with projects with multiple developers. The file lets each developer set properties specific to their system. The last three lines allow the ant script to control a full Tomcat server.

Let's continue on through the build.xml file:

```
<path id="master-classpath">
  <fileset dir="{web.dir}/WEB-INF/lib">
    <include name="*.jar" />
  </fileset>
  <!-- We need the servlet API classes: -->
  <!-- * for Tomcat 5/6 use servlet-api.jar -->
  <!-- * for other app servers - check the docs -->
  <fileset dir="{appserver.lib}">
    <include name="servlet*.jar" />
  </fileset>
  <pathelement path="{build.dir}" />
</path>
```

This section scans the lib directory in the war folder and in the server's home directory, loading up libraries for the compile.

```
<target name="build" description="Compile main source tree java files">
  <mkdir dir="{build.dir}" />
  <javac destdir="{build.dir}" source="1.6" target="1.6"
    debug="true" deprecation="false" optimize="false"
    failonerror="true">
    <src path="{src.dir}" />
    <classpath refid="master-classpath" />
  </javac>
  <copy todir="{build.dir}" preservelastmodified="true">
    <fileset dir="{src.dir}">
      <include name="*.hbm.xml" />
      <include name="*.properties" />
    </fileset>
  </copy>
</target>
```

Here's where the java magic happens. First we make sure that our build folder exists, then we compile, then we copy some necessary files over into the build folder. This will copy our commons-logging.properties folder over, and when we upgrade to using Hibernate, our hibernate mapping files.

The mxmllc portion of our ant file is also different than a regular mxmllc ant command. We've included services and context-root directives, along with some compiler.source-path modifiers. The context-root directive is the root directory, hence \${name}. The source-path modifiers add the locale and data folders to the path, like we did previously in eclipse.

```
<mxmllc file="{flex.src}/humanresources.mxml"
  output="{flex.dir}/ResourceManagerJava.swf"
```

```

services="war\WEB-INF\flex\services-config.xml"
context-root="{name}"
incremental="true"
debug="true"
keep-generated-actionscript="false">
<load-config filename="{flex.home}/frameworks/flex-config.xml"/>
<compiler.source-path path-element="{flex.home}/frameworks"/>
<compiler.source-path path-element="flex/locale/en_US"/>
<compiler.source-path path-element="flex/data"/>
<compiler.library-path dir="flex" append="true">
  <include name="libs" />
</compiler.library-path>
</mxmxc>

```

To be sure, there are many more interesting sections in the build file, but you should be able to figure out what they do easily enough. We're on page 13 and haven't written any Java code yet, so let's be finished with our configuration and start some development!

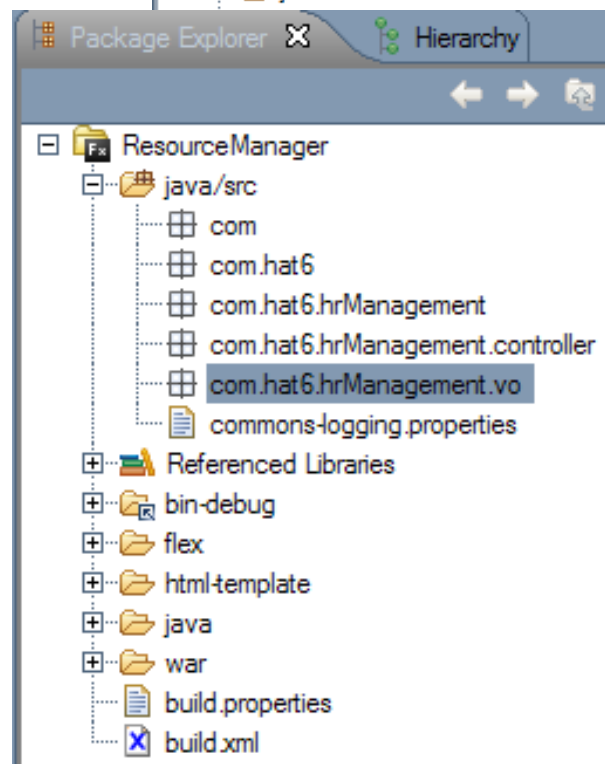
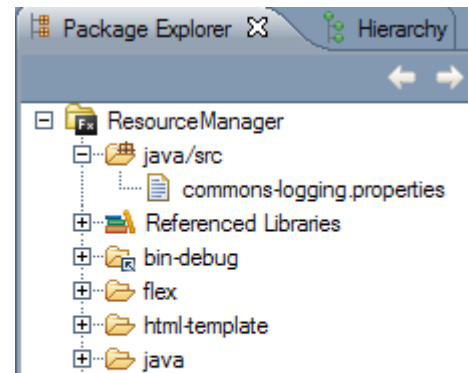
## Java Programming

