



Capstone Courseware, LLC

33 Boylston Street
Jamaica Plain, MA 02130

877-227-2477
capstonecourseware.com

JavaServer Faces

Version 1.2.3

Instructor's Guide



Overview

JavaServer Faces seems to have “arrived” with the 1.2 release and implementation. The basic ideas of JSF have a natural appeal: treating the page as a system of GUI components, aligning those components with backing beans, component-scoped conversion and validation, etc. Mostly the software industry seems to have been waiting for the technology to mature, with the MyFaces implementation of JSF 1.1 drawing the most interest but most shops either completely unaware of JSF or holding off and using Struts instead. Though there are a few bugs and limitations, the RI is quite solid, and Sun especially is giving it a lot of TLC so we can expect continuing improvement.

You should find JSF to be a fun technology to teach, which is a tribute to JSF itself and to its conceptual simplicity. As with all our courses, in this one we try to peel back the skin a bit and see some of the inner workings; but with JSF (more than with Struts, more than with Spring MVC), we could probably do pretty well without knowing much about the underpinnings. Students should find the idea of custom tags for components and their connectivity to model properties to be intuitive, and the course is able to move from Hello-world-style examples to more interesting pages and flows very quickly.

Also, and again more than for most courses, in this one we've been able to sort out the various moving parts of JSF and present it in a mostly linear fashion, progressing from lifecycle and navigation through view-building with components, backing beans, events, converters, and validators. Granted, it's Chapter 5 on events before students are building what I would consider complete web applications, with view, model, and controller all in place; but as early as Chapter 3 they are doing real and interesting JSF work, just with some of the pieces of case studies yet to be built. There shouldn't be much need for “stiff-arming” students who want to jump ahead to see the whole picture, because the early pieces that they're seeing are interesting and rewarding in themselves.





Timeline

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

Day 1

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

Day 2

2 hours	Chapter 4
3½ hours	Chapter 5
1½ hours	Chapter 6 - partial

Day 3

1½ hours	Chapter 6 - completion
3 hours	Chapter 7
1½ hours	Chapter 8

In no case do we recommend skipping any of the chapters except the last one. This course does indeed build linearly, and any early chapter missed will leave some confusion in later discussion and exercises. The only possible, partial exception would be Chapter 2, in which either lifecycle or navigation could be skimmed to save some time – but don't skip either.





Tools Deployed with the Lab Software

This course's software requires separate setups of the 5.0 JDK, the Crimson text editor, 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**:

- **JSF 1.2** reference implementation, in **JSF1.2**. Here you'll find the standard JSF 1.2 RI deployment, plus the 1.2 configuration schema. The schema and older DTDs are with the JAR files in the **lib** directory; the API docs are in **javadocs**. The JSF sample applications are here, too, if you want to take a side trip into showing any of those.
- **JSTL 1.1**, in **JSTL1.1**. The JARs and TLDs for JSTL tags used in many of the course-exercise JSPs are found here.
- **Tomcat 6.0**, in **Tomcat6.0**. This is our target web server for deployment and testing. As mentioned in the coursebook, we need Tomcat 6 to support JSF 1.2 because JSF 1.2 uses the Java-EE-5 Unified EL, and earlier versions of Tomcat don't support that. For this initial release of the course we're using the beta because it's the latest available; we'll migrate to the final when it comes out but we've seen absolutely no issues with the beta in developing the course.
- **Crimson**, a freeware text editor, in **Crimson3.70**. This is made available so that we're sure students have a decent code editor on hand, though they can certainly use their own favorite text editor or IDE. Also, Crimson is primed with support from a second tool, the **XML Validator**, to make a nice little XML editor for working with the **faces-config.xml** file. Hit F9 to save the current file and run an XML parse to check syntax; hit F10 to save and validate against the JSF config schema (though this requires either Internet access or a change to the schema location at the top of the file). Note the mention in the Troubleshooting section about diagnosing and fixing configuration-file syntax and validity errors.





Ant Build Process

We use **ant** for all our builds. 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\JSF\Ant**. Information here defines a routine for building a JSF 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** and **faces-config.xml**. Then the build will draw in JSF 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 it's 30 seconds to sweep for changes and re-install before testing from a browser.





Eclipse Overlays

Capstone Courseware 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 before using the overlay package in the classroom. 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 instead of the text editor and command-line tools that are standard for the course. See the file `c:\Capstone\JSF\Eclipse\ReadMe.html` for general notes on how to use the Eclipse overlays for Capstone courses. Be prepared to walk students through the first few exercises in Eclipse; the notes in this file are for experienced Eclipse users, and will not be clear to many students on their own.

The most important thing to remember is that you must start Eclipse itself with the environment described in the coursebook: the `CC_MODULE`, `JAVA_HOME`, and `PATH` variables all matter, because Eclipse will spawn Ant and Tomcat processes. Ant will need the right version of Java, and Tomcat will need `CC_MODULE` as a base location for data files. The problems created by building and testing projects with the wrong settings are limited to those projects – but they are also quite hard to repair.

