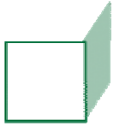
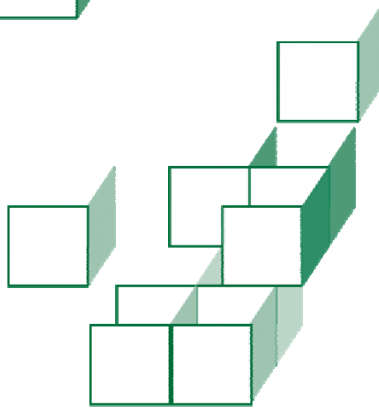




CHAPTER 4

RMI FRAMEWORKS



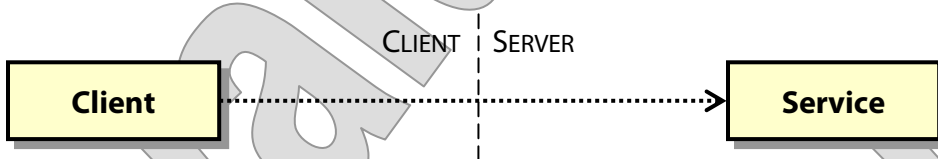
OBJECTIVES

After completing “RMI Frameworks,” you will be able to:

- Explain the advantages of an RMI approach to Ajax.
- Use two popular RMI frameworks to power Ajax applications:
 - Direct Web Remoting, or DWR
 - jabsorb
- Understand and address various issues with RMI frameworks:
 - Controlling serialization of objects and object graphs
 - Designing the Java APIs to make best use of remote vs. serializable designations for client-accessible objects
 - Being aware of RMI as an attack vector for web hackers, and applying appropriate security measures

Back to the Future II

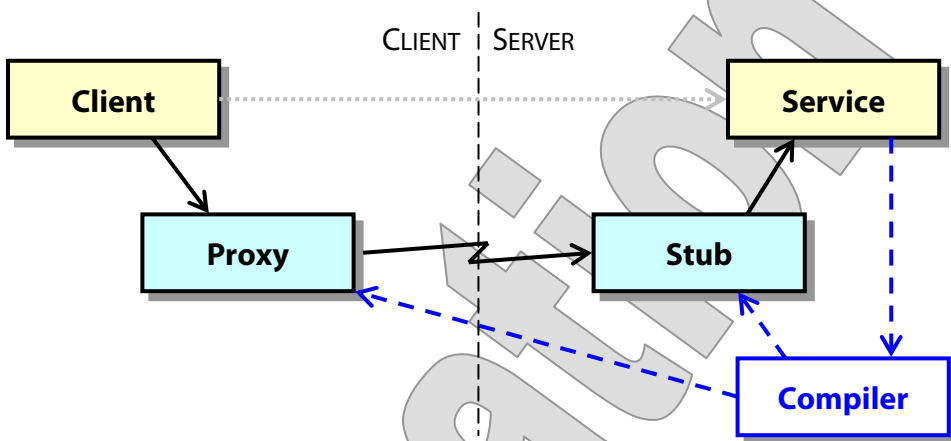
- Ever since some wizened monk thought to combine the concepts of object-oriented software and remote procedure calls, there have been a long string of remote method invocation frameworks – just a sampling follows:
 - Microsoft’s **DCOM**
 - **CORBA**
 - **Java RMI** and **EJB**
 - **JAX-WS** and other WSDL-driven web-service frameworks
- The appeal of RMI is that it exposes the object-oriented interface of a software component to remote clients – again, OO + RPC.



- That feature really resonates with the JavaScript programmer who’s trying to make things happen on the server side via Ajax.
 - If we’re going to be triggering method invocations on server-side objects, why not make that process as transparent as possible?

Java Objects in JavaScript?

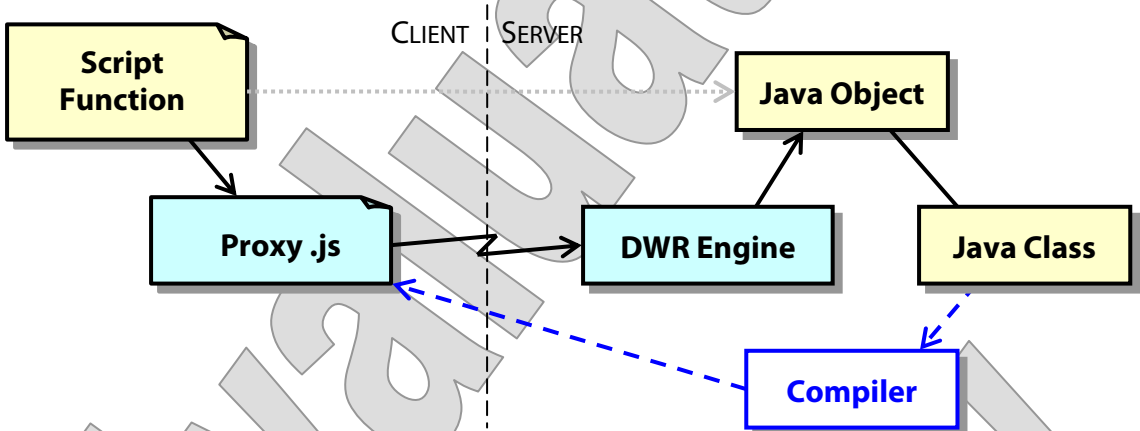
- Any RMI technology works by a process roughly like this:



- **Generate** code from a server-side object that will automate the remote invocation process
- **Deploy** the generated client-side artifacts along with hand-written client software
- Let the client make its calls on a **proxy object** that looks and feels like the actual server-side object, offering the same business methods, but that actually only knows how to marshal an RMI
- **Without much fuss, this basic pattern can be mapped to Java for the server side and JavaScript for the client side.**
 - The generated client-side artifact is a **.js** file that implements the proxy as a JavaScript object type.
 - Scripts can call methods on that proxy that are reasonable facsimiles of the public methods on the original Java class.
 - Those calls result in Ajax requests that are interpreted on the server side to trigger calls on the target Java object.

Direct Web Remoting

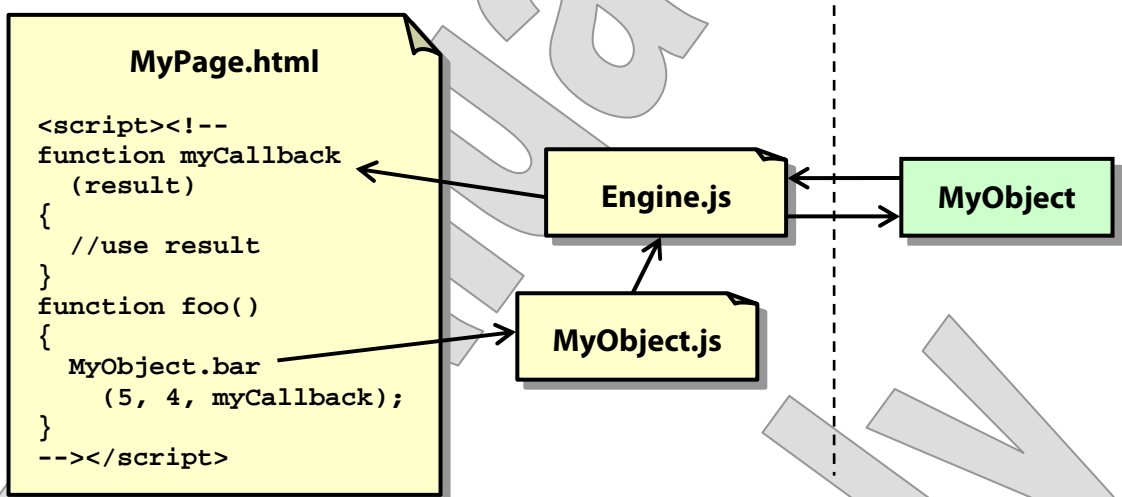
- The leading Ajax RMI framework is **Direct Web Remoting**, or **DWR**.
- Find DWR at:
<http://directwebremoting.org>
<http://getahead.org/dwr>
- DWR expresses the RMI proxy pattern we just discussed, for Java components and JavaScript clients:



