

# **Developing Java Web Services**

**Version 2.1**

**Instructor's Guide**

## Overview

This course begins with an intentionally broad definition of a web service, and the first chapter expands on the idea that many sorts of software might legitimately claim to be web services. In the Java world there is a corresponding diffusion of technology: adding “web services” to J2EE for the 1.4 release brought four new required standards (JAX-RPC, SAAJ, JAXR, WS4J2EE) and one optional one (JAXM); and there’s been the sneaking suspicion of a scattershot approach by those in charge of the platform specs.

Java EE 5 finds significantly sharper focus, and this course follows suit by putting most of its time into JAX-WS. This standard plays the “kingpin” role that belonged to JAX-RPC before, but more than being only the most attractive of a handful of approaches, JAX-WS succeeds (in my opinion) in unifying the many possibly strategies for web-service development under one consistent programming model. (As the coursebook describes, when we say JAX-WS we probably mean JAX-WS plus WS4JEE plus WSMetadata; but still the core JAX-WS specification draws in SAAJ, JAXB, and other XML technology, via the Provider/Dispatch APIs.)

Where the previous version of this course laid out the more nuts-and-bolts approach to web services development represented by JAXM, before proceeding to the JAX-RPC approach, this course moves more aggressively to get typical JAX-WS practice in front of students as early as is manageable. Even so, there’s a lot to talk about before we can start building JAX-WS services and really know what we’re doing. A quick example of JAX-WS development comes in Chapter 2, but then we pause for several chapters to consider SOAP, JAXB, and WSDL, before getting into JAX-WS in earnest. Then, students should get as much JAX-WS as they can keep down, with five straight chapters forming a mini-module in the center of the coursebook and five more covering alternatives and advanced topics under JAX-WS.

The area of web services – even after years in the public eye – also lends itself to potentially endless speculation about what specs and tools and visions are going to “win” eventually. An admitted sucker for these sorts of discussions, I’ve nonetheless tried to maintain a healthy skepticism about what topics, techniques, and insights about WS development are truly valuable to the developer with an application to build, and to spend less time on the soapbox and instead to cut to the chase: what works, what doesn’t, what techniques are interesting to what sort of company and for what reasons.

This course also puts a little less “web” in “web services” than did its predecessor. We’re no longer driving the study of web services as primarily expansions or refactorings of existing web applications. Even though web services and web applications often exist side-by-side or delegate to each other, the web service is fundamentally a middle-tier component, and as such we treat it a bit more like an EJB, with a few web clients thrown in to show the variety of client types, but mostly sticking with command-line clients, or no clients at all except for the SOAPPad utility.

## Timeline

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

### Day 1

2 hours Chapter 1

2 hours Chapter 2

2 hours Chapter 3

(Get a start on Chapter 4)

### Day 2

3 hours Chapter 4

2½ hours Chapter 5

1 hour Chapter 6

### Day 3

3½ hours Chapter 7

2 hours Chapter 8

2 hours Chapter 9 (spills into next day)

### Day 4

2 hours Chapter 10

2½ hours Chapter 11

3 hours Chapter 12 (spills into next day)

### Day 5

2 hours Chapter 13

1½ hours Chapter 14

1½ hours Chapter 15

If your student group isn't very strong in Java to start with or for other reasons isn't moving at the necessary pace over the first couple days, consider any or all of the final four chapters to be optional, depending of course on interests expressed by the students. To leave with a strong sense of what Java EE web-services development is and how to go about it, students should cover the first 10 chapters in depth, and Chapter 11 is a significant bonus. The remaining material is all valuable, but generally speaking none of it is essential to building web services.

(If students know that handlers will be a part of their work, then both Chapter 12 and 13 become required reading as well; you might even consider putting them ahead of Chapter 11 to be safe.)

## Tools Deployed with the Lab Software

This course's software requires separate setups of a Java EE SDK and the Crimson text editor; otherwise it is self-contained, as the lab installer sets up not only the lab software but also the MySQL relational database manager, which is used in a few of the case studies in the course.

The deployment in **c:\Capstone\Tools\MySQL5.0** is not the full MySQL 5.0 distribution, but a tiny kernel that's enough for our purposes: basically the **mysqld** server, the **mysqladmin** utility, the **mysql** terminal, and supporting libraries and administrative databases. We use MySQL as our RDBMS for all applications that use JPA entities. The instructions for startup and shutdown of MySQL should be sufficient for classroom use; the only hangup that's at all common in the classroom is a port conflict and on dedicated training PCs this should not happen at all.