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.



Teaching Notes

Chapter 1

The opening part of this chapter is more or less in the tradition of our other courseware, but with a slight twist. Usually we open with some history and motivation, as in – this is what was there before, how it came about, and why it's good but not good enough, and so here are the problems that the new technology will solve. This is a little different in that we're not trying to set up other alternatives as inferior and needing JSF's help: rather we want to focus on three pre-existing technology areas that, in their own design choices, will shed light on JSF, why it is the way it is. This is why many of the page titles start with "Perspectives" – it's just that, a perspective from which we can understand JSF better, and not so much a past failure or as if JSF is built on the ashes of Servlets and Struts. You'll probably get the most mileage out of the interesting parallels between JFC and JSF, and this may form a touchstone for several later chapters as we see JSF increasingly as rooted in the UI component tree.

One question over which we intentionally glide is this: are JSF backing beans really model components? This sort of question arises often in MVC web-app theory, but it may be exposed more obviously in JSF. Do we wire components to the properties of business beans directly? Or do we use a layer of backing beans as presentation-tier components, and then pass values from there to the business tier? Various solutions are possible and there's plenty of debate about (a) which approach is better and (b) what to call it! Engage in this with students as much or as little as you like. Frankly, from the perspective of JSF and this course, it's more or less a moot point: either the beans are business beans or they're adapters or mediators to the business tier, and one way or the other it doesn't really affect any decisions we make in building our applications. See Chapter 4 for a little more on this.





Chapter 2

Having seen a very simple, but reasonably complete, web application demonstrated in the overview chapter, we quickly move to the nitty-gritty of the JSF lifecycle. While this isn't a topic that's going to bring a bunch of new features into a case study right away, it's an important conceptual foundation. It's also not very hard to learn, and the tracing demo should keep everyone's attention. The second part of the chapter on page navigation should flow nicely out of lifecycle, because to understand JSF navigation one must grasp the stateful nature of the JSF view – that we have to pass through JSF on our way to any JSP, even on the initial request to the application.

You may find that one of the few “stiff-arms” you need to raise occurs in this chapter, as sharper students will start craving details about components and beans. Do try to defer discussion to the next two chapters so that you can move through them in an orderly way.

Chapter 3

If JSF is an MVC web framework, then this chapter is about the View. We stay completely focused on view-building: UI components, tree relationships, Java class vs. the JSP custom tags that incarnate them, etc. The model of UI controls will be simple for most students – only the selection components might take some extra illustration if people in the class don't know HTML forms very well. More subtle is the mechanism by which the tree is first created, then stored prior to releasing the response; reconstituted on a later request and re-connected to the client side via HTTP request parameters. It's quite elegant – so much so that for most purposes we don't need to think about it much, but it will be good to get it across that this link from HTML `<input>` to JSF component object is not magic.



Chapter 4

This chapter treats the MVC Model to back the JSF View. See earlier comments about the question of whether backing beans are the model or a means to contact the true model. Again, mostly this is moot for our purposes, but one point probably does bear some discussion, and that's the impact of this choice on the use of JSF managed beans. That is, if we do commit to the idea that backing beans are model beans, a.k.a. business beans, then it may not be JSF's proper responsibility to create them. Shouldn't business-tier objects be incarnated in some other way – Spring IoC beans, Hibernate-mapped objects, even a JAXB-generated object model? Meanwhile the JSF IoC system (managed beans) is good but not great, Spring IoC being clearly superior in flexibility and power. A tool stack involving JSF, Spring, and Hibernate is an attractive idea – this may be getting way beyond your students, but more advanced folks will already be asking questions along these lines.

One thing purposefully left out of the coursebook is any real emphasis on the **binding** attribute as available on many custom tags. Again, with an advanced or inquisitive group, this is probably the best place to bring that up, chalk-talk it a bit.

Chapter 5

So finally the third of three MVC chapters: it doesn't say so in the title any more than the other two did, but this one's about Controllers. The JSF View interpretation is fundamentally different from the ones in other MVC frameworks, and the Controller is, too: take every opportunity to hammer home the distinction between a request-driven framework like Struts or Spring and an event-driven one like JSF.

The early diagram of the Observer pattern is not followed by a UML diagram of the Java/JSF event model – this was left out largely to allow you to develop it interactively in class. Most students will already know about Java events, so it makes for a good Q&A session to keep students involved. Annotate the diagram in the book with answers from students: which is the Subject, which is the Observer? What's the analogue to attach and detach methods? Are there methods like update and notify, or what does Java do differently?

It's also worth a quick mention, after seeing the JSF-standard events working, that application code can fire events, too: UI components, controllers, etc., can plug into the event-queueing framework as producers, not just consumers.





