

Comparing Ajax Strategies for Java Applications



Will Provost

Version 1.0

Comparing Ajax Strategies for Java Applications

Version 1.0

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of the author.

A publication of Capstone Courseware, LLC.
877-227-2477
www.capstonecourseware.com

© 2008 Will Provost. All rights reserved.
Published in the United States.

Genesis of this Document

- As part of the development of lab software for a training course on developing Ajax applications in Java¹, the author built a case-study application in several variant forms.
- For purposes of the course, this application helped to highlight differences in programming technique.
- It has also provided the basis for side-by-side comparisons between a few Ajax/Java frameworks.
- This document summarizes that analysis.

¹ Capstone Course 202, “Ajax in Java Applications:” <http://capcourse.com/202>.

Goals

- This is not intended as a product comparison – or, not quite.
- We're focusing more on the different strategies taken by various framework products – and there are many examples of each strategy:
 - Plain-vanilla **XHR** implementations (including the use of JavaScript-only toolkits such as Prototype and Dojo), in which client-side scripts exercise direct control over request and response content
 - **RMI** frameworks (represented by DWR and jabsorb)
 - **JSF** component libraries (examples being RichFaces and Trinidad)
- For each, we're looking for some hard numbers ...
 - **Amount of code** involved in the implementation
 - **Request and response sizes** that result
- ... and also some insights into less quantifiable pros and cons ...
 - **Elegance** of the architecture: how clean/intuitive is the programming model, and how complete is the set of features?
 - Impact on **development and testing time**
 - Availability and quality of **documentation**

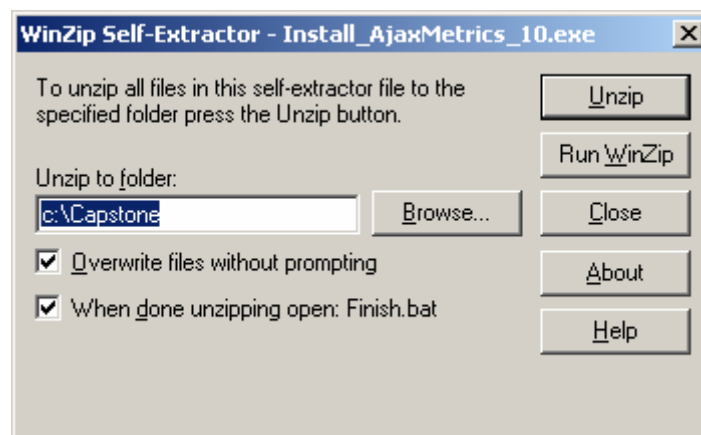
Methodology

- First, the author implemented a simple web application, using servlets, JSP, and the XHR browser object for Ajax features.
 - As much as possible, the Ajax request/response code was isolated, on both server and client sides.
- Then the application was refactored to use progressively more ambitious Ajax frameworks:
 - **Prototype2**
 - **DWR** and **jabsorb**
 - **RichFaces** and **Trinidad**
- When each variant was running well and seemed to have stabilized as a body of code, we started counting things.
- See the later pages on metrics for more on the exact methodology by which we're trying to quantify certain pros and cons of each implementation strategy.
- Generally, we're taking snapshots – of code, of runtime traffic – and looking to categorize with an eye toward scalability, even though our basis is a small application and a quick user session:
 - The parts that are “**overhead**” or otherwise exist only once per application or user session
 - Parts that will **repeat** – with more features that must be coded or more requests that must be handled at runtime

² The Prototype implementation isn't captured in the accompanying lab code, nor was it measured closely as the others were. A quick look made it clear that the differences from the XHR implementation, in terms of code size and request/response size, were very minor compared to the shifts we see with the other frameworks listed here.

Installing and Testing the Code

- To see the source code, and to test it out, do the following:
 1. Download the lab archive from the following URL:
`http://capcourse.com/Library/AjaxMetrics/Install.zip`
 2. Open the ZIP archive and run the installer found there – this is the only file in the archive, **Install_AjaxMetrics_10.exe**, and it is a WinZip self-extractor.