## Ant Build Process

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\JavaWS\Ant**. Most applications follow one of two approaches to build a JAX-WS service in a deployable EAR, but especially in early chapters there's more of a mix-and-match of a few key targets:

- **compile**, which builds Java source code in **src** to **build/classes**
- **WAR**, **EJB**, and **EAR**, which build various Java EE archive formats
- **WSDL-to-Java**, which runs the **wsimport** tool
- **Java-to-WSDL**, which runs the **wsgen** tool
- **XML-to-Java**, which runs **xjc**
- **Java-to-XML**, which runs **schemagen**
- **deploy**, which deploys the completed EAR
- **run**, which launches a predefined application class
- **init-DB**, which builds a MySQL database and configures JDBC resources in AS9

Those familiar with Ant will get the picture pretty quickly by reviewing two files:

- **c:/Capstone/JavaWS/Ant/JavaTargets.xml**
- **c:/Capstone/JavaWS/Ant/GlassFish2.1Targets.xml**

## Eclipse Overlays

Capstone Courseware provides an optional package of workspace and project files for Eclipse Galileo 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. Also, Eclipse doesn't automatically understand that JAX-WS builds include code generation, and so, on its own, it would produce a great number of false errors in the Java code that it can see, failing to resolve references to Java code that it can't see. We smooth this over considerably by configuring our projects with "builders" that trigger code generation via Ant prior to Eclipse's own Java compilation.

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 documentation inside the workspace itself for general notes on usage: when you first open the workspace you'll see a **ReadMe.txt** that directs you to deeper, HTML-based documentation. 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. See also the file **JavaWS Module Notes.html** (available in the IDE as well) for specific matters related to using Eclipse with this course.

Even if you've used Eclipse with other Capstone courses, you should take some time to get familiar with this one; and your students will doubtless need a little coaching. The basic process, say for an example that has a WSDL-to-Java service and a client project, is: **F9** to build the service; run the **deploy.bat** to deploy to the server; **F9** to build the client, and **F10** on the client's application class to test. (In fact it's not necessary to build the service in Eclipse for **deploy.bat** to work, either, but if there's any service-side coding involved you must build the project before Eclipse will open the source files without showing false errors.) Eclipse can be quirky, and this workspace is no exception, but all projects have been tested and do run cleanly, sometimes with a little coaxing.

Running SOAPSniffer in the IDE is a nice trick, too. There is a launcher already configured to run **SOAPSniffer.java** with arguments "8079 8080"; you can leave that console running and even pin it around the workspace to watch traffic while other applications run. (There's a launcher already set for SOAPPad, too, though this is a bit less compelling; it's just a separate GUI so there's no real integration.)

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

As I mentioned in the overview, this chapter starts right out with an intentionally broad definition of the term “web service.” It could be narrower – as in WS-I BP – or broader – it’s been suggested to me that JSON requests from AJAX pages might be aided by “web services” and Microsoft’s terminology actually favors this interpretation. I’ve gone with this one because, for one thing, it provides a good perspective on what web services aren’t: web protocols distinguish them from DOC frameworks and programmatic use means they’re not traditional web applications. But this definition also comes pretty close to defining the scope of the course itself – at least it does if you add Java and take away REST. Most of what we do is WS-I-BP-conformant, but, as it says in this chapter, BP is more an organizing principle than a goal. And many of our examples would fit neatly in a rigorous SOA, but many would not – some are even more obvious candidates for REST.

Use this chapter to communicate that web services in general are our target for development – we’ll be writing software that produces and consumes SOAP at runtime, WSDL at build time – and to paint the bigger picture of web services, what they’re used for, how they fit with SOA, and so on. Of course this is also the time to find out what corner of that big picture your students occupy: they may be much more interested in some things than others. Plenty of groups want the soup-to-nuts treatment that is this coursebook, but plenty of others would rather skip some things and concentrate on others.

A few points that didn’t find their way into the coursebook but that you might want to add in your lectures – I’ll sprinkle these into the notes for several upcoming chapters as well:

- There is a whitepaper on how to register service WSDLs in a UDDI repository, and that serves as the most formal “glue” available on combining these technologies: find it at <http://uddi.org/bestpractices.html>.
- We leave security topics alone for the rest of the course, after the very brief and perhaps appetite-whetting overview at the end of this chapter. Capstone Courseware offers a separate, three-day course in “Securing Java Web Services.” If your students have a strong interest in this topic, please contact us for supplemental materials: we can send you chapter presentation PDFs and point you to lab software, so that you can do an informal presentation.