While FlexBuilder is based on the JDT, it'll still take a bit of getting used to. The first major difference that you'll notice is that when viewing the Java perspective, the java/src folder is listed by itself, rather than as a child of the java folder.

JDT displays source folders separate from the rest of the folders, so don't be alarmed when you look in the java folder and can't find src. When you switch back to the Flex perspective, the source folder will be right where you expect it. If you remember when we edited our configuration files, BlazeDS is going to look for our service at:

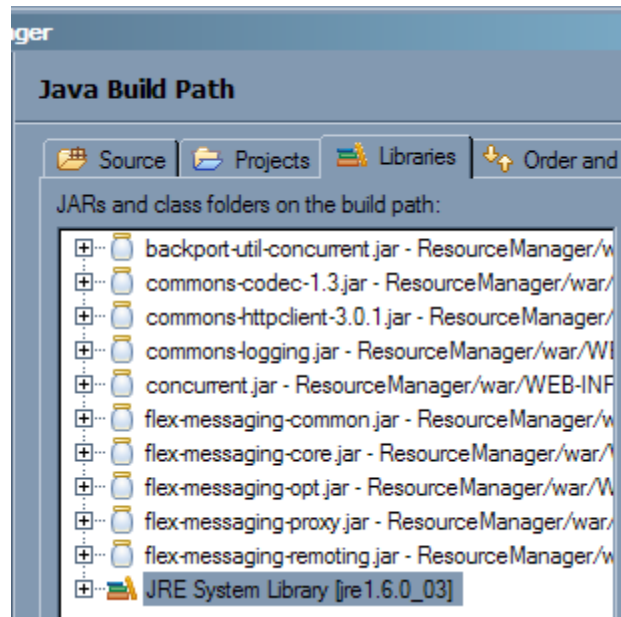
```
com.hat6.hrManagement.controller.HRService.
```

We'll create the required namespace by right clicking on the java/src folder, and



adding new folders. In this screenshot, I've created a folder for vo, to hold our POJOs, as well.

Right click on the controller namespace and select New -> Class. Type in HRService as the name, and select “Constructors from superclass” before pressing Finish. HRService.java will open up in the editor, but you'll see that there are some problems. The first problem tells us to fix the build path, so go to the Project Properties and look at the Java Build Path, Libraries tab. We need to add in the main Java libraries, and add in the libraries in our war/WEB-INF/lib directory. First, select “Add Library”, then select “JRE System Library”, then select “Workspace Default” and press Finish. Then select “Add JARs”, navigate to your war/WEB-INF/lib directory, and select all of the JAR files there. You'll end up with this: (may be slightly different libraries listed)



Press “OK” and Eclipse will rebuild, and tada, no more problems!

If you run ant deploywar, your application will be compiled and compressed into a .war file, then copied into the webapps directory, where it will be automatically deployed. Your tomcat window will show you any status messages, and will most likely give you an INFO warning about HttpFlexSession. This means that your application was loaded, and you can view it in the Tomcat Management console.

Our Java service is loaded into BlazeDS, so we'll make a couple of POJOs that mirror our Flex objects and have a handy constructor.

```
package com.hat6.hrManagement.vo;

public class HRPerson {

    public String id;
    public String firstname;
}
```

```

public String name;
public String location;
public String genre;

public HRPerson(String id, String firstname, String name, String location, String genre){
    this.id = id;
    this.firstname = firstname;
    this.name = name;
    this.location = location;
    this.genre = genre;
}
}

package com.hat6.hrManagement.vo;

public class HRAbsence {

    public String resourceId;
    public String reason;
    public String startTime;
    public String endTime;

    public HRAbsence(String resourceId, String reason, String startTime, String endTime){
        this.resourceId = resourceId;
        this.reason = reason;
        this.startTime = startTime;
        this.endTime = endTime;
    }
}

```

As you can tell, there are only minor differences in syntax between Java and Flex, so porting these VOs across is pretty easy.

