



Capstone Courseware, LLC

33 Boylston Street
Jamaica Plain, MA 02130

877-227-2477
capstonecourseware.com

Ajax in Java Applications

Version 5.0

Instructor's Guide



Overview

Probably the main thing to understand about this course is that it samples various approaches to supporting Ajax in Java EE web applications – rather than proposing a single best practice and showing that in depth. Ajax techniques are still evolving; there is no relevant standard; there is vibrant competition between various frameworks and toolkits; and ultimately the relationship between Ajax and Java has yet to be fully understood. Also, there are too many popular Ajax frameworks out there to try to cover all of them even lightly.

So, our policy has been to (a) focus on what's challenging to the Java EE developer about integrating Ajax functionality, (b) break down the available solutions into a few digestible categories, and (c) let just two candidates from a particular category illustrate patterns that are found in that category. For example we look at DWR, but not just as one among scores of undifferentiated options. Rather, we see DWR as a leading implementation of the RMI approach to Ajax client/server interactions. We show a less popular tool called jabsorb side-by-side with DWR, and in doing so we separate the architectural approach (RMI) from the syntax and other product specifics. We do the same for JSF frameworks, letting RichFaces and Trinidad demonstrate the general “win” of JSF for Ajax. (And we're echoing a pattern from Course 201, which is often a lead-in to this one, and which contrasts Prototype and Dojo as JavaScript frameworks for Ajax.)

We also spend a good bit of time on programming techniques for Ajax that have nothing to do with any particular add-on framework: using JSP tag files to streamline the component architecture of an Ajax application; best tools for processing JSON and XML on the server side; etc. This nuts-and-bolts material actually gets us started, and then we climb through increasingly “slick” toolkits; but, as always, those ugly little techniques that we show first are still useful even once a DWR or a RichFaces is in play, because the framework will never do 100% of the job for 100% of its users.

So the goal of this course is to develop a solid understanding of the complexities of Ajax programming for Java EE – far more than simple literacy, but not an in-depth training in any one of the frameworks we consider, either. Students may not be able to walk out of the class and build the best, most elegant and complete RichFaces application, for example; but they will know why they're using RichFaces in the first place, the key concepts that make JSF, Facelets, and RichFaces function the way they do, and how to explore the rest of the technology stack.





Timeline

The following breakdowns are approximate, and every class will vary.

Day 1

1½ hours	Chapter 1
2½ hours	Chapter 2
2½ hours	Chapter 3

Day 2

2½ hours	Chapter 4
½ hour	Chapter 5
½ hour	Chapter 6
3 hours	Chapter 7
½ hour	Chapter 8

This results in a split between day one focusing on standards-based techniques for Ajax applications – JSP and servlet techniques – and day two moving into add-on frameworks including DWR, jabsorb, RichFaces, and Trinidad.

Chapters 5 and 6 are super-quick primers on JSF and Facelets. For some audiences they will be unnecessary – and you can definitely skip them without worrying about continuity in the book or the labs – but they don't affect the timeline much one way or the other.





Tools Deployed with the Lab Software

This course's software requires separate setups of the 6.0 JDK, the Crimson text editor, Firefox 3.0, and/or Eclipse WTP; otherwise it is self-contained, as the lab installer sets up not only the lab software but necessary tools – these will all be found in directories under **c:/Capstone/Tools**:

- Some Apache Commons libraries that are used by other tools are deployed in **Commons20071101**.
- **DWR**, in **DWR2.0**. Several exercises in Chapter 4 use DWR; the only file we really need is **dwr.jar**, and this is included in web applications as part of Ant and Eclipse builds.
- **Facelets**, in **Facelets1.1**.
- **jabsorb**, in **jabsorb1.2**. Several exercises in Chapter 4 use jabsorb. We use several JAR files, and these are included in web applications as part of Ant and Eclipse builds. There is also a JavaScript file, **jsonrpc.js**, which must be copied into the appropriate web root directories by the Ant build and is simply dropped into the **docroot** directory by the Eclipse overlay.
- **JSF**, in **JSF1.2**. JSF is fundamental to all the exercises in Chapter 7, whether using RichFaces or Trinidad. The JAR files are included in Ant and Eclipse builds.
- **JSTL 1.2**, in **JSTL1.2**. The JARs and TLDs for JSTL tags used in many of the course-exercise JSPs are found here.
- **RichFaces**, in **RichFaces3.1**. Several exercises in Chapter 7 use RichFaces; the JAR files are included in web applications by Ant and Eclipse builds, and also placed in the compile path for some lab exercises.
- **Tomcat 6.0**, in **Tomcat6.0**. This is our target web server for deployment and testing.
- **Trinidad**, in **Trinidad1.2**. Several exercises in Chapter 7 use Trinidad; the JAR files are included in web applications by Ant and Eclipse builds, and also placed in the compile path for some lab exercises.