3. Click **Unzip** to put the files in a tree **c:\Capstone\AjaxMetrics**. Paths mentioned the rest of the way through this document will be relative to this root.
 4. Close the installer when done.
- You'll see five implementations of the case-study application, each in a subdirectory of **Examples/Flights**.
 - There are other directories as well, and these will be explained later in the document.
 - There are also many supporting tools (such as DWR, jabsorb, etc.) set up in directories under **c:\Capstone\Tools**.

Installing and Testing the Code

- If you'd prefer to install the lab code on a non-Windows machine, download the following ZIP archive instead:

`http://capcourse.com/Library/AjaxMetrics/Code.zip`

- Then, just unzip this file to any empty directory.
 - The main difference is that Javadoc won't be generated from the source files, as it is when using the installer.
 - Paths mentioned in this document will then be relative to a new directory **AjaxMetrics** under your chosen root directory.
 - Tools will appear in **Tools** under your root directory.

Tools and Environment

- To build and test the installed code, set your environment as follows:
 - The examples require a Java 6 developer's kit. Set your **JAVA_HOME** variable accordingly.
 - Include both your Java-6 **bin** directory and the directory **c:\Capstone\Tools\Ant1.6\bin** in your executable path.
 - Set an environment variable **CC_MODULE** to **c:\Capstone\AjaxMetrics**.
- From **c:\Capstone\Tools\Tomcat6.0\bin**, run **startup** to start the Tomcat web server.
 - Leave it running now, but type **shutdown** from this directory when you're ready to shut down the server.

The “Raw” XHR Implementation

EXAMPLE

- We don’t have the usual full write-up of the example code as we do in the full courseware.
- Neither do we try to explain the inner workings of each variant.
- But we’ll visit three variants, each of which is typical of one of the major strategies we’re considering: “raw” XHR, RMI, and JSF.
- Let’s start with the most basic variant, which is found in **Examples/Flights/XHR**.
- Two different pages make Ajax requests.
- **docroot/airports.jsp** allows the user to search for airport codes by location, with an HTTP GET at a URL of the following form:

```
/AirportSearch?address=location-string
```

- The expected response provides the airport code in plain text.
- A traditional page request causes the application to move from the first page to the second, after running a query for available flights between two airports.
- **docroot/flights.jsp** sends pricing queries, supplying outbound and inbound flight numbers in a URL like this:

```
/Pricing?outbound=code&inbound=code
```

- The expected response is again plain text, this time providing a string representation of the round-trip fare, in dollars.

The “Raw” XHR Implementation

EXAMPLE

- We’ll run through a build-deploy-test process now.
 - All five examples packaged with this document can be built, deployed, and tested in this same way.
 - Note that deploying one will “hide” the other, as they’re all configured to use the same request URLs.

- **Build and deploy this application using Ant:**

`ant`

- **Test in your browser, at the following URL:**

`http://localhost:8080/Flights`

Airline Reservations

Flying from: (or search for airports near:)

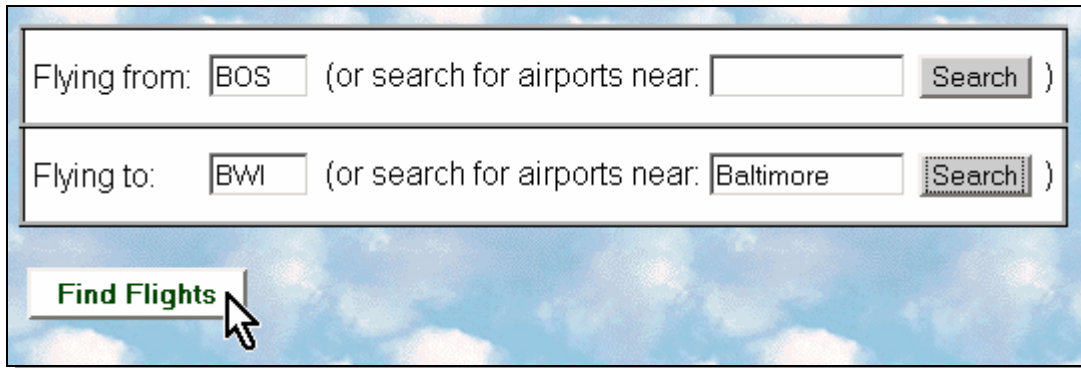
Flying to: (or search for airports near:)

- Let’s say we know our own airport code, but not the code of the place we want to visit. Enter “BOS” in the “Flying from:” field, and then “Baltimore” in the “airports near:” field on the lower right of the table, as shown above.