- The major variation is that DWR implements a **dynamic proxy**.
 - It generates **.js** files with proxy objects in them at **deployment time**, not at build time.
 - RMIs sent by the proxy (in the form of Ajax requests) are **interpreted** by generic code in the DWR library – there is no generated stub as in the classic RMI pattern.

Direct Web Remoting

- Thus, DWR simplifies the programming model on the client side.
 - Rather than asking the caller to construct an explicit representation of a remote request, DWR provides the **proxy object**, with functions for each supported Java method.
 - The client script makes a function call that is about as intuitive as it could be, with the **same name and parameter list** as the corresponding Java method.



- The remote request is encoded **implicitly** by the proxy function, in concert with the DWR **engine.js** code.
- The **wire format** for DWR is **invisible** to the caller and the called object, and requests and responses are translated on both sides.
- The invocation is still **asynchronous**, and a separate response-handling function is still required. A reference to this function appears as an extra argument in the remote function call.

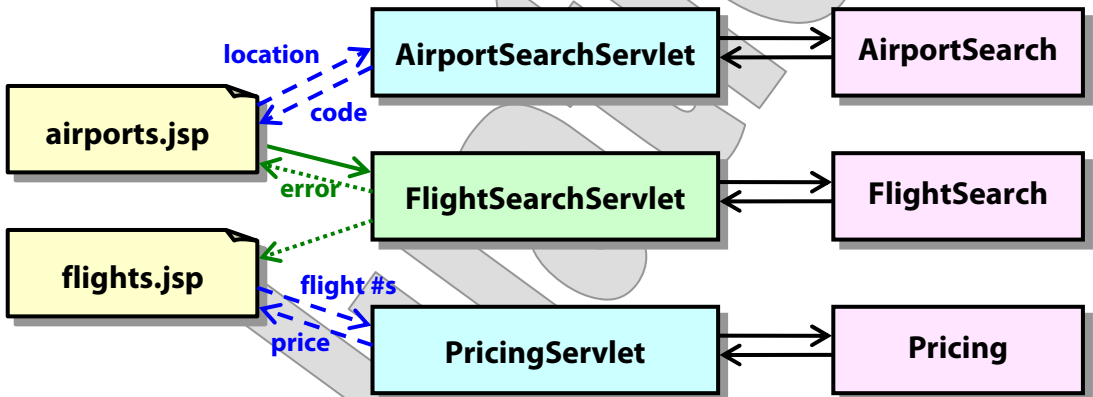
Configuring DWR

- To use DWR in a web application one needs at least three things:
 - The JAR file, **dwr.jar**, to be placed in the **WEB-INF/lib** directory
 - **Servlet and filter** declarations to be placed in **web.xml**
 - A configuration **dwr.xml** file to be placed in **WEB-INF** that declares what Java classes should be instantiated, mapped, and made available for DWR invocations
- The DWR configuration file exposes one or more Java classes for remote invocation, by way of **creators**:

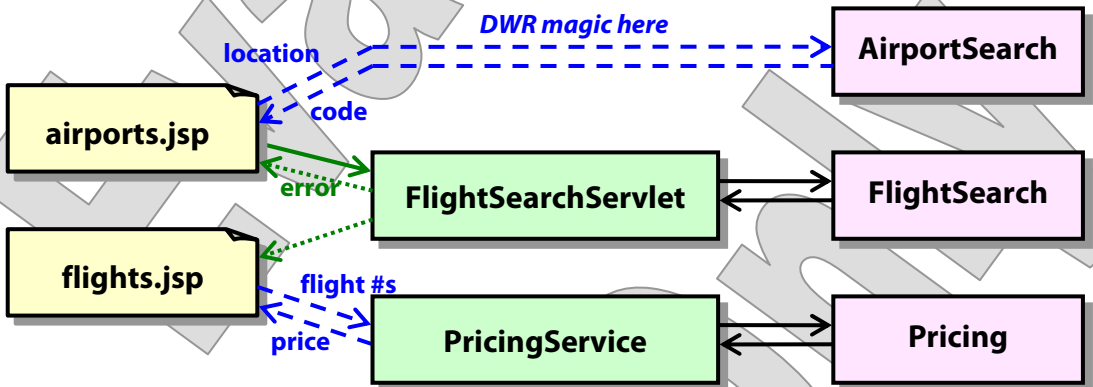
```
<dwr>
  <allow>
    <convert converter="bean" match="me.MyClass" />
    <create creator="new" javascript="service" >
      <param name="class" value="me.NyService" />
    </create>
  </allow>
</dwr>
```

- These are analogous to RMI **Remote** objects (or EJBs, or JAX-RPC/JAX-WS web service endpoints, etc.).
 - There are several styles of creator; the one shown above will simply create a plain old Java object using **new**.
- For any objects whose state must travel in either direction to support a remote method, declare a **converter**.
 - These are like RMI **Serializable** objects (or EJB transfer objects, or JAX-RPC/JAX-WS value objects).

- An example of DWR can be found in **Examples/Flights/DWR**.
- In prior versions of this application, we had Ajax-oriented servlets whose jobs were, essentially, to pipe request parameters along as arguments to method calls on domain objects, and then pipe the return values back in the response:



- Now, we let DWR expose those domain objects:



- In one case, we can let DWR do all the piping to and fro – and simplify the client-side coding considerably in the bargain.
- We still have an intermediary to the **Pricing** logic, but even this is in place as a performance optimization; more soon on this.

- Let's start our exploration of this first DWR application by looking at the configuration files.
- **docroot/WEB-INF/web.xml** still has the conventional servlet, but the mappings for the two Ajax servlets are gone.
 - In their place is a mapping to the front-controller servlet that drives DWR at runtime:

```
<servlet>
  <servlet-name>DWR</servlet-name>
  <servlet-class>
    org.directwebremoting.servlet.DwrServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>DWR</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

- Note that the front controller is mapped to all URLs beginning with “/dwr/”.

- That servlet looks to **docroot/WEB-INF/dwr.xml** to understand the remoting requirements for this particular application.