Ant Build Process

We use **ant** for our web-application builds. (One standalone-application project uses simple **build** and **run** batch files.) Though most students will be happy to let the **ant** command take care of things, some students (and most instructors) will want to understand the inner workings here a little better. Each project in the course has its own **build.xml** and **build.properties** files; these rely on the master environment variable **CC_MODULE** to import targets and properties in a central directory, **%CC_MODULE%\Ant** which is typically **c:\Capstone\AjaxJava\Ant**. Information here defines a routine for building a web application.

The build will create a **build** subdirectory that is the root of a Java web application, and compile Java classes from **src** into **build/WEB-INF/classes**. It will copy the full tree of files under **docroot** to **build** – these are our JSPs, CSS, supporting data and image files, and configuration files including **web.xml**. Then the build will draw in JPA and JSTL libraries and archives to the appropriate places in the web-app structure under the **build** directory.

Prior to building (and you could do the above tasks only, by typing **ant build**), the default Ant target (**all**) will undeploy any prior version of the application, and after the build it will (re-)deploy. It does this by generating a **<Context>** file into Tomcat's **conf/Catalina/localhost** tree, thus pointing Tomcat to the **build** directory as the root of the web application.

All this means that students can simply type **ant** from the command line set to any example step, demo or lab directory as the working directory – if there's a **build.xml** in the directory, you're good to go. Any change to Java, JSP, XML, etc., will be reflected in the re-deployed application – just give Tomcat its 30 seconds to sweep for changes and re-install before testing from a browser.





Eclipse Overlays

Capstone provides an optional package of workspace and project files for Eclipse WTP 2.0 for this course. (See the course Setup Guide for download URLs.) Instructors, use this package on your own initiative and at your own risk. You should have experience yourself with Eclipse WTP before using the overlay package in the classroom, and preferably only students with Eclipse experience will use the overlay themselves. The workspace and projects have been tested lightly with the course but are not part of the standard product.

That said, this overlay should save a good deal of work for those who wish to use Eclipse as the primary code editor for the course. All projects support interactive deployment to the managed Tomcat server, testing, and interactive debugging on the server. See the file **ReadMe.html** (shown when the workspace is first opened) for general notes on how to use the Eclipse overlays for Capstone courses.

Again, Capstone Courseware can only offer complete technical support on the standard course, and while we hope this overlay is convenient, it is not as thoroughly tested as the core lab image at this time. If a given exercise is giving trouble, please be certain to build and run it from the command line, using the SDK tools as prescribed in the student guide, before contacting Capstone.

We always welcome feedback on our courseware, and we would appreciate whatever comments and criticism you might have. Eclipse, like all IDEs, tries to promote ease of use by providing many different ways of doing the same thing; this is convenient for users but does leave a lot of questions as to what's best practice. At this time we believe consensus is still forming around a number of basic practices, and we'd like to hear how you use Eclipse for training situations and how this overlay works out for you. Please contact Will Provost at provost@capstonecourseware.com.





Teaching Notes

Chapter 1

This is the typical overview chapter that sets up the rest of the course. Since we assume an understanding of Ajax itself before the course begins, this chapter establishes that we'll be focusing on the challenges involved in implementing Ajax for Java web applications. So we worry more about the server side – but not just the server side. We're looking to go beyond the understanding of the page author or JavaScript programmer who writes some XHR code, or uses Prototype or Dojo or some other toolkit, lets magic happen, and processes the HTTP response. It should be clear by the end of this chapter that, while bolting a little Ajax functionality onto a web application here and there is pretty straightforward, really absorbing Ajax into the inner workings of a Java webapp will require some deeper thought and some coordination between HTML, JSP, servlets, and possibly other components. Otherwise Ajax has the potential to bring chaos to an otherwise orderly architecture and development process.