## Chapter 2

Now we put our Java glasses on and talk about how the Java EE platform facilitates development of web services. Surveying the Java-EE+web-services landscape has gotten a lot simpler since EE 5, because JAX-WS and WS4JEE really unify the programming model, so it's not such an either/or choice as it was with JAX-RPC and JAXM/SAAJ. So this chapter isn't meant to catalog every relevant API, but rather to focus on JAX-WS and to speak peripherally about JAXB, SAAJ, and then other interesting technology that might be involved, such as JAXP. If you find strong interest in the more "manual" approach, forward-reference Chapters 11 and 12, and probably put a priority on covering those in depth. Chapter 11 lays out the Provider/Dispatch APIs, which are one of the unifying forces in JAX-WS, and also gives a bit more background on JAXP and how it helps to glue different XML representations together.

One clarification: in the first example, we go to some lengths to explain the process by which the container understands how to host a component as a web service, and we talk about it as if it's happening on demand, when the request comes in. Students may wonder if this is actually the timing, and it isn't: the container will ask most of these questions in advance and index things so that it can quickly route HTTP requests through JAX-WS to the appropriate SIB.

## Chapter 3

For the first few chapters of the course we continue to toggle between platform-independent and Java-specific topics: here we return to the interoperable arena with a look at SOAP itself. The goal here is for students to get familiar with the language so that they can understand the workings of services the rest of the way through the course, and so that they can diagnose problems in their own services after class. We go so far as to have students hand-write SOAP messages, but it's worth emphasizing, prior to these labs, that this is the last of this they'll be doing: their JAX-WS software will do it for them going forward.

For most of us, the SOAP header system is the most interesting and challenging part of the language: not just the XML content model, but the behavior of originators, endpoints, and intermediaries in SOAP processing. So you may want to take some extra time there, possibly expanding on the diagram of headers being added and removed along a messaging route.

## Chapter 4

JAXB, though obviously important to JAX-WS services, is a hard topic to place in the flow of this course. I chose to put it here because prior to SOAP didn't make a lot of sense, and after WSDL I really want to roll right into JAX-WS and stay there for a while. Even so, it may seem like an odd transition, from SOAP to JAXB to WSDL. The best explanation for this is simply that we need all of these things as underpinnings of our primary topic of study, which is JAX-WS; and that if SOAP is about wrapping arbitrary XML content in its body, JAXB alone can manage the body content, given a schema. Whether we let JAXB do that more automatically (performing data binding en route to a JAX-WS SIB) or manually (through a JAXB Provider) will be an interesting choice to make a little farther down the line.

Some additional points of interest:

- The book mentions that, in the Java-to-XML case, the **ObjectFactory** must be written by hand. This isn't absolutely necessary; we could devise either a JAXB tool that would produce the object factory class as part of schema generation, or a build process by which we'd use **schemagen**, then **xjc**, to create the factory class and copy it into our distribution so that it would be available at runtime.
- There is an additional variant of the Housing example – **Examples/Housing/JAXB/Default** – that shows what the schema would be like with no Java annotation at all. This might be an interesting comparison, though for reasons of space and time we left it out of the formal chapter.

## Chapter 5

Now we move back to the interop space and study WSDL in depth. Though both this and the SOAP chapter cover their targets thoroughly and have hands-on exercises in writing and reading them, the motivation is different. Here, we are developing skills in WSDL authoring for real, practical development purposes, and not just for background familiarity or so that we can understand what's going on while debugging. Part of the philosophy developed over the next five chapters on JAX-WS is that it is well worth getting to know WSDL and to embrace it as part of the source code for an application – rather than treating it more cheaply as a means of translating our service model into a non-Java language, as is more the attitude when developing Java-to-WSDL.