```
<dwr>
  <allow>
    <convert converter="bean"
      match="cc.flightdb.Airport" />
    <create creator="none"
      javascript="airportSearch"
      scope="application" >
      <param name="class"
        value="cc.travel.AirportSearch" />
    </create>
    <create creator="new" javascript="pricing" >
      <param name="class"
        value="cc.travel.web.PricingService"/>
    </create>
  </allow>
</dwr>
```

- The configuration above tells DWR to do three things:
 - Automatically serialize any instances of class **Airport** that it encounters during remote method calls.
 - Allow remote access to a single instance of the **AirportSearch** class – which it finds at the servlets application scope under the attribute name “airportSearch”.
 - Allow remote access to a single instance of the **PricingService** class – which it creates on its own, using **new**.

- The **AirportSearch** instance that we've promised to make available to DWR is created by a servlet context listener.

- See `src/cc/travel/web/Initializer.java`:

```
public static final String
    AIRPORT_SEARCH_ATTRIBUTE = "airportSearch";
...
public void contextInitialized
    (ServletContextEvent ev)
{
    ...
    context.setAttribute (AIRPORT_SEARCH_ATTRIBUTE,
        new AirportSearch (DB.getAirports ()));
}
```

- The class itself offers utility methods for finding airport codes.

- See `src/cc/travel/AirportSearch.java` – here we show a **javap**-style listing of the method signatures only:

```
public class AirportSearch
{
    public AirportSearch (Airports airportList);
    public void setAirportList (Airports airportList);
    public Airport findAirportByCode (String match);
    public Airport findAirportByLocation
        (String match);
}
```

- We'll focus on this remote-invocation path first, and then come back to see how the pricing part works.

- The result of all this server-side configuration is that a JSP can invoke the **AirportSearch** object quite naturally.
 - See `docroot/WEB-INF/tags/cc/airport.tag`, which uses both stock and generated script files ...

```
<script ... src="dwr/engine.js" ></script>
<script ... src="dwr/util.js" ></script>
<script ... src="dwr/interface/airportSearch.js" >
  </script>
```

- ... and thus is able to get its airport codes as simply as this:

```
function searchForAirport ()
{
  airportSearch.findAirportByLocation
    (dwr.util.getValue (baseId + "-location"),
     receiveAirportObject);
}

function receiveAirportObject (airport)
{
  dwr.util.setValue
    (baseId + "-code", airport.code);
}
```

- But for the need to handle an asynchronous response, this could have been boiled down to a method invocation just as direct and intuitive as calling the same method from another Java class.

- Build and deploy the application as usual:

ant

- Before we even test the application itself, try pointing your browser at the following URL:

`http://localhost:8080/Flights/dwr`

Classes known to DWR:

- ♦ [pricing](#) (cc.travel.web.PricingService)
- ♦ [airportSearch](#) (cc.travel.AirportSearch)

- This page, and the dynamic test interface that it offers, are provided automatically by DWR whenever it is configured with the servlet initialization parameter **debug** set to **true**.

- Click the link for **airportSearch** – and get a nice tour of what DWR is doing for us under the hood:

Methods For: `airportSearch` (`cc.travel.AirportSearch`)

To use this class in your javascript you will need the following script includes:

```
<script type='text/javascript' src='/Flights/dwr/interface/airportSearch.js'></script>  
<script type='text/javascript' src='/Flights/dwr/engine.js'></script>
```

In addition there is an optional utility script:

```
<script type='text/javascript' src='/Flights/dwr/util.js'></script>
```

Replies from DWR are shown with a yellow background if they are simple or in an alert box otherwise.

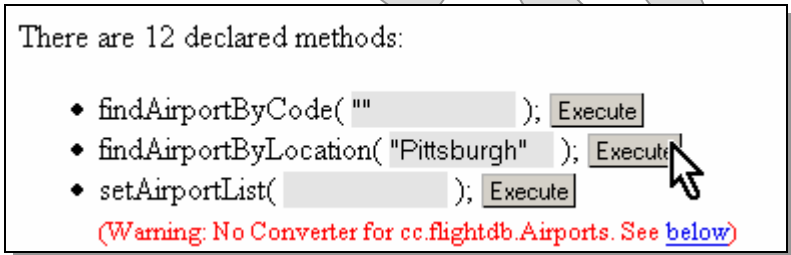
The inputs are evaluated as Javascript so strings must be quoted before execution.

There are 12 declared methods:

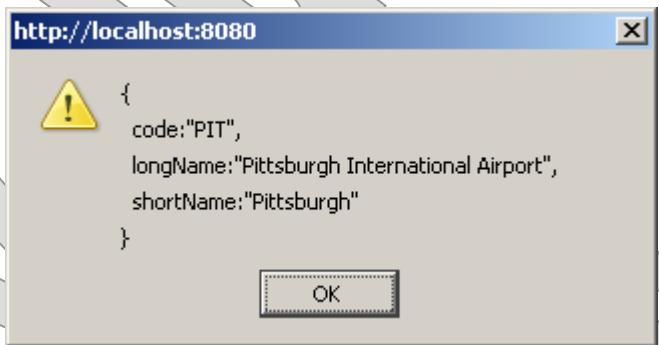
- `findAirportByCode("");` Execute
- `findAirportByLocation("");` Execute
- `setAirportList();` Execute
- (Warning: No Converter for cc.flightdb.Airports. See [below](#))
- `hashCode();` Execute

- Our script functions will use code found in the three **.js** files mentioned at the top of this page – one of which is generated for this specific Java class, and two of which are generic.
- Notice that all the public methods of **AirportSearch** are available for remote invocation – though DWR excludes methods from **java.lang.Object**.

- For each supported method, this page offers a generated test script that's ready to leap into action.
 - Try entering a location string for the second method and clicking the associated **Execute** button:



- The generated script fires off a call to the Java object, just as our client scripts will do, and shows the results as raw JSON:



- So we see that two important pieces are in place for the use case in which the user looks up an airport code:
 - **AirportSearch** itself is **remotely** available.
 - DWR is ready to **serialize** instances of the **Airport** class when it encounters them; the test results in the message box above show such an instance, serialized using JSON.

- Now we'll try the application proper, and pass HTTP traffic through our sniffer application to see how DWR requests and responses look on the wire.

- Run **sniff** from **Examples/HTTPSniffer**.

sniff

Forwarding local port 8079 to local port 8080 ...

- Visit the following URL in your browser, and search for an airport by location:

http://localhost:8079/Flights



- The code is filled in on the resident HTML page.

- In the sniffer console we see the request and response content:

```
POST /Flights/dwr/call/plaincall/airportSearch.findAirportByLocation.dwr HTTP/1.1
Content-Type: text/plain
Content-Length: 236
Connection: Keep-Alive
Cache-Control: no-cache
Cookie: JSESSIONID=9B8C8A2CF33364E58A837338BBDA508D
```

```
callCount=1
page=/Flights/
httpSessionId=9B8C8A2CF33364E58A837338BBDA508D
scriptSessionId=CF3124CA58887628815DD10E7F17F724270
c0-scriptName=airportSearch
c0-methodName=findAirportByLocation
c0-id=0
c0-param0=string:Pittsburgh
batchId=0
```

```
HTTP/1.1 200 OK
Content-Type: text/javascript
Content-Length: 153
```

```
//#DWR-INSERT
//#DWR-REPLY
dwr.engine._remoteHandleCallback('0','0',
{code:"PIT",longName:"Pittsburgh International
Airport",shortName:"Pittsburgh"});
```

- Try out the rest of the application now:
 - Choose two airport codes and request flight options.
 - Choose pairs of flights to see the pricing.
 - You'll see the pricing roundtrip in the sniffer as well:

```
POST /Flights/dwr/call/plaincall/  
pricing.getRoundtripPrice.dwr HTTP/1.1  
Content-Type: text/plain  
Content-Length: 339  
Connection: Keep-Alive  
Cache-Control: no-cache  
Cookie: JSESSIONID=9B8C8A2CF33364E58A837338BBDA508D
```

```
callCount=1
```

```
...
```

```
c0-param0=string:6611
```

```
c0-param1=string:3605
```

```
batchId=0
```

```
HTTP/1.1 200 OK
```

```
Content-Type: text/javascript
```

```
Content-Length: 77
```

```
// #DWR-INSERT
```

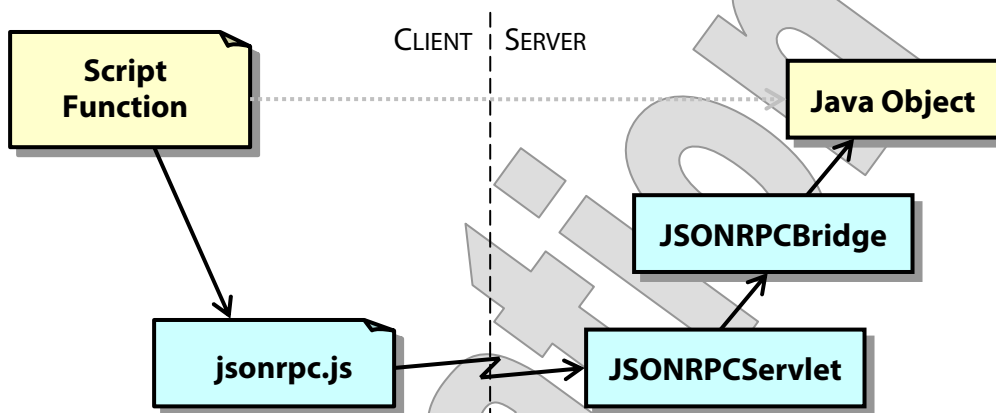
```
// #DWR-REPLY
```

```
dwr.engine._remoteHandleCallback('0','0',308);
```

- We'll investigate the pricing part of the code, and consider some subtler issues of the RMI approach to Ajax, later in the chapter.

jabsorb

- The **jabsorb** framework takes a similar tack to DWR.

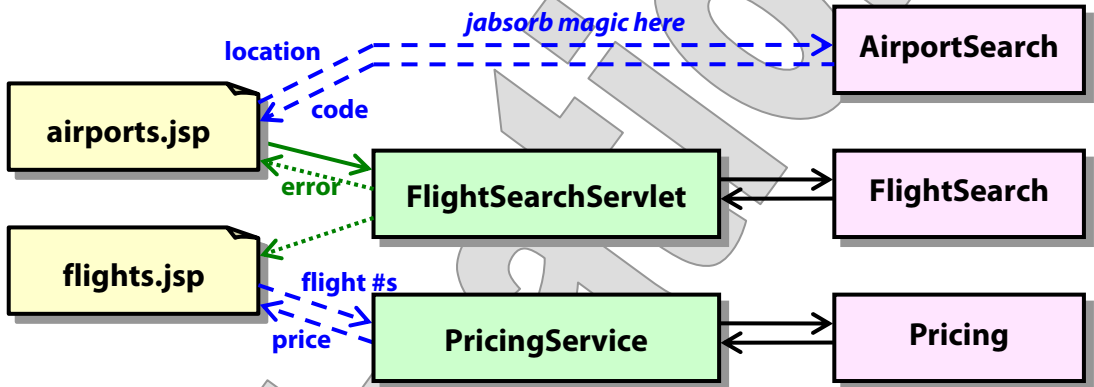


- Find jabsorb here:

<http://jabsorb.org>

- As we'll see, jabsorb is not quite as slick as DWR.
 - Its **programmatic configuration** is generally not as clean and convenient: one must write Java code to create a master **bridge object** for each user session, and **register remote objects** with it.
 - There is **less control** over what objects can be **serialized**.
 - Its **security model** isn't as strong.
- But it uses a conventional wire format, which is JSON-RPC.
 - Remember, DWR's wire format is unique to DWR – it's an open format, and simple enough, but not standardized at all.
 - JSON-RPC isn't a true standard in some eyes, either: it hasn't been ratified by an organization such as the ISO, W3C, IETF, etc.
 - But it is based on JSON, is vendor-neutral, and is easy to read by eye and write by hand as necessary.

- **Examples/Flights/jabsorb** gives us a simple compare-and-contrast between jabsorb and DWR.
- The basic components and request paths are the same:



- Let's start with a look at **docroot/WEB-INF/web.xml**.

– We see the stock front-controller servlet ...

```
<servlet>
  <servlet-name>JSONRPCServlet</servlet-name>
  <servlet-class>org.jabsorb.JSONRPCServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>JSONRPCServlet</servlet-name>
  <url-pattern>/JSON-RPC</url-pattern>
</servlet-mapping>
```

– ... and a context listener – this is our own class that does custom configuration of jabsorb for the application:

```
<listener>
  <listener-class>cc.travel.web.BridgeBuilder
  </listener-class>
</listener>
```

- The **JSONRPCServlet** handles all the RMI requests.
- It looks to a delegate of type **JSONRPCBridge** to help it understand what objects are available, and how to dispatch method calls to them.
- It is the application's responsibility to create this bridge.
 - Many jabsorb examples do this with **scriptlet code** in a JSP.
 - A stronger practice is to do the configuration in a central Java component, such as an **initialization servlet** or **context listener**.
- The key is to create the bridge object at the beginning of a session, as the **JSONRPCServlet** will expect to find it there, as an attribute named "JSONRPCBridge".
- See **src/cc/travel/web/BridgeBuilder.java** for our approach:

```
public class BridgeBuilder
    implements HttpSessionListener
{
    public void sessionCreated (HttpSessionEvent ev)
    {
        JSONRPCBridge bridge = new JSONRPCBridge ();
        HttpSession session = ev.getSession ();
        bridge.registerObject ("airportSearch",
            session.getServletContext ()
                .getAttribute ("airportSearch"));
        bridge.registerObject ("pricing",
            new PricingService ((FlightSearch)
                session.getServletContext ()
                    .getAttribute ("flightSearch")));
        session.setAttribute ("JSONRPCBridge", bridge);
    }
}
```

- Because we've registered an **AirportSearch** instance with the bridge object, we're able to invoke it from JSPs.
 - See `docroot/WEB-INF/tags/cc/airport.tag`, which now includes just one external script file:

```
<script ... src="jsonrpc.js"></script>
```

- Invocation of airport searches now looks like this:

```
var RPC = new JSONRpcClient ("JSON-RPC");  
var baseId = "";
```

```
function searchForAirport ()  
{  
    RPC.airportSearch.findAirportByLocation  
        (receiveAirportObject,  
         document.getElementById  
             (baseId + "-location").value);  
}
```