Maybe the most eye-opening part of this chapter, for many students at least, will be the idea of MVC being applied to Ajax. Many Ajax frameworks are not MVC-oriented, and at first blush the idea may seem odd – where, for one thing, is the “view” in an Ajax operation? So it can be a bit of a revelation that MVC really still makes sense, and get people thinking much more in architecture and pattern terms about Ajax processing. Then the JSF strategy especially seems compelling, and we can close with a side-by-side comparison that more or less defines the arc of the rest of the course.





Chapter 2

Some students will be familiar with JSP tag files, and some won't. This is the first of a few techniques and technologies that we need for our discussions and exercises, but can't stop to teach in depth. So we do a quick review and try to stick to the essentials as they'll be needed for Ajax development.

Then, we start to see the JSP tag file as a neat way to encapsulate. It won't be the last encapsulating technique we try, either, but it's a nice start: being able to marry UI definitions to functions that those UI components require is big all by itself. Managing the inclusion of scripts (each, once only, or parameterized per UI widget) is an even better technique. But probably the biggest win in this chapter is the use of tag files to consolidate the coding necessary to deliver the same HTML fragments as part of page and Ajax requests; if one thing gets through from this chapter, that should be it.

Chapter 3

This chapter is mostly about managing the variety of Ajax wire formats that might occur. As such it becomes a bit of a laundry list of different APIs and tools that we can draw on for different sorts of processing – especially JSON and XML – and it can get a little dry. The beginning section on mapping strategies – servlet-per-request, mixed-use, front-controller – is a bit more interesting conceptually, and foreshadows the RMI and JSF solutions a bit. And we finish with a section on sessions and security that, while it doesn't contain any huge surprises, should help check some boxes in everyone's heads regarding what they can and can't expect from Ajax requests when processing them in a servlet context.



Chapter 4

Some students may have experience with some sort of RMI – in fact it's pretty likely in a mixed group of 10 or so students that someone will have done Java RMI, EJB, CORBA, XML-RPC, SOAP web services – or maybe DCOM from another life. Those people will recognize DWR straight off, and everyone else will need to be introduced to the joys and the sorrows of RMI. The joys are shown early, with demos of DWR and jabsorb.

The rest of the chapter is progressively more sorrowful, as we consider the limitations of automated object serialization, sharing remote objects, etc. The species-of-the-day examples show a fairly satisfying solution to the problem of over-serializing a response; but it's worth mentioning at this point that there's no one perfect solution to this problem for the general case. It often means writing an adapter layer just for DWR – and there are other factors that drive us in this direction eventually, as the Pricing example has already shown at this point.

Regarding serializable-versus-remote: if the group seems interested, you might put this in the historical context of OO-vs.-SOA. SOAs tend to break OO encapsulations by splitting application logic into state (serializable objects) and behavior (remote objects) – resulting in what Martin Fowler has called the “anemic domain model.” EJB, CORBA, DCOM, and web services have all had to wrestle with this, and now Ajax has to do the same thing.

Make sure to discuss the final major point from the summary page – this is indeed the biggest issue with RMI frameworks for Ajax, and becomes a real pain as an application grows. This also leads in nicely to the JSF approach as covered in the next few chapters.

One note about jabsorb: notice the way it handles collections, by creating a pseudo-property **list**. DWR is more natural here; also this is not documented well in jabsorb at all, just something one must discover empirically.

Chapters 5 and 6

Each of these two chapters may or may not be necessary for a given group; find out about your students' backgrounds and adjust accordingly. Conversely, there may be sufficient interest in JSF to go a little deeper than just the quick example covered here. As with JSP tag files, we're trying to get the kernel of JSF as a technology across to students, with a further focus on what parts of JSF are really key to Ajax implementation – and leave it at that. Chapter 6 on Facelets is easily the most skippable chapter, but it's nice to get a little intro to it before jumping into the RichFaces examples in Chapter 7, if time allows.