Some additional talking points:

- RE multiple-port services: you might note that most JAX-RPC tools fail to serve multiple-port services correctly, for one reason or another, usually having to do with clumsy name mappings. This includes the J2EE 1.4 SDK; but the Java EE 5 SDK handles multiple ports gracefully. You will see this in practice in the **Prototype** example in the next chapter.
- The **namespace** attribute in a `<soap:body>` is generally not well understood, and it's one of those things that often gets overlooked as students are a bit overwhelmed with language details already. You might want to take a little extra time to highlight this and explain exactly what it means. Note that the WSDL's own **targetNamespace** governs the names of abstract-model components and concrete-model components, but never involves anything related to a particular protocol; then, the **targetNamespace** of any schema that's involved would govern the namespaces used in all document-style elements and of everything under the operation element in RPC style. That leaves a gap, and this attribute simply fills the gap. Try varying this value in a working service, and then notice the impact on SOAP messages that work with the service.
- The **Dynamic** client should be an interesting illustration of the power of metadata; but it's worth noting the simplifying assumptions that go into it. The target service must be RPC-style, use only single-value parts, and mustn't rely on any imported WSDL descriptors. These are all limitations of this piece of software, and not of WSDL itself.
- There is a library **WSDL4J**, from Apache, that provides a nice intuitive object model for WSDL, much as SAAJ does for SOAP. The **Dynamic** client could be rewritten to use this library, instead of doing all its work via DOM processing. (In fact a version like this was written; it got more aggressive about parsing all imported schema, and then ran afoul of the weird URL rewriting being done by AS9. But the code was a lot smaller!)

## Chapter 6

This chapter begins a five-chapter mini-module within the course, devoted to JAX-WS. Here, we introduce the API and programming model in general; the next three chapters each focus on a different development task; and Chapter 10 provides a wrap-up and raises some intermediate and advanced issues.

As to the initial discussion (and criticism) of the three closely-related specifications: in fairness to the expert groups that wrote these specs, each one does have a clear and distinct mission: JAX-WS is the basic API; WSMetadata is working at a higher level, and with an ease-of-use focus, over the existing JAX-WS capabilities; and WS4JEE develops the metadata, assembly, programming and deployment models of which JAX-WS is a major part. Personally I guess my gripe is not so much that it's confusing how these specs are separated; it's that they were separated at all.

The two examples focus on code generation, which is why the services in question are highly abstract. The idea is for each named element in a source file to be easy to track on the other side of the code-generation process; in class I like to compare this to tagging birds and observing their migration. And we're working our way gradually into a deep understanding of how JAX-WS works: in Chapter 2 we saw a working service but kept our study to a high-level view of the main moving parts and the deployment model; here we get more into the mechanics of code generation but hold off on full study of WSDL/Java mappings and other development particulars; and in the next three chapters we finish the process with detailed study of mappings, and a number of hands-on exercises.

## Chapter 7

This isn't necessarily the most important chapter in the course, but it is the one that provides the most direct hands-on exercise in the most common JAX-WS development process, which is that of building a service based on a WSDL descriptor. This chapter has three labs, and while one is marked optional, if you are not pressed for time at this stage of your class, I do recommend running all three labs to give students lots of work in this area so they come away with confidence that they can build more services of their own.

Each time a new tool has been introduced to the build, we've studied it and run it by hand: **xjc** and **schemagen** in Chapter 4, **wsimport** and **wsgen** in Chapter 6. Still, students may be curious about how these tools are working in the Ant builds – which are a convenience of course but can have the effect of hiding these details and making it all seem like magic. In this and each of the next two chapters, you might want to choose an example or a lab to review with an eye to how the build works: read the asant output in detail and point out where various files are appearing in the build tree, or even walk students through the same process using all manual commands from DOS, if there's a real appetite for this. For this chapter, consider the Tracking example as completed in the first lab: show how **wsimport** precedes **javac** and how the WSDL and WS4JEE metadata is assembled into the final EAR along with everything else.

## Chapter 8

This short chapter lets students get familiar with the client side of JAX-WS. There's not much more to talk about, because the service and client sides are surprisingly similar under JAX-WS, in terms of what tools they use, code they generate, and how they use that code. The big difference is the proxy factory created just for clients, and so we talk about that a little before jumping into some straightforward lab work.

A review of this chapter's lab might highlight the use of **wsimport** for the client side, and also how when a converter is involved, the sequence is actually **javac** (for converters only), then **wsimport**, and then **javac** again (for everyone else).

A talking point:

- Many servers, Sun AS9 included, generate web pages designed to let the user test a deployed service. These of course are based on the WSDL and are a lot like the **Dynamic** client from Chapter 5. In the Sun server, you find these by logging into the admin console and then drilling down to the deployed application and component.