The “Raw” XHR Implementation

EXAMPLE

- Click **Search**. The Ajax request is sent and handled by **AirportSearchServlet**, and the response gives the correct code, which is then plugged into the form field:



Flying from: (or search for airports near:)

Flying to: (or search for airports near:)

- Now click **Find Flights**. This is a traditional page request, and we get a pair of tables showing flights in each direction:



Select Your Flights

Outbound: BOS to LAX			
Flight #	Departure	Arrival	
<input checked="" type="radio"/> 5211	10:45:00	14:00:00	
<input type="radio"/> 8812	17:30:00	20:45:00	

Inbound: LAX to BOS			
Flight #	Departure	Arrival	
<input type="radio"/> 1986	08:00:00	16:15:00	
<input checked="" type="radio"/> 2367	22:30:00	06:45:00	

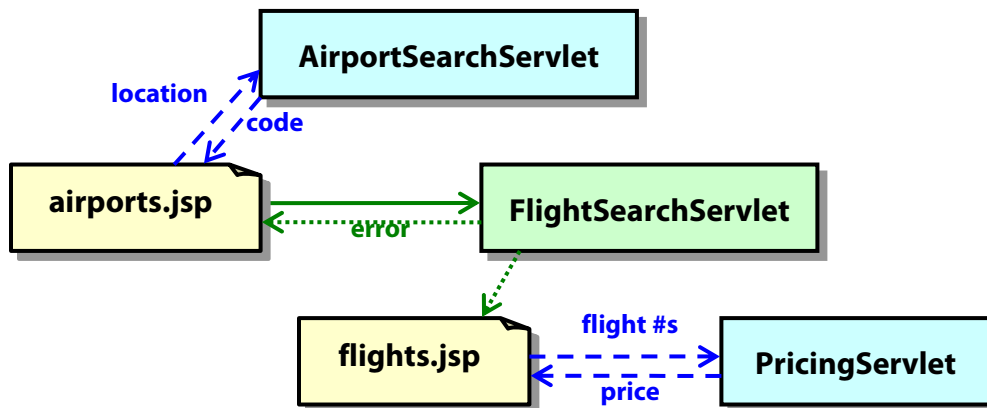
The fare for your selected itinerary is \$498.

- The paragraph at the bottom of the page is a placeholder for price information. Try selecting two flights, and see that a second Ajax request results in a price showing in this paragraph.
- Use the up and down arrows on the keyboard to quickly scan prices while varying your choice of either outbound or inbound flight. Each new selection triggers another Ajax request/response.

The “Raw” XHR Implementation

EXAMPLE

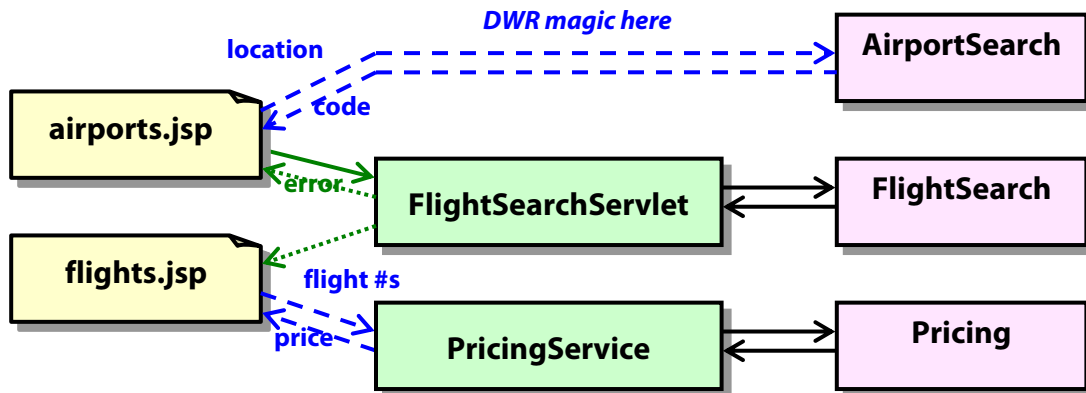
- The JSPs use plain-vanilla XHR code to send the Ajax requests.
 - Requests are HTTP GETs with request parameters.
 - Responses are plain-text answers such as “BWI” and “298”.
- On the server side, the application handles all requests with dedicated servlets:



The DWR Implementation

EXAMPLE

- In the DWR variant, we expose certain domain objects for more or less direct invocation by client-side scripts:

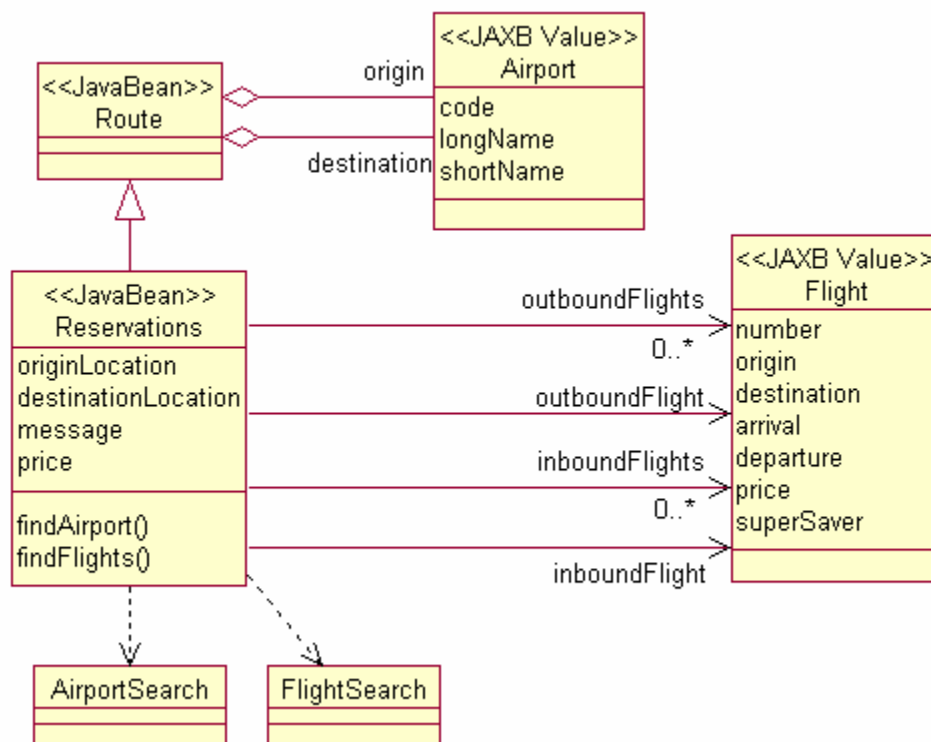


- These objects exist in the XHR implementation as well; they just aren't shown on the previous diagram.
 - Now we can peel away most of the Ajax servlet layer and let DWR make our connections for us.
 - In one case, we can let DWR do all the piping to and fro – and simplify the client-side coding considerably in the bargain.
 - We still have an intermediary to the **Pricing** logic, as a performance optimization.
- The `jabsorb` implementation is almost identical from a design standpoint.

The RichFaces Implementation

EXAMPLE

- The two JSF implementations are dramatically different from the rest; we'll consider the RichFaces variant specifically here.
 - The Trinidad implementation is largely identical, but uses different custom components and event handlers.
- Now, a **reservation** backing bean holds all the necessary state – and it may seem odd at first, because it holds both persistent information and session/conversational state, side-by-side.



- The **AirportSearch** and **FlightSearch** utilities are its gateways to the full travel-agency database.
- Over the session, it accumulates user inputs and relevant data, as illustrated on the following page.