Chapter 6

As of Chapter 5 we have a complete view of JSF-as-MVC and of the JSF request/response cycle. We can define views, accept input, produce output, and trigger application actions. This and the next chapter offer refinements to that basic framework; it's as if we've climbed the hill and are exploring on the plateau now.

Converters are pretty simple to work with and to implement, and you should find the progression from basic concepts to using standard converters to building our own to be straightforward. A few notes not included in the coursebook: for one thing, the JSF RI seems to be non-compliant in not recognizing a converter registered for the String class. How strong a case there is for ever registering a custom String converter I don't know, but it is explicitly required by the JSF spec and seems absent as a feature in the RI.

Another note is that the full name of the "boxing type" should be used when registering converters for that type or of the corresponding primitive. That is, properties of type **double** and **java.lang.Double** will both be covered automatically by a converter registered for the latter.

It might be worth emphasizing, depending on the group, that converters are not the right way to deal with getting the right presentation strings into an HTML `<select>/<option>` tree. At first it may seem that way, but here there are really three values: the backing-bean property, the UI component value, and the UI component label. It's the label that shows; the value is a client-side but still programmatic thing, and converters only deal with values. The bit about dealing with enumerations should make this point clearly but it's applicable to other data types and to things like lookup tables in databases.





Chapter 7

Another chapter that brings some refinements to the system, now focusing on validation. This chapter progresses in a similar way to the previous one: concept, standard tools, custom tools; but we also have to talk about managing error messages, and there are more variations on the theme of creating a custom validator than there are with converters. This chapter also loads up more on labs, partly because there's more to chew on and partly because we've gotten to the point at which we might want to take some time for more exercises on JSF in general. Doing all four labs in this chapter is perfectly fine if you're not pressed for time; each one really does add a new tool to the student's toolbox.

Students may ask, and for the sake of brevity we don't say in the book, but `<h:messages>` will produce the summary message and `<h:message>` will produce the detail. Either one is configurable via attributes: see the TLD docs for specifics.

One oddity: this isn't specified but, in the RI at least, both of the phase-listening methods must be registered with the `<f:view>` component – even if you only need one of them. Try taking the **before** attribute away – suddenly neither method gets a call.

Chapter 8

This final chapter on data tables is entirely optional, but there should be time for it. Even if students don't have strong interest in that particular component, it makes for a good example of what JSF components can do that goes beyond the single-value, single-HTML-tag mapping, and it provides a natural jumping-off point for discussions of custom component development, component libraries (including Tomahawk), and Ajax.





Revision History

Version 1.2.3 is a maintenance release, with fixes for a few typos and minor code problems.

Version 1.2.2 is the initial release of the course.





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:

- Watch the Tomcat console for installation of a newly-deployed application. Tomcat looks for updates every 30 seconds by default. It is possible to jump the gun in testing a code change, by running **ant** and immediately opening a browser and requesting the application's entry-point URL. It's a good idea to keep the bottom few lines of the Tomcat console visible, and to wait for output there that confirms undeploy/redeploy of the application, before testing. Otherwise, you'll get either a "resource-not-available" message or, worse, you'll successfully request a page from the old version of the application before it's un-installed; this can get quite confusing.
- If you see a SEVERE error in Tomcat after deploying an application, but little explanation of why it won't install (things like aborting "due to prior errors" that aren't shown) ... start by looking at the **faces-config.xml**. Any ill-formedness or invalidity in this file will produce the symptoms just mentioned. Use the Crimson editor and its pre-configured hotkeys to parse (F9) and validate (F10) the file, and you'll most likely see a simple syntax error.
- When you see error messages in the Tomcat console or the browser of the form "Property {1} is not writeable on bean X", know two things. First, "{1}" is an ordinal value, but not related to the bean in question – it's not the first property on the bean or something. It's an unresolved message key – the property name isn't supplied, probably because the error is due to some more fundamental problem. Second, this message usually means there's no matching mutator on the target bean – which is sort of what it says, but it's a little misleading because X will be the type of the property that it can't set – not the type of the target bean that has the property. This one has wasted a fair amount of the course author's time – don't be misled!





Errata

The following observations and issues have arisen since the latest release. These will be addressed in the next release of the course.

- When starting Tomcat with a JSF application installed (or on the first deployment of a JSF application to the running Tomcat server), you will see the error message “Error Instantiating ExpressionFactory” and an exception trace regarding a failure to load class **com.sun.el.ExpressionFactoryImpl**. This is a side effect arising from the use of the Sun JSF 1.2 RI outside of the Java EE SDK server: the JSF code is looking for an expression factory found in the SDK server’s main JAR. The error is benign (perhaps even spurious), since Tomcat 6 provides a different EL resolver. In future releases we may try to whittle down the JSF RI JARs, or hopefully find truly portable releases of them from Sun; these had to be ripped out of the SDK server download as the best available RI when this course was released.





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.