## Chapter 9

Now we hop back to the service side and consider the alternative of Java-to-WSDL development. It is an implicit endorsement of the WSDL-to-Java approach that we've waited this long to talk about Java-to-WSDL in any depth – and after this chapter come some more explicit arguments to the effect that Java-to-WSDL is an inferior approach for most purposes. Students may draw you into discussions of “Which Way to Go?” even before the section that starts with that page title in the next chapter, and you'll have to play it by ear as to when it's really best to jump into this. I've had groups that were unified on the goodness of WSDL-first, groups that leaned the other way, and plenty of mixed groups. So it can be a fun discussion or it can be a short and matter-of-fact one.

In this chapter either the Love Is Blind demo project or the Housing lab would make good subjects for a review of the build process: of course now **javac** runs and then **wsgen** cranks out the WSDL, and most everything else is symmetrical.

One interesting talking point in this chapter:

- For years there's been this tension between the requirements of first the JAX-RPC and now the JAX-WS specs that method parameter names are to be mapped to part and child-element names in the WSDL and/or schema – and the failure of almost all tools to do this. On one hand we can see from the standards perspective that this is the mapping we'd like to have; from where the implementers stand, however, there's a fatal flaw in the requirement, which is that Java .class files don't even carry this information – so how is a tool to generate from those .class files information they don't have in them to start with? You may also note that WebSphere's JAX-RPC implementation goes the extra mile and does meet the spec requirement, and it does this by parsing the source file rather than the class file. This is JAX-WS, not JAX-RPC, that we're studying, but still it's an interesting proof of concept.

## Chapter 10

There are really three sections within this chapter: which way to go, polymorphism, and services as EE components. As mentioned in the notes on the previous chapter, the first of these sections may bring a lively discussion or not much of one at all; you have to see how things develop, but be ready to rumble!

As to polymorphism, there's an ongoing argument here: SOA theory generally dismisses polymorphism and even urges against it: SOA is not OO, and there's a sense that polymorphism is generally unneeded and if anything can make service semantics unnecessarily confusing. Yet JAX-WS 2.1 exhibits full polymorphism, and there's the limited polymorphism option that's popped up here and there as well. The primary point here is just that it's good to understand what to expect in this regard, and that's the aim of the "Gimme" example, and since there are no formal specifications on this we really have to rely on empirical research such as this. It's also fair to flag this as a potential interoperability issue, plain and simple.

Most of the chapter is about component services and how a JAX-WS SIB can do some of the things we're used to doing in servlets, EJBs, etc. For experienced EE developers the need should be evident; for SE developers with no EE experience the concepts of dependency injection, component environment, and resource lookups may be a bit more foreign. The techniques for finding resources, parameterizing the service, and so on are harder than they should be, but not hard to get across.

One additional talking point:

- We talk about scenarios of service developers writing the WSDL (or generating it), clients or third parties writing it and letting service developers generate Java from it, etc. What about a situation where both the WSDL and the Java classes already exist? Maybe we have existing application classes but have decided to work WSDL-to-Java for other reasons – what to do? Remind students that no one requires that they run a code generator! If the WSDL and Java classes are in synch (perhaps with the help of a few well-placed annotations), they can be assembled and deployed and the server doesn't care how they got there.

## Chapter 11

This chapter moves us beyond basic JAX-WS development, and also fulfills the promise made very early in the course that one can use JAX-WS to implement services and clients in a range of other APIs, including JAXB, JAXP, and SAAJ. We leave SAAJ for the next chapter, and we've already studied JAXB. That leaves JAXP as neither a prerequisite nor a topic of direct study, but still something that we need for our code examples in this chapter. There's not much to do about this – if students have done any JAXP coding they'll get this material on one level, and if they haven't they'll still get it but it will feel a bit more like a survey. We do talk briefly about Sources and Results as conduits. See how your class reacts to the material, and tune your presentation accordingly.

## Chapter 12

In this chapter we dig into SAAJ in depth. It may well be a bit much for students with a greater interest in the JAX-WS/JAXB binding-based approach, and it is one of the drier chapters, as we're really studying an API, interface by interface. For a little motivation, remind students that though this chapter's exercises are SAAJ-based services – which they may never want to do – the next chapter on handlers will make it clear that SAAJ skills are important, regardless.

Not mentioned in the coursebook is a completed SAAJ client application for `LoveIsBlind`; if you have extra time you might build **Examples/LoveIsBlind/SAAJ/Client** and demonstrate it after the second lab. (Don't confuse this with the **XOPClient** application, which is implemented in SAAJ but is intended for study of MTOM/XOP, as explained in Chapter 15.)