```
function receiveAirportObject (airport, e)  
{  
    document.getElementById (baseId + "-code").value  
        = airport.code;  
}
```

- By comparison to DWR, the `jabsorb` script code is ever so slightly less simple.
 - We have to create a global **JSONRpcClient** instance.
 - Remote calls all flow through this **master proxy object**, and are dispatched on the server side by the front controller and bridge.

- Again, with the sniffer running, we'll test the resulting application.

- Build and deploy the application:

ant

- Visit the following URL in your browser, and search for an airport by location, as you did before:

http://localhost:8079/Flights

- Over the wire, the remote invocation of **airportSearch.findAirportByLocation** looks like this:

```
POST /Flights/JSON-RPC HTTP/1.1
```

```
Content-Type: text/plain
```

```
Content-Length: 79
```

```
Connection: Keep-Alive
```

```
Cache-Control: no-cache
```

```
Cookie: JSESSIONID=8CACD1975962B01F714A9988512F984C
```

```
{"id":2,"method":"airportSearch.  
findAirportByLocation","params":["Pittsburgh"]}
```

```
HTTP/1.1 200 OK
```

```
Content-Type: text/plain;charset=utf-8
```

```
Content-Length: 137
```

```
{"id":2,"result":{"longName":"Pittsburgh  
International Airport","javaClass":  
"cc.flightdb.Airport","code":"PIT",  
"shortName":"Pittsburgh"}}
```

Suggested time: 45 minutes

In this lab you will implement remote method invocation of a server-side Java object from an HTML page, using DWR. The starter code is a version of the Ellipsoid application, which we've seen in other incarnations in previous chapters; but this one is missing the logic to actually get the page and the server talking. You will configure DWR to expose a new **EllipsoidService** object, and invoke that object from script functions in the JSP.

Students will typically pursue either this lab or the following one – which is a similar exercise using jabsorb – but not both. Choose whichever framework interests you more, and carry out that lab exercise; you may want to review the answer code for the other lab.

Detailed instructions are found at the end of the chapter.

Suggested time: 30-45 minutes

In this lab you will implement remote method invocation of a server-side Java object from an HTML page, using jabsorb. The starter code is a version of the Ellipsoid application, which we've seen in other incarnations in previous chapters; but this one is missing the logic to actually get the page and the server talking. You will register a new **EllipsoidService** object with jabsorb, and invoke that object from script functions in the JSP.

Students will typically pursue either this lab or the preceding one – which is a similar exercise using DWR – but not both. Choose whichever framework interests you more, and carry out that lab exercise; you may want to review the answer code for the other lab.

Detailed instructions are found at the end of the chapter.

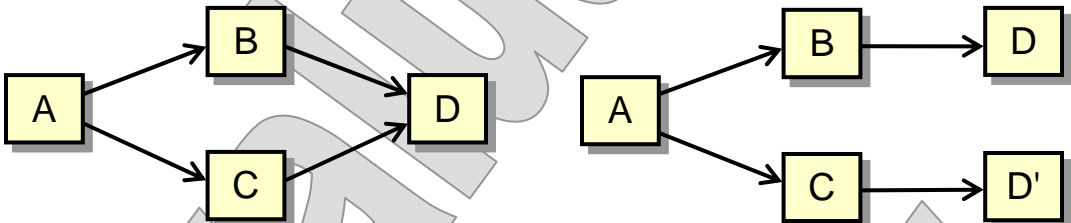
Serialization Issues

- Any RMI framework – Ajax or no – must confront a now-familiar set of technical challenges and design issues.

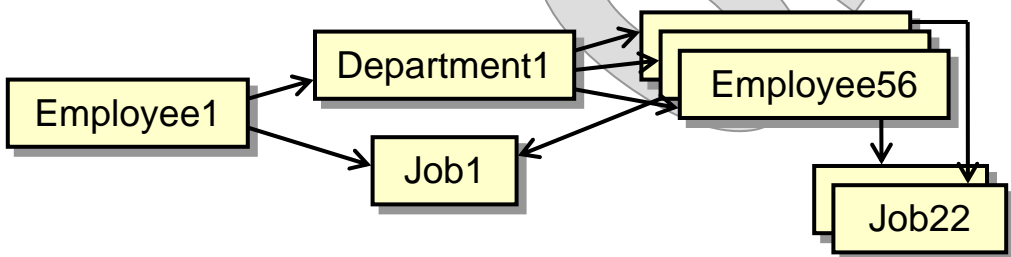
- **Null references:** are these written and parsed cleanly?
- **Circular reference paths:** what happens if objects A and B hold references to each other?



- **Multiple reference paths:** would serializing the object graph shown at left result in a “ghost instance” when de-serializing, as shown at right?



- **Transient properties:** is it possible to exclude certain properties from serialization, if necessary?
- **Controlling response size:** in a complex object graph, is it possible to “cut off” serialization at some threshold, so that the one object we want to transmit doesn’t drag the entire database with it over the wire?



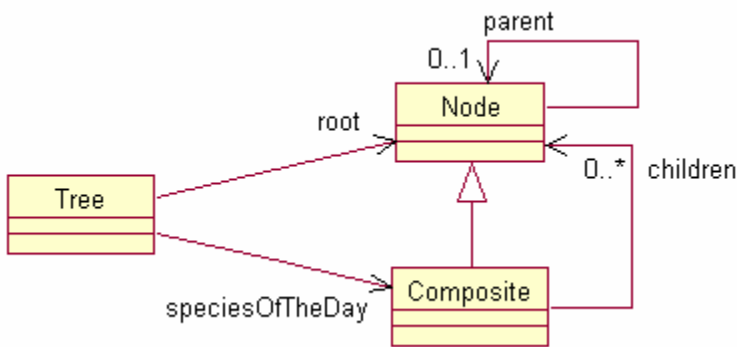
Serialization Features – A Comparison

- By way of illustration and summary, consider the following table comparing four RMI frameworks with regard to how they address the issues mentioned on the previous page:
 - **Java RMI** is based on the **Serialization API**, and most of the issues shown here are really addressed at that level.
 - **JAX-WS** is a web-services API, and it relies on **JAXB** for basic XML binding and serialization.
 - **DWR** and **jabsorb**, we've met ...

Issue	Java RMI	JAX-WS	DWR	jabsorb
Null references	Yes	Yes	Yes	Yes
Circular paths	Yes	Config	Yes	Yes
Multiple paths	Yes	Yes	No	No
Transients	Code	Config	Config	Code

- **Key:**
 - **Yes** means handled automatically, no issues.
 - **Config** means one can have problems but they can be addressed through additional configuration work (or internal metadata, in the case of JAXB annotations).
 - **Code** means one can have problems but they can be addressed by writing additional Java code.
 - **No** means no.
- **Controlling response size is excluded from this table; this is ultimately more of a design issue, and we'll talk about that next.**

- Many of these serialization features are poorly documented.
- The good news is they're also pretty easy to test.
- One feature we can easily try out is how circular reference paths are handled: we'll observe the serialization of a tree data structure in which the primary node type holds references both to children and to parents:



- As it happens, this tree structure holds information about ... trees.
 - Hemlocks, spruces, etc. – the application presents a taxonomy of the pines family.
- We present this example in dual:
 - Examples/Trees/DWR/Step1
 - Examples/Trees/jabsorb/Step1
- The problem is pretty much the same in both cases, but the solution is different.
 - See the domain classes in **src/cc/data** under either example.
 - The UI is also identical, in **docroot**, except for the DWR or jabsorb specifics of remote calls.

- **docroot/index.jsp** makes a simple Ajax call (DWR or jabsorb) to get a hard-coded tree of objects, and then builds an outline of lists and list items to show that information.
 - We don't reproduce the code here, largely because in this example we're more interested in what will come down from the server when we ask for the root node; UI-building is peripheral to that.
 - But here's the key DWR call:

```
function getTreeOfTrees (-)
{
  tree.getRoot (showTreeOfTrees);
}
```

- Here's the jabsorb call:

```
function getTreeOfTrees ()
{
  RPC.tree.getRoot (showTreeOfTrees);
}
```

Circular Reference Paths

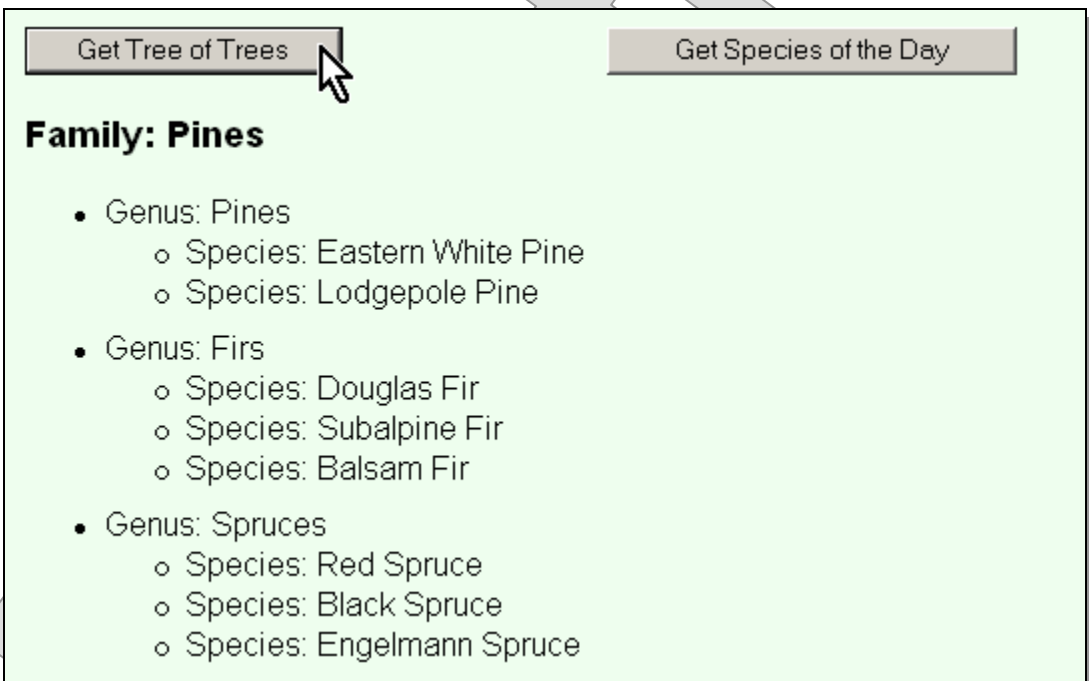
EXAMPLE

- Build and deploy either application (not both! as they share a context URL):

ant

- Test at the following URL, by clicking **Get Tree of Trees**:

<http://localhost:8080/Tree>



- So we can see that the circular reference paths in the data set (for example, Firs has Douglas Fir as a child, which has Firs as a parent) don't make everything come crashing down; both DWR and jabsorb handle these cases gracefully.

- To see exactly what happens, pass the same request through the sniffer application.
- You'll see the following responses. In different ways, each RMI framework is assuring a clean and reproducible object graph.
- DWR defines named references to objects, and then uses/re-uses those named references as appropriate.

– Here's an excerpt from the DWR response:

```
var s0={};var s4=[];var s1={};var s5=[];var
s6={};var s7={};var s2={};var s8=[];var s9={};var
s10={};var s11={};var s3={};var s12=[];var
s13={};var s14={};var s15={};
s0.children=s4;s0.name="Family: Pines";s0.parent=
null;s4[0]=s1;s4[1]=s2;s4[2]=s3;s1.children=s5;
s1.name="Genus: Pines";s1.parent=s0;
...
dwr.engine._remoteHandleCallback('0','0',s0);
```

- **jabsorb** takes a different tack: because the master proxy object is in play on the client side, the JSON representation can be first and foremost a hierarchy.
 - Then, a top-level array of “**fixups**” travels along with the basic tree structure, and the master proxy uses this to “wire up” any reciprocal references before handing the reconstituted object back to the JavaScript caller.
 - We don't reproduce the jabsorb response here, as it's very hard to present a specific reference cycle concisely, especially without digging into the code for the master proxy.

Serialization vs. Remote Invocation

- Any RMI framework must distinguish between serializable and remote objects.
- There will be some technique for sorting these out.
 - **DWR** has you declare them explicitly in **dwr.xml**.
 - **jabsorb** has you register remote objects, and **automatically serializes** anything else it encounters.
- But this split can lead to some interesting design challenges.
- Exposing too many fine-grained objects remotely can produce an explosion of remote invocations, one for every little “get” or “set” operation.
 - The solution usually involves the **transfer object** design pattern, by which separate classes are developed to represent the pure state of a remote object, so that one remote call gets all that state at once.
- Conversely, serializing an object – or a graph of them – can result in a massive response that amounts to a large download, when perhaps the user was led to expect quick feedback.
 - Sometimes the fix here is to go the other way: develop some remote objects in lieu of serializable ones, so that the use case involves multiple requests, each with a manageable response size.
 - Or, the remote interface may be rewritten explicitly to give the caller control of the transfer pace – a practice sometimes known as **throttling**.

Creating Objects on the Client Side

- In that same vein, here's a subtle problem with Ajax RMI frameworks: objects that are passed back and forth in Ajax calls cannot be represented effectively in page requests and responses.
- For example, what if we want to serve a page that's primed with a list of instances of a serializable type, so that the user can choose one and then kick off an Ajax request with that object as an argument to a remote call?
 - There's no decent way to wire a **JSON representation** of an object **into an HTML page** when it is first served, such that it could be passed as part of a later Ajax request.
 - And it's not certain that doing so would be a good idea.
 - The more common practice has been to provide **ID values**; process those IDs on the server side by finding the associated objects; and then handling the request with the help of those objects.
 - The Ajax/RMI approach is to **pass the whole object**; but only the server side is really in a position to manufacture new instances.
- **This turns out to be something of an Achilles heel.**
 - Early in this course, we identified unity of processing logic between Ajax and page request-handling components as a major goal.
 - The wire format of objects plays a big role in that processing logic
 - and that unity, or the lack of it.
- We'll see in the next chapter that JSF frameworks have an advantage in this regard.

“Creating” Flight Records

EXAMPLE

- Let’s go back and take a look at the Flights pricing process.
- See code in **Examples/Flights/DWR**.
 - We don’t explore the jabsorb solution here, but it’s identical from a design perspective (and doesn’t show us anything new at the implementation level). It can be found in **Examples/Flights/jabsorb**.
- Recall that the DWR configuration for the **PricingService** instance is:

```
<create creator="new" javascript="pricing" >  
  <param name="class"  
    value="cc.travel.web.PricingService"/>  
</create>
```

- So, whereas we told DWR to expect to find an instance of **AirportSearch** at application scope, here we’re asking DWR to create an instance of **PricingService** by its own lights.

“Creating” Flight Records

EXAMPLE

- See `src/cc/travel/web/PricingService.java`.

- This class aggregates the **Pricing** utility ...