Chapter 7

Even with the intros in the previous couple of chapters, it may be jarring for students to jump into things like RichFaces and Trinidad. If they haven't done any JSF up until now, it will no doubt be somewhat bewildering, and you may need to take some time going over those first few examples to explain where everything is and how it fits together. But, the real focus of the chapter is on the way JSF component libraries can handle Ajax requests, and that part is more well-bounded: we're talking about the JSF view and how it "lives" on both the client and server, so that HTTP requests are, up to a point, the components talking to themselves over the wire. Plenty of interesting stuff happens behind the UIComponent tree on the server side – but none of it varies between page and Ajax requests. The only differences are the things we discuss here: partial submits, partial rendering, error handling, polling, and so on.

There is no specific example that shows off these component libraries for their UI sophistication, but after the initial Ajax examples (Flights), it might be worth visiting the demo sites for RichFaces and Trinidad, and letting those give each product a little extra "wow" factor.

You might note, during the Polling demo, that the default polling interval varies: 5 seconds for Trinidad and 1 second for RichFaces. And it's good to point out the tradeoff between responsiveness and use of network bandwidth. A fun way to illustrate this is to put either of the polling examples up via port 8079, and then let the sniffer window show while discussing the performance issues – students will notice the sort of unnerving sight of a full HTTP request and response going back and forth every 2 seconds – even after the user has finished and is just sitting there. (Try the RichFaces version for maximum effect – the request and response sizes just happen to be a lot larger.)

Chapter 8

This quick chapter is partly meant as an opportunity to wrap up the major threads of the course – servlet/JSP, RMI, and JSF strategies for Ajax – and to talk about where Ajax is headed. The not-too-scientific metrics that occupy most of the chapter should make for some interesting discussion, but ultimately the distinction between these different strategies defies quantification. As Ajax technology continues to evolve, there will probably be more to say here, and eventually we should start to see standardization, at least for Java EE.





Revision History

Version 5.0 is the initial release of the course, covering Java EE 5 practice.





Troubleshooting

If you run into any trouble with code exercises, the first and best thing to do is to double-check that the classroom machines have been set up precisely according to the course setup guide. Especially, the wrong version of a tool can cause significant problems; don't wander off-book in this way unless absolutely sure you can support the software that you prefer and that we haven't tested. Check environment variable settings carefully, too; these are the cause of a great many classroom glitches.

Below are some specific pitfalls that have come up in previous offerings of the course:

- Be certain of the **CC_MODULE**, **JAVA_HOME**, and **PATH** environment variables – and that the student has the same settings for Windows as a whole and for any open DOS boxes. A common mistake is to set these variables correctly in one console, open a second console by running **cmd.exe**, and expecting that the environment will carry over. Encourage students either to set the variables via the control panel (and then open all new DOS consoles) or set them in a console and use the **start** command to spawn other consoles with the same environment.
- RichFaces includes the CGLIB code-generation utility. This tool is known to cause memory leaks in the PermGen memory space (which is not garbage-collected, because it's supposed to hold only class information, which was expected to be stable, before all these code generation and AOP tools came along). The problem gets worse with repeated deployments of RichFaces applications, as each new deployment generates more classes. Eventually Tomcat will slow down considerably, and then crash outright with a memory error. Restarting the server is a simple fix, even if it's a little irritating to have to do it.
- In Lab 7C, if things don't work, try creating a **try/catch** block against any **RuntimeException**. In case of exception, print a stack trace and re-throw the exception. This may seem pointless, but otherwise the RichFaces event dispatcher will gobble up exceptions without logging them. So this can shake out problems that weren't appearing before.





Errata

At this time there are no errata for the course.





Feedback

We very much appreciate whatever feedback we can get on our courseware – especially from the instructor's perspective. Naturally, the more specific, the better, and we strongly encourage you to make notes on issues you may encounter in the classroom, whether they're typos, missing files, or suggestions for clearer language to explain a concept. We can't guarantee that we'll act on every suggestion, but we're aggressive about stamping out problems and try to be highly responsive. Hopefully this means that when you give us good feedback, you get a better course the next time you need to teach it.

Please direct all courseware feedback to

Will Provost
Capstone Courseware
<mailto:provost@capcourse.com>
877-227-2477

For anyone who's interested, we have a very informal defect-tracking system, based in Excel spreadsheets with columns to capture defect location, nature, status, and author feedback. Ultimately, feedback goes into these sheets, so if you want a template, we'll be happy to provide one, to facilitate the reporting process.