An additional point for this chapter:

- One of the most frustrating things about SAAJ parsing has been the need to test for the presence of non-**Element** nodes in the iterator returned by the **getChildElements** overload that takes no arguments. SAAJ 1.3 at least clarifies that this can happen, but it leaves us with a very clumsy algorithm for most cases. Oddly, the SAAJ 1.3 RI does not include whitespace-only text nodes in this iteration, where the 1.2 RI did. So in practice it would be possible to skip all the “instanceof **Element**” testing and just get all the elements, or the first one or only one or whatever. But since the spec says text nodes can appear, it's really still necessary to code defensively, in the style shown in the chapter and in SAAJ-based services in this course.

## Chapter 13

Handlers are a key piece of the JAX-WS architecture, but it's only at this stage of the course that we're really ready to "handle" them: we've needed to understand SOAP, WSDL, JAX-WS, and SAAJ first. The split between logical and protocol handlers is elegant and a nice reflection of the message/payload split in the Provider/Dispatch API – though the bit about shuffling a handler chain so that all the logical handlers come first takes some getting-used-to.

In the final demo on client-side handler chains, students may wonder if the proper solution wouldn't involve true localization based on the browser-supplied locale settings. They'd be right! We don't wander onto this territory because the resulting exercise would be a lot more about web applications and localization than JAX-WS and handlers, but the answer would probably be for the JSF bean to derive the locale information from the **FacesContext** and post it to the message context under some chosen property name; then the handler would always be configured, and would choose dynamically whether to "announce centimeters" based on the country code.

## Chapter 14

Somewhat like Chapter 11, this chapter will be more interesting to those who've already done some work with a peripheral technology, in this case EJB. For those with no prior experience in EJB, the chapter will probably feel more like an overview, and the labs may be a little bewildering, but still it's probably worthwhile to include it unless time is growing short. For most groups I would do Chapter 15 in favor of this one – in fact I keep the order as it is so that in either case we have a strong finish!

## Chapter 15

JAX-WS has improved dramatically on JAX-RPC in its ability to handle binary content with a minimum of fuss. Inline content is nearly trivial, unless you need to establish unusual Java bindings to the octet-stream XML content, and the final demo shows how easy it is to enable MTOM/XOP. It should make a nice "closing act" to throw that switch and compare the two response message from *LoveIsBlind*.

If students are interested in SAAJ support for attachments, you might explore the API docs in more detail on this: the model is quite simple although it requires a fair amount of code construction to manage the JAF DataHandlers, flavors, MIME headers, etc. Inline binary content via SAAJ is possible, but the API doesn't really offer any special facilities for it: it's more a matter of doing your own data binding by base-64 decoding the text node and then parsing the resulting byte array.

## Tomcat Add-On

We've had occasional requests for JAX-WS examples on Tomcat 6. While we've ported this course to other major application servers, including WebLogic and JBoss, a full port to Tomcat didn't seem to make sense, because so many features would be disabled.

Tomcat of course is only a web server, not an application server, and doesn't support many topics that we cover in the course: JSR-109 metadata, full dependency injection, EJBs, JMS integration, etc.

But basic JAX-WS does indeed work, and so we've put a little add-on module together. You can find the installer at the following URL:

**[http://capcourse.com/Courses/561/21/561\\_Lab\\_Windows\\_20\\_Tomcat6AddOn.zip](http://capcourse.com/Courses/561/21/561_Lab_Windows_20_Tomcat6AddOn.zip)**

You'll find one of our usual self-extracting installers there, along with a read-me file. Open the read-me and take it from there, but the basic idea is that this installer should be run after the lab installer for the course proper, and will lay in a Tomcat 6 image, the JAX-WS 2.0 reference implementation, and an alternate version of the Ellipsoid service that uses non-standard deployment metadata and a front-controller servlet provided by the RI. The read-me has full instructions on how to demonstrate the working service.

