



Capstone Courseware, LLC

33 Boylston Street
Jamaica Plain, MA 02130

877-227-2477
capstonecourseware.com

Developing Java Web Services

Version 1.5.3

Instructor's Guide



Overview

This course provides Java developers with an understanding of the interoperable and Java-specific Web services architectures, and with skills in building services and clients – using either JAXM/SAAJ for low-level SOAP messaging or JAX-RPC tools for rapid development of component-based services using WSDL and SOAP.

The first three chapters provide an architectural overview along with a short introduction to the tools that will be used for development in the rest of the course. The flow of the first day's discussion moves from the interoperable Web-service protocols – SOAP, WSDL, and UDDI – to the Java APIs and tools that wrap or otherwise implement them – JAXM/SAAJ, JAX-RPC, and JAXR, primarily. Chapters 1 and 3 especially are appropriate for non-developers with an interest in architecture and technology implications.

The next three chapters introduce SOAP and the low-level SOAP messaging APIs, SAAJ and JAXM. We study SOAP first, and then learn how to “do” SOAP in Java, and it is here that students will build their first fully-functional services and clients. Of these three chapters, SOAP and SAAJ are important to high-level services using JAX-RPC (which is where most of the world seems to be going), while JAXM is (a) strictly about low-level work and (b) optional for J2EE 1.4.

The course then moves onto the higher level of component-based, RPC-style services using WSDL and JAX-RPC for code generation and runtime SOAP serialization. Again, we start with the interoperable part, which is WSDL, and then move into Java with JAX-RPC and a chapter each on the two major development paths: what's symmetrical and what's not. A best-practices chapter, some of the EJB/JSP chapter, and a chapter on JAX-RPC handlers rounds out the heavier treatment of high-level services.

The last two chapters cover additional techniques – SOAP attachments and security – and apply to both low- and high-level strategies.





Timeline

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

Day 1

Chapter 1	Interoperable Web Services
Chapter 2	Building and Hosting Web Services
Chapter 3	The Java Web Services Architecture
Chapter 4	SOAP (runs to Day 2)

Day 2

Chapter 4	SOAP
Chapter 5	SAAJ
Chapter 6	JAXM

Day 3

Chapter 7	WSDL
Chapter 8	JAX-RPC
Chapter 9	Generating Web Services from Java Code

Day 4

Chapter 10	Generating Java Web Services from WSDL
Chapter 11	Best Practices and Techniques
Chapter 12	EJB, JSP and Web Services

Day 5

Chapter 13	Service Lifecycle and Message Handlers
Chapter 14	SOAP Attachments
Chapter 15	Security

See also suggestions on which labs to include in the teaching notes for Chapter 9, later in this document.





Tools Deployed with the Lab Software

This course's software requires separate setups of the J2EE 1.4 SDK, the Crimson text editor, and/or Eclipse 3.x; otherwise it is self-contained, as the lab installer sets up not only the lab software but necessary tools:

- **The Java API for XML Messaging, or JAXM.** This is the one API that we study that is not required of a J2EE 1.4 application server, so we set it up separately and our builds draw JAXM JARs into web applications and standalone-application classpaths.

Ant Builds

We use **asant** as provided by the J2EE SDK for all our builds. Though most students will be happy to let the **asant** 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**.

All projects that define web services build and deploy with **asant**, relying on the default target "all". This includes undeploying any old version of the application, building, and deploying; building in turn includes cleaning and re-creating a build directory, Java compilation, WAR and sometimes EJB assembly, EAR assembly, and EAR verification. The entire process is repeatable with no stray artifacts.

Client projects and other standalone applications will simply compile with **asant**. Then run them with **asant run-WS-client** or whatever other instructions are given in the coursebook.

Some projects require a database, and so the first time any of the versions of an application is built you should run **asant init-DB** before **asant**. You only need to run this once per application, though – that is, only once for Love Is Blind (the first demo of the course), once for SharewareWS. This runs two targets, "create-DB" which runs a SQL script to the RDBMS, and "create-DB-config", which uses **asadmin** to configure JDBC connection pool and resource for deployed EARs to use. If you want to return to a clean slate, run **asant asant undeploy**, then **asant clear-DB**.

asant list will show a complete list of deployed applications; this is handy when helping students diagnose problems on the client side, because you can quickly check if a given service is really deployed and ready for requests.



Teaching Notes

Chapter 1

First, the chapter is meant to put the Web-services buzz in broader context. “Why do we need these things, anyway?” should be the question that the instructor is implicitly answering for the first ten pages or so. At the end of the chapter, as well, there is an attempt to step back from the SOAP hype of the moment and note that there are non-SOAP approaches, and that there are just a few fundamental things that one can say about all Web services.

The big middle of the chapter, though, walks through the consensus architecture, which is of course SOAP-based: the interoperability stacks as laid out by Microsoft and IBM are discussed, and it’s a good idea to note how little of the architecture has really found a consensus technology to support it: SOAP, WSDL, and UDDI account for a fairly small part of the bigger picture of Web-based business software integration. The WS-I Basic Profile is of course more concrete and more important at this time, but even this fails to address a wide range of practical problems in the RPC space.

Chapter 2

This chapter acts as a pointedly tool-specific companion to the one before and the one after it. Essentially, we need to take a moment to talk about the practical matters of how to build and host services, so that we can start doing coding exercises, and for that we need a J2EE server, which implies some specific logistics for deployment and testing. We