Let's update our Service to return a list of people and absences:

```

package com.hat6.hrManagement.controller;

import com.hat6.hrManagement.vo.*;
import java.util.*;

public class HRService {

    public List<HRPerson> getHRPersonList()
    {
        List<HRPerson> theList = new ArrayList<HRPerson>();
        theList.add(new HRPerson("1", "Dusty", "Jewett", "Seattle", "male"));
        theList.add(new HRPerson("2", "Ali", "Daniali", "Seattle", "male"));
        theList.add(new HRPerson("3", "Marty", "Michelson", "Seattle", "male"));
        return theList;
    }

    public List<HRAbsence> getHRAbsenceList()
    {
        List<HRAbsence> theList = new ArrayList<HRAbsence>();
        theList.add(new HRAbsence("1", "Travel", "2007/09/21", "2007/09/25"));
        theList.add(new HRAbsence("1", "Travel", "2007/08/21", "2007/08/25"));
        theList.add(new HRAbsence("2", "Travel", "2007/08/25", "2007/08/29"));
        theList.add(new HRAbsence("2", "Travel", "2007/09/25", "2007/09/29"));
        theList.add(new HRAbsence("3", "Travel", "2007/08/01", "2007/08/07"));
        theList.add(new HRAbsence("3", "Travel", "2007/09/05", "2007/09/15"));
        return theList;
    }
}

```

```
}
```

For our first look at functional Java code, we'll explain some things. We'll be returning a List object back to Flex, and Lists must be typed (and can only hold items of that type). You can't create an actual List type, instead you create a subclass, which determines how the list acts internally. We'll use ArrayList for a simple implementation. Then we add new items, then return the list. The syntax is very easy to get used to if you already know AS3.

Now that we have our Java ready, we need to modify our humanresources.mxml to connect up to the service. We'll be replacing the init function, and adding some new functions:

```
private function init():void
{
    //loadDataModel();
    dateFormatter.formatString = getString("absence.tip.date.format");
    flash.net.registerClassAlias("com.hat6.hrManagement.vo.HRPerson", HRPerson);
    flash.net.registerClassAlias("com.hat6.hrManagement.vo.HRAbsence", HRAbsence);
    initializeRemoteObjects();
}
private var hrService:RemoteObject;
public function initializeRemoteObjects():void{
    trace("Connecting to Service");
    hrService = new RemoteObject("hrManagement");
    hrService.source = "com.hat6.hrManagement.controller.HRService";
    hrService.addEventListener(FaultEvent.FAULT, handleROError);
    hrService.getHRPersonList.addEventListener(ResultEvent.RESULT, handleGetHRPersonList);
    hrService.getHRAbsenceList.addEventListener(ResultEvent.RESULT, handleGetHRAbsenceList);
    hrService.getHRPersonList();
}
private function handleROError(result:FaultEvent):void{
    Alert.show("Connection Error");
}
private function handleGetHRPersonList(result:ResultEvent):void{
    resourceChart.resourceDataProvider = result.result as ArrayCollection;
    hrService.getHRAbsenceList();
}
private function handleGetHRAbsenceList(result:ResultEvent):void{
    resourceChart.taskDataProvider = result.result as ArrayCollection;
    configureSample();
}
}
```

Deploy the war file, wait a bit for Tomcat to deploy, then look at your application, you'll see the results:



# Resource Chart Sample



First Name	Last Name	Q3, 2007	
		August	September
Seattle			
Dusty	Jewett		
Ali	Daniali		
Marty	Michelson		

## Edit Absence

- New Absence
- Delete Absence

## Properties

### Employee:

### Reason:

- Travel
- Mission
- Vacation
- Sickness

Data Visualization Elixir Trial

I'm guessing it didn't turn out like that the first time around.