## Revision History

**Version 2.1** mostly keeps up with evolving tools: JAXB 2.1, JAX-WS 2.1, and the Java EE 5 SDK, Update 7 (meaning GlassFish 2.1). Beyond these updates and basic re-testing, and various small cleanups and fixes, here are significant changes:

- In Chapter 4: under JAXB 2.1, the custom converter for **java.util.Date** results in a binding directly to that type, and not to **JAXBElement<Date>** as before.
- In Chapter 7: we've removed the note about **wsimport.bat** needing to be fixed to sneak extra nodes into the classpath. It's still true, but now our builds use the **<WsImport>** Ant task, which doesn't suffer this limitation and is the recommended practice generally.
- In Chapter 7: the Series of Unfortunate Events demo has been modified slightly. To make the point more blatantly that the client's interpretation of the SOAP fault is entirely dependent on the WSDL and the incoming SOAP, we redirect the client to a couple of JSPs that produce the appropriate SOAP for a normal fault and an **IKnewItException**, and see how the client reacts.
- In Chapter 8: there is an additional third step to this same case study, which shows the **BraveAttempts.java** client explicitly catching **IKnewItExceptions**.
- In Chapter 10: JAX-WS 2.1 exhibits different behavior with regard to polymorphism, and the Gimme, Gimme example produces different results. See the Chapter 10 teaching notes for more on this.
- In Chapter 11, and later chapters: JAX-WS 2.1 requires a more complete WSDL when hosting a **@WebService Provider**: just service and port don't do it anymore. So many case studies have more complete (though in some cases still incomplete) WSDLs. This doesn't really show to students, but there it is.
- In Chapter 12: there was a bug pretty well hidden away in the **Examples/LoveIsBlind/SAAJ** tree. I say hidden because in the "normal" run of the course, taking all instructions as written, it wouldn't surface; but when it did it was a doozy. The SQL scripts in the SAAJ version were not the same as those in the JAX-WS version – a difference in how the enumerated type **Sex** was being mapped – but the entities were the same. So, if for any reason a student or instructor re-built the database while working on the SAAJ labs, everything would fall apart. This has been fixed.

**Version 2.0** re-invents the course for JAX-WS, JAXB, and other Java EE 5 standards. more has changed than has stayed the same, but some of the major differences are:

- More focus on WS-I Basic Profile in theory, and most services in the course are indeed WS-I BP conformant. Especially, we've gotten rid of SOAP Section 5 encoding completely (hurrah!).
- Stripped down approach to the first two overview chapters, trying (a) to get code in front of students sooner and (b) to present the simpler Java EE picture more simply.
- New JAXB Chapter 4.
- New JAX-WS Chapters 6-10.
- New Provider/Dispatch Chapter 11.
- New Chapter 13 on JAX-WS handlers, which share much of their mission with JAX-RPC ones but which work differently in configuration and at the API level.
- New Chapter 14 on EJB services, simplified and of course targeting EJB3.
- New Chapter 15 on handling binary content, including MTOM/XOP.
- Labs work to Java EE 5 SDK, the only other tool being MySQL for a few exercises that use databases.
- Some interesting examples of using JAXB overlapping with JPA.
- JAXM is gone.
- More focus on standalone clients and use of SOAPPad to test services, and therefore less on web applications, except as front ends to the web services (see the TrackingUI project in Chapter 10).

Instructors should plan for significant preparation time before teaching this version, even if they've taught earlier versions many times before.

**Revision 1.5.3** is a maintenance release with fixes for typos and other minor defects. This is the final revision for J2EE 1.4. We've also undertaken some broader updates:

- Labs have been reworked to deploy to the 1.4\_03 version of the J2EE SDK, which is current at the time of this release.
- Ant files have been centralized and are imported from individual **build.xml** files.
- The short name of the module, used in file paths, has been changed from **WSJava** to **JavaWS**. This lines up with Capstone's naming convention and especially with the follow-on course on Securing Java Web Services, which has the name **JavaWSSecurity**.
- The JAXM RI is now bundled with labs; no need to set it up separately.

**Revision 1.5.2** is a maintenance release with fixes for typos and other minor defects.

**Revision 1.5.1** is a maintenance release with fixes for typos and other minor defects.

**Revision 1.5** restructures the course from three separate modules into a single five-day course, while preserving the feasibility of shorter timelines based on subsets of the total chapter list. It also brings all the material up to the J2EE 1.4 final release and runs on the J2EE 1.4 SDK (aka AppServer 8).

**Revision 1.4** brings all the material up to the final drafts of the J2EE 1.4 specifications and runs on the J2EE reference implementation beta.