```
private static final Pricing delegate =
    new Pricing ();
```

- ... and wraps its business method in a method of its own:

```
public int getRoundtripPrice
    (int outboundNumber, int inboundNumber)
{
    ...
    Flight outbound =
        finder.findFlightByNumber (outboundNumber);
    Flight inbound =
        finder.findFlightByNumber (inboundNumber);
    ...

    return
        delegate.getRoundtripPrice (outbound, inbound);
}
```

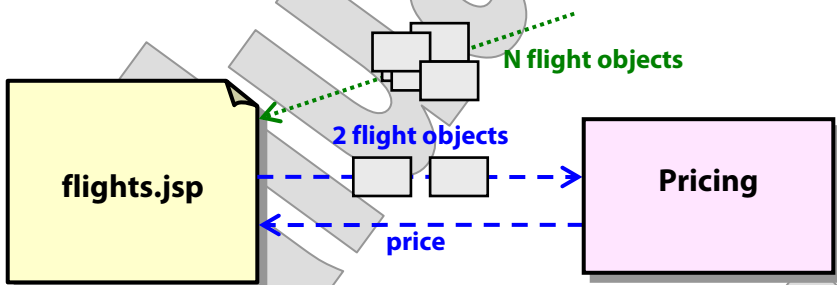
- So this intermediary is doing mostly what DWR would do for us: directing traffic to and from a plain old Java object.
- Why then are we doing it the hard way? Why not just configure a `<creator>` for **Pricing** itself?

“Creating” Flight Records

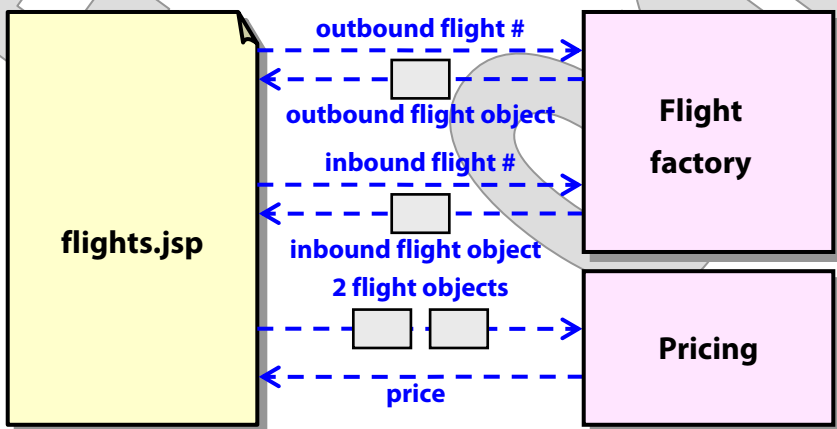
- Here’s the method signature of the raw business method:

```
public int getRoundtripPrice (Flight, Flight);
```

- How would we interact with this remote method?
 - We’d need to provide complete **Flight** instances for each of the outbound and inbound flight choices.
- Where would these objects come from?
 - The client could whip them up using JavaScript – but only if the page had been primed with all the flight information originally:



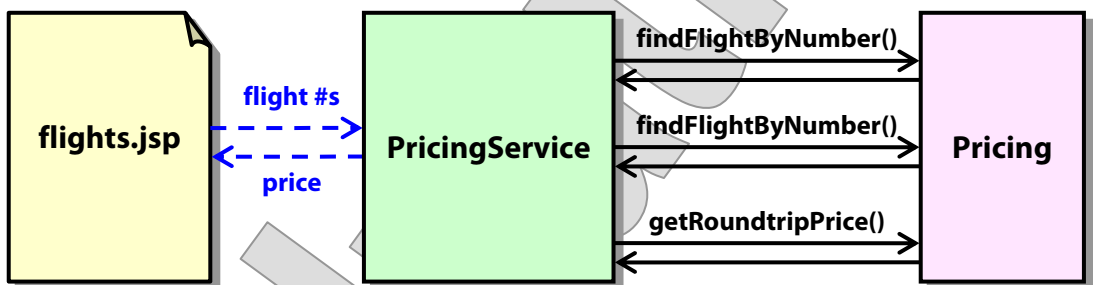
- Or, the client could first request server-side creation of two flight objects, maybe based on provided ID values, and then pass those to the business method – but that’s three roundtrips for one call.



“Creating” Flight Records

EXAMPLE

- It’s not that the above is an unsolvable problem; but generally a simple wrapper or service object is at least as easy, and helps foster the sort of client-service interaction that’s most appropriate (and high-performing).
- So, in this case, we implement the **PricingService** as a simple façade over the process of ...



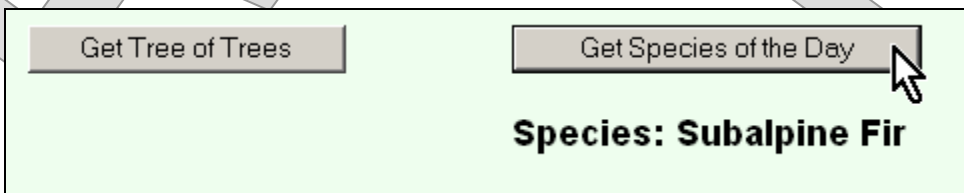
- Finding each of the flight records by the given ID
- Deriving the round-trip price and returning it
- So it’s the **difference between what one can do, and what one should do.**
 - DWR and jabsorb make any of the above interactions possible.
 - But the one we choose is the sanest, in terms of code design, and will perform the best.

- We'll look now at a serialization issue with the Trees application.
 - Do your work in **Demos/Exclude/DWR** or **Demos/Exclude/jabsorb**; again we'll consider these in parallel.
 - The completed demos are in **Examples/Trees/DWR/Step2** and **Examples/Trees/jabsorb/Step2**.
- We've seen that both frameworks can handle circular reference paths as found in this application's tree node type.
- This same data structure provides some challenges, though, when one is interested in serializing a single node, rather than the whole tree or a subtree.

1. Run the HTTPSniffer application, if it's not already running.
2. Test the application as already deployed from the previous example:

http://localhost:8079/Tree

- This, time, click the **Get Species of the Day** button:



3. Fine. But take a look at the response that actually traveled from the server to the browser:

– DWR excerpt:

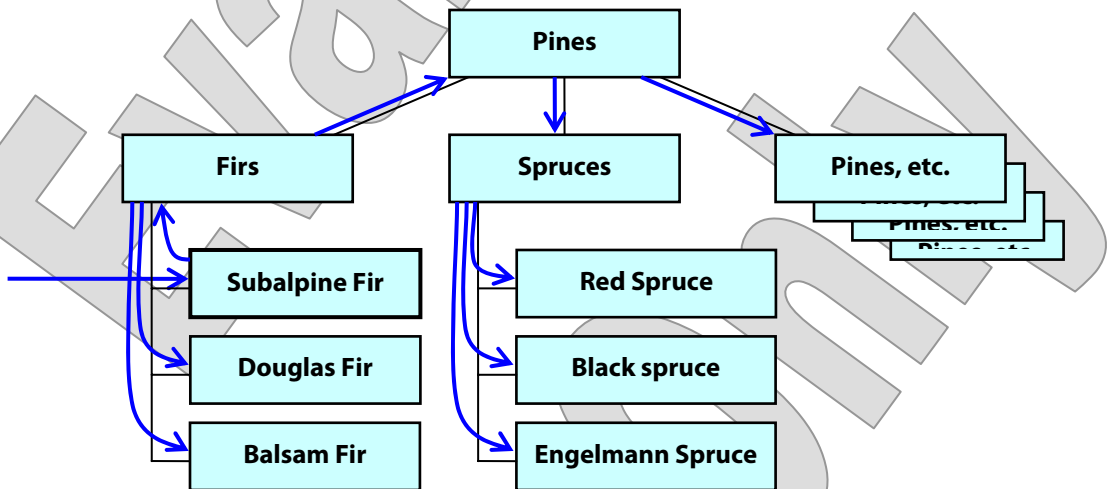
```
...s4.name="Species: Subalpine Fir";  
s4.parent=s2;s2.children=s5;  
s2.name="Genus: Firs";s2.parent=s0;...
```

– jabsorb excerpt:

```
...{"name":"Species: Subalpine Fir","javaClass":  
"cc.data.Node","parent":{"name":"Genus: Firs",  
"javaClass":"cc.data.Composite"},...
```

– The problem is that the serialization process is traversing the parent and children nodes wherever they appear.

– So instead of just one species node, we get the whole tree:



– This problem is typical of object-serialization frameworks.

– In fact it crops up in object/relational mapping and database querying, as well: where to stop following relationships?

- Both DWR and jabsorb give some control over serialization.
- For DWR, just explain to the engine that parent references are to be excluded from serialization.

- Open `docroot/WEB-INF/dwr.xml` and add a `<param>` to the `<converter>` for the **Node** type:

```
<convert converter="bean" match="cc.data.Node" >  
  <param name="exclude" value="parent" />  
</convert>
```

- For jabsorb, it's necessary to register a custom serializer.

- Open `src/cc/data/web/BridgeBuilder.java`, and add the following line of code to register the (already implemented serializer):

```
bridge.registerObject ("tree", new Tree ());  
bridge.registerSerializer (new NodeSerializer ());  
session.setAttribute ("JSONRPCBridge", bridge);
```

- These serializers take some time to create and test – more work than DWR, but also more complete control for more complex customizations we might need to do.

7. Rebuild and deploy the application:

`ant`

8. Test again, and see that everything still works, but your response transfer size has come down considerably.

- DWR – content length was 968, is now 105:

```
dwr.engine._remoteHandleCallback('0','0',  
{name:"Species: Subalpine Fir"});
```

- jabsorb – content length was 1900, is now 78:

```
{"id":2,"result":{"name":"Species: Subalpine  
Fir","javaClass":"cc.data.Node"}}
```

- Note that we'd still get subtrees if we were to serialize a non-leaf node in the tree, because we still traverse the **children** property.
 - This is necessary, of course; otherwise **getRoot** wouldn't work anymore.
 - Ultimately, we'd like method-by-method control of how much information is to be serialized for a given purpose.
 - Neither framework gives us that, and eventually we have to start writing adapter code that takes more complete control over the process.
 - For example we might construct a deep copy of some part of the data set, and “snip off” the parent and/or child relationships at a certain point; then let DWR or jabsorb serialize our temporary copy.

Ajax RMI vs. Web Services

- The term “web services” can mean a lot of different things.
 - Generally, it means an **XML** message format, and often a specific XML vocabulary called **SOAP** – the Simple Object Access Protocol.
- As we study Ajax, it’s starting to resemble web services – especially as we look at it as essentially a context for RMI.
 - The wire format is usually **JSON**, and that of course reflects the JavaScript component that is part of the point of Ajax.
- Beyond wire format, typical Ajax applications differ from conventional web-service architectures in other ways:
 - Web services are generally for **business-to-business** interactions, where Ajax is meant to power **user interfaces**.
 - Web services usually involve **explicit, external metadata** (in a language called WSDL) to allow messages to be created, sent, received, and processed automatically. Ajax RMI frameworks usually use **Java Reflection** to achieve automation.
 - Web services, as part of **service-oriented architectures**, or **SOAs**, are almost always **stateless** and **coarse-grained**. Ajax requests are often **fine-grained**, and a page may rely on **session state** as part of the context of its Ajax requests.
- That said, Ajax “services” can function more like web services.
 - **Mashups** prove this point: they make Ajax requests to services that are more programmatic in nature: stock quotes, map data, etc.
 - Those services are **public, stateless, and coarse-grained**.
 - Service **metadata** is not as mature an area for Ajax, though.

Security Issues

- RMI frameworks obviously introduce some powerful new tools for Ajax design and implementation.
- They also begin to expose your domain model to remote invocation – from just about anywhere, over HTTP.
- Any Ajax interactions must be considered from a security perspective, but when using DWR or jabsorb, one’s exposure is greater, because callers (hackers?) can potentially invoke business logic.
- DWR has a reasonably strong model here.
 - Remember that element name from the configuration file: `<allow>`. They mean that: only creators and converters listed here will “allow” access to remote or serializable objects.
 - Then properties and methods can be **included and excluded**.
 - Security **roles** can be **granted access** to individual remote objects and methods – tying into container-managed authentication at the servlet level.
- **jabsorb** doesn’t have quite the same security mindset.
 - It’s still true that **only objects registered** with the JSON-RPC bridge will be available remotely.
 - The rest, though, is left to your **API design**, and perhaps to **custom serializers**.
 - Any **role-based access control** must be done **programmatically**.

SUMMARY

- Compared to the raw servlets-and-JSP approach of the past two chapters, RMI frameworks offer a clear advantage in convenience, productivity, and code design.
- Especially, this is the first time we've seen a tool that attempts to solve the Ajax problem on both sides of the remote connection.
 - A server-side framework dispatches requests to Java objects.
 - Generated JavaScript, and dynamic marshalling and unmarshalling of requests and responses by stock JavaScript functions, facilitate these remote requests for your own scripts.
- RMI frameworks allow the page/script author to stop focusing on the details of an HTTP request/response pattern, and instead use a nice, natural programming model of objects and methods.
 - A response handler is still required, since at bottom we are still dealing with asynchrony and multiple threads in the browser.
- Ultimately, a big limitation of the RMI approach is that remote calls, and object state transfers, happen differently for Ajax requests than for conventional page requests.
 - At first this seems like a small thing.
 - But the more complex the application, the more this tends to come up: we'd really like to have a mechanism for processing requests that does almost everything the same way, whether it's about to serve a whole new page or just provide a little extra information.
 - DWR and jabsorb fall down on that; we're about to see that JSF frameworks really shine in this regard.