The RichFaces Implementation

EXAMPLE

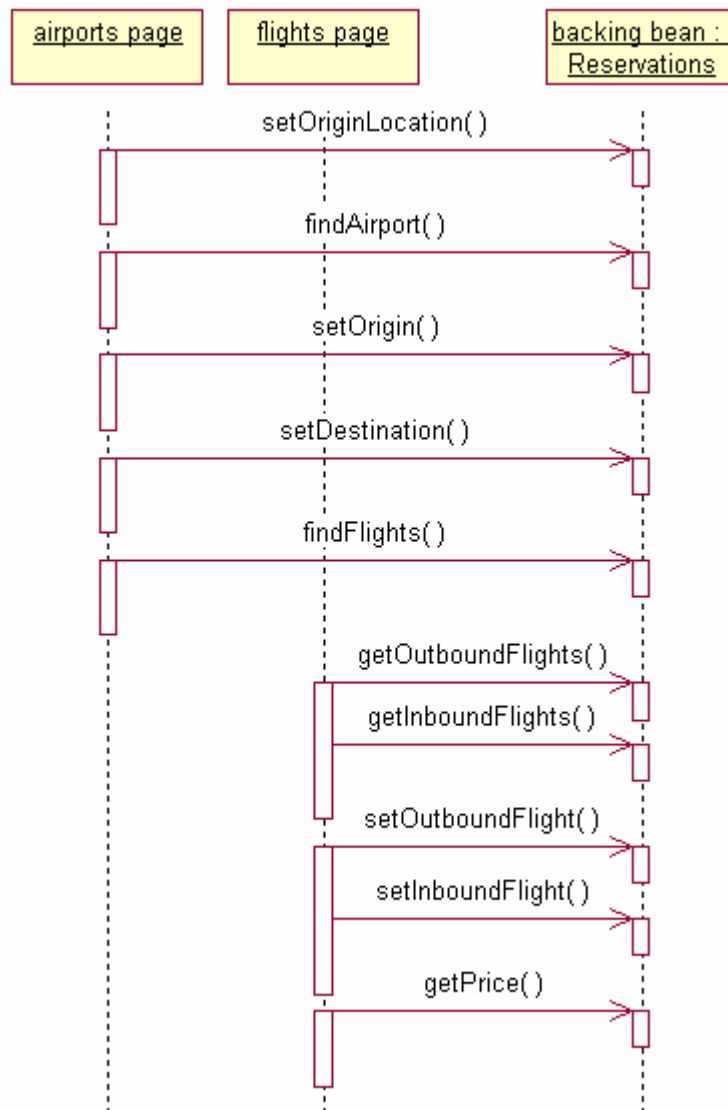
- The main use case for the application now occurs as a series of interactions between JSF views and that backing bean:

1. If the user needs to search for an airport, the Ajax request sets the **originLocation** or **destinationLocation** property, and then calls **findAirport**.

2. Once airport codes are entered, the full page request sets **origin** and **destination** properties, and calls **findFlights**.

3. This results in population of the **outboundFlights** and **inboundFlights** properties, which are read by the second page.

4. Each time the user sets **outboundFlight** and **inboundFlight** values, the page calls **getPrice** and shows the result.



The HTTP Sniffer

EXAMPLE

- For those interested in tracing the runtime behavior of any of the examples, we've provided a tool in **Examples/HTTPSniffer**.
- Build and start the sniffer (in a separate console from your build process, since this process will hang and listen for requests):

```
build
```

```
sniff
```

```
Forwarding local port 8079 to local port 8080 ...
```

- Then, test any deployed application at its usual URL, but with a port number of 8079 rather than 8080 – such as:

```
http://localhost:8079/Flights
```

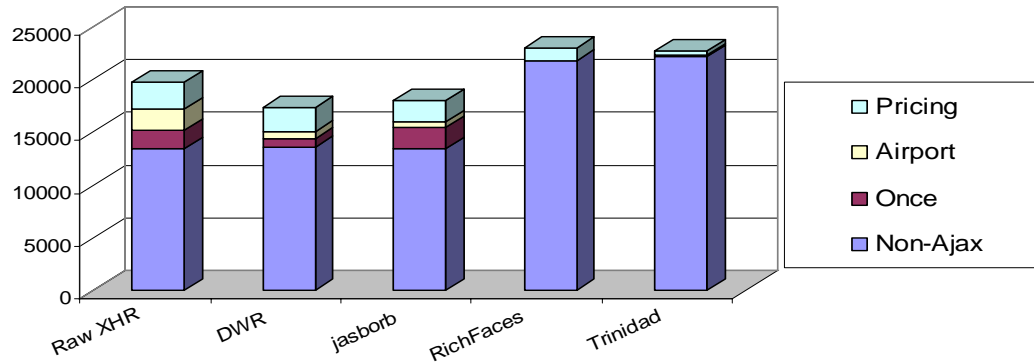
- You'll see request and response content logged to your console (and also to a file **Traffic.txt** which is created fresh each time you run the sniffer) en route to the actual server at port 8080.
- The actual logs of HTTP traffic for each of the five strategies can be found in **Examples/Traffic**.
 - These are **XHR.txt**, **DWR.txt**, etc.

Benchmarks: Code Efficiency

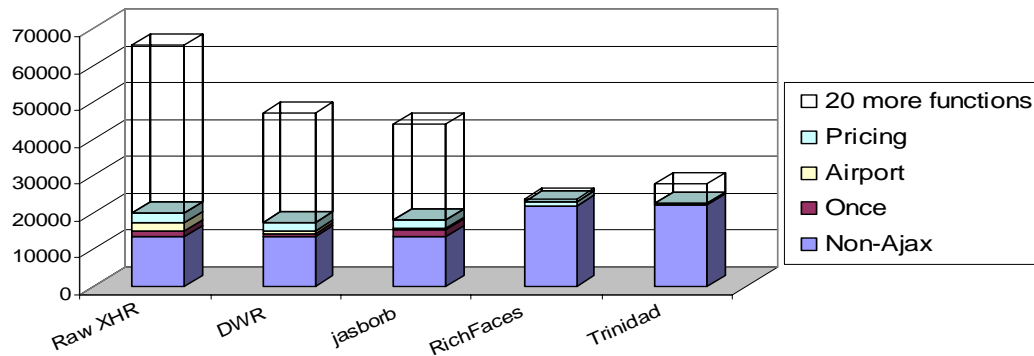
- So – on to the results!
- First we'll look at how much code had to be written for each of the major strategies, to get the same basic application and features.
- In theory, the more sophisticated frameworks will offer a benefit in the form of reduced coding effort.
 - There may be an **investment** of additional coding to set up the framework for an application: configuration files, extra JARs in the build path, imports and includes in pages and source files.
 - But then there should be a reward for developers, in that they have to write **less code per page** or **per class**.
- On the following page is a chart of the total code sizes of our various implementations.
- For each, we've analyzed the code base and found four categories of functionality:
 - **Non-Ajax** code – everything from the deployment descriptor and basic HTML layouts to the servlets and filters that make the basic application function.
 - Ajax code that must be written **once per application** – for instance the configuration of the DWR or jabsorb servlets
 - Ajax code that must be written **once per business function** – and here we lump individual pages and individual server-side objects together, saying for example that certain code is unique to the process of looking up an **airport code** or pricing an itinerary, and other code necessary to the **pricing** function

Benchmarks: Code Efficiency

- Those categories are each measured in bytes of code written, and stacked in the following bar chart:



- At first, this seems to suggest that “the simpler, the better” – as JSF frameworks just bloat the code base, and RMI frameworks offer only limited benefits.
- This is why we’ve broken out per-function coding!
- Consider the same chart with the added extrapolation of 20 or so additional business functions of similar complexity:



- The investment of framework configurations starts to pay off for more complex applications.
- Now JSF seems to fulfill its promise, at least for those applications.

Benchmarks: Code Efficiency

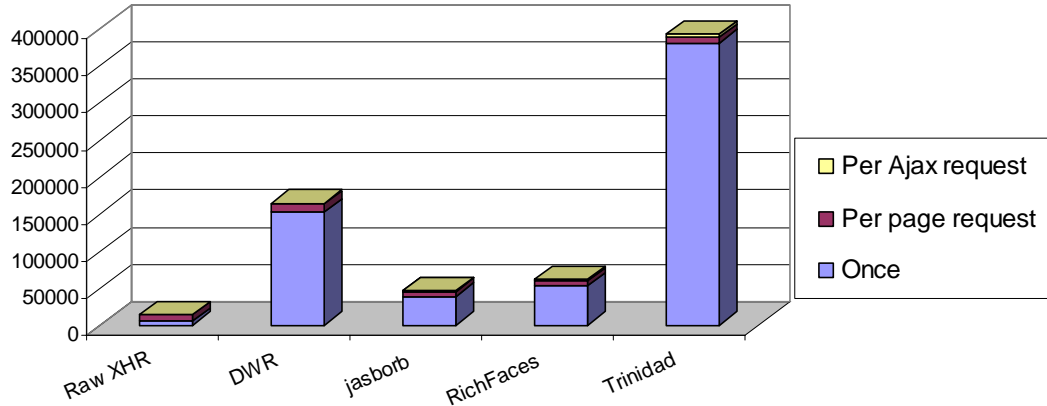
- Following are a few notes on the findings on the previous page.
- Only code for presentation-tier components was counted.
 - The **domain classes** in **cc.travel** were left out.
 - So were the JAXB **persistence classes** in **cc.flightdb**.
- We didn't review concrete examples of Prototype or Dojo in this course, but these were considered as well.
 - They were slightly more code-efficient on the client side – and, naturally, identical on the server-side.
- Some of the category choices are subjective.
 - Does the configuration of Trinidad's view handler count as Ajax code? Or is it part of the framework of a non-Ajax application?
 - RMI frameworks seem to get charged for Ajax-dedicated code where JSF frameworks get a pass, because (for example) DWR is only about Ajax, while RichFaces is about general facility in building JSF/web applications.
- We extrapolated the per-function code size based on the simpler airport-code lookup, rather than the pricing query.
 - The latter involves some atypical client-side logic.
 - For example, if we extrapolated from the `<a4j:jsFunction>` and `<script>` elements needed in the RichFaces implementation, the per-function cost jumps 3122%.
 - This is no knock on RichFaces; the Trinidad implementation sidesteps this logic – and winds up sending a number of unnecessary Ajax requests as a result.

Metrics: Runtime Efficiency

- The next question we'll consider is: what does the use of Ajax do to an application's consumption of network bandwidth?
 - And, how is this impact affected by the choice of Ajax technology?
- We logged the HTTP traffic between browser and server (using HTTPSniffer) for a simple scenario:
 1. Visit the Flights application (page request).
 2. Look up an airport code by location (Ajax request).
 3. Enter the other airport code directly.
 4. Search for flights between those two airports (page request).
 5. Select flights and see the price (Ajax request).
 6. Change one flight choice and see an updated price (Ajax request).
- The average sum of request size and response size, in bytes, is then derived for
 - Resources that are only requested **once per user session**
 - Individual **page requests**
 - Individual **Ajax requests**

Metrics: Runtime Efficiency

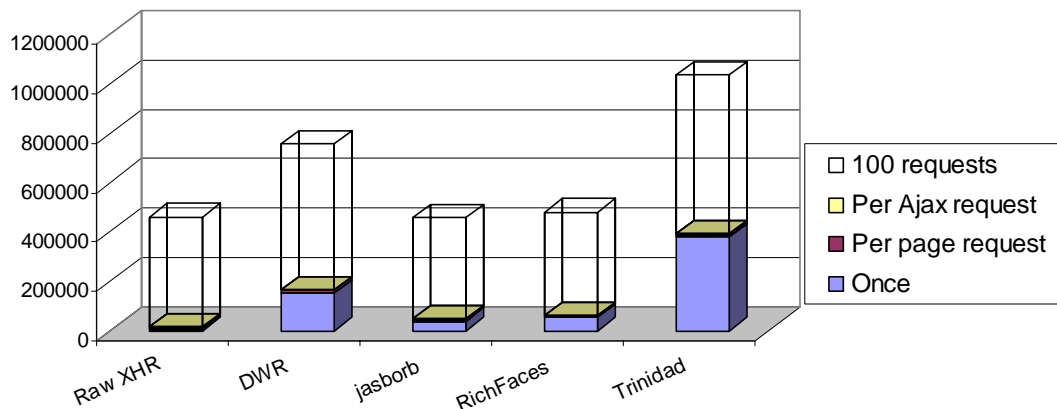
- Here are the results for our five target Ajax technologies:



- Two observations:

- There is a general trend toward request and response bloat as the technology gets more sophisticated; i.e. easier development might mean less efficient runtime interactions.
- DWR and Trinidad lay it on pretty thick – bandwidth usage is out of proportion to the functionality they’re delivering.

- Here’s an extrapolation of bandwidth usage assuming 100 total page and Ajax requests per user session:



Metrics: Runtime Efficiency

- Following are a few notes on these findings.
- DWR discourages caching of its primary script file, **engine.js**.
 - Perhaps they have their reasons, but this seems a poor choice.
 - The file is **44 kilobytes** in size.
 - Practically, this means an extra 44k download on **each page request** – instead of an HTTP 304 “not modified” message.
 - The generated interface script files aren’t cacheable either, but the bandwidth hit is minimal.
 - We’ve assumed a fix for this in our metrics; otherwise the per-request cost would be, literally, off the charts.
- Trinidad has a similar problem with its main script file, which would incur a 185k hit per page request.
 - Again, we’ve assumed a fix and charged this large round-trip to the session, rather than as a per-request cost.
- So both frameworks look wildly inefficient for a single use case, but come a little more into line when we extrapolate to more user interactions per session.

Intangibles: Completeness and Correctness

- All the benchmarks we've seen seem to argue for the use of some framework for Ajax development.
 - To a lesser extent, they show that JSF holds a lot of promise for larger enterprises, as it seems to scale up nicely over more functionality and more requests.
- Harder to quantify are various questions of framework quality, architecture, and ease of development.
- Between RMI and JSF strategies, there's a familiar tradeoff between simplicity and sophistication.
 - **RichFaces** and **Trinidad** require a **commitment to JSF** in order to take advantage of their UI and Ajax features, and JSF involves certain overheads such as configuration files, tag libraries, etc.
 - **DWR** and **jabsorb** are relatively simple to plug in to an existing servlet/JSP architecture – but they leave **more request-by-request programming** to the developer.
 - They also suffer the significant limitation that they only publish objects for remote invocation via Ajax requests; page requests can't use the same channel or serializable data formats.
- **Error handling is a major concern for Ajax developers, since the usual browser presentation of HTTP errors isn't in play.**
 - Generally, JSF is in a better position here, too, as our earlier exercises showed: a component-oriented error-reporting framework is built in and ready to use.

Intangibles: Ease of Development

- It's also worth considering how productive one might be when working with a given tool.
- JSF can be rather awkward in this regard.
- Applications are bigger, take a little longer to deploy, and often a lot longer to install once the server has them.
 - You noticed the **slower build/test cycles** during labs: RichFaces and Trinidad both do a good bit of setting up before they're really ready to handle requests.
- JSF applications are highly stateful, because each view encodes unique identifiers on the client side that must be resolved on successive requests.
 - **Incremental testing** is therefore harder.
 - Instead of just reloading the fourth page of a five-page process, we usually have to **roll back to page one**, reload, and run the scenario forward again, just to test a change to page four.
- Unfortunately, one can't assume much about the quality of developer documentation when choosing an Ajax framework.
 - Ultimately, Ajax isn't a standard; and implementors tend to decide for themselves how much documentation is enough.
 - All of the frameworks we've used have some basic documentation, and some have developer's guides, and so forth.
 - But this is still an emerging area of technology, and so documentation is often incomplete, and examples of specific programming techniques can be hard to find.