**Revision 1.0** is the initial public release.

## 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:

- The setup guide requires that the Java EE SDK be set up with an admin logon of **admin/ccstudent**. If any other username or password is set when the server is set up, the Ant builds will fail. You can adapt to the real admin logon by setting values in two files found in **c:/Capstone/JavaWS/Ant: GlassFish2.1.properties** has the admin username, and **GlassFish2.1.password** has the admin password.
- MySQL as deployed with the lab software rarely gives any trouble, but the one common killer is a port conflict. MySQL wants 3306 and if it can't have it it will fail to start. Try to be sure that nothing else has grabbed this port – including AS9, which can grab 3306 among many other dynamic assignments it makes as it runs. You may find that starting MySQL before GlassFish helps. Otherwise, you can configure a different listening port for MySQL using a **mysql.ini** file positioned in **c:/Capstone/Tools/MySQL5.0**; see the template files already in this directory such as **mysql-large.ini** for the syntax.
- In classrooms that are networked, you may run into a quirk of JMS on GlassFish, by which the server might be started on one IP address or name, but by the time the JMS client is running, the local machine has jumped on or off of a network or two and translates “localhost” differently, and then fails to connect. This can affect the Energy example, so if you get connection failures when running **readAMeter**, try re-starting the application server and testing again.

Separately, here are some common error conditions that come up when working with GlassFish and web services – these aren't so much troubleshooting issues with the classroom setup as tips on how to help students diagnose problems during their lab work:

- If you see an error message during the deployment phase of the build, to the effect that the server can't find `<impl-class>Service.wsdl`, when the actual WSDL has a different name entirely, know that this is a secondary error that indicates some sort of failure to parse the service metadata. Often it means that `webservices.xml` is out of synch with either the names and namespaces in the WSDL, or with the name of a Java SIB class or its `@WebService` annotation. The best advice is just to comb through `web.xml`, `webservices.xml`, the WSDL and/or Java source files, and make sure that the namespace URI and the service and port names all match.
- In the server log, a `NullPointerException` with a stack trace that includes a lot of `preProcessXXX` calls near the top is a different manifestation of the same sorts of problems as the “can't find the `<impl-class>Service.wsdl`” mentioned above.
- In testing a deployed service via SOAPPad, or seeing the traffic via a SOAPSniffer, if you see a fault response with the message “Could not find the dispatch method,” this means that the request message's operation element couldn't be resolved to a Java method on the SIB. Look at the WSDL `portType` and `operation` names, vs. the Java class/interface and method names, and at any Java annotations that affect how those names are mapped. Frankly, when this happens in a lab it's almost always a simple mistype in a source file somewhere. A missing custom-binding file can cause this, too – in our Ant builds, be sure that any custom-binding file is declared via the `build.properties` file so that the `wsimport` or `wsgen` command includes it with the `-b` switch.
- If you see an “unexpected argument name” fault response, this indicates a mismatch of the XML content against the schema-declared content model – so not about an RPC-style operation element but about document-style body content that doesn't match the message schema. Common causes include a missing `endpointInterface` attribute on the `@WebService` annotation, and a missing `@WebParam` annotation to customize a method parameter name so that it's not expected to be “arg0”.
- A successful message roundtrip, but with an empty response operation element, or other well-formed but unlikely response messages, usually indicate that arguments under the operation element in an RPC-style service were not as expected. Strangely this doesn't produce a fault; binding will proceed and arguments that were expected but not matched will be zero, `false`, or `null`. However this is handled in the SIB will dictate the response content, but for instance a `null` return value will appear as a response operation element with no child elements.

- If you've installed the Java EE 5 SDK in a location other than the default of **c:\Sun\SDK**, and you're using the Eclipse overlay, you will need to edit a couple values in the workspace before things work. In the main Preferences dialog, find **General / Workbench / Linked Resources**, and edit the **JAVA\_EE\_HOME** value that you see there. Also, under **Run/Debug / String Substitutions**, edit the value of **ASANT\_HOME**. You should clean all projects after making this change, and then rebuild selectively.
- If you're using the Eclipse overlay, and you see build errors along the lines of "can't find the class java.lang.Object on the build path" – and you've tried the obvious things like refreshing the project, checking the installed JRE(s), cleaning the project, etc. – try simply cleaning, shutting down Eclipse, and restarting. This bizarre problem has been observed in one classroom and it went away on a restart; we don't know what causes it but it may be triggered by the Java EE SDK being set up in a location other than the **c:\Sun\SDK** that is coded into the workspace in a couple places (see above note).

## **Errata**

There are no errata for the course at this time.

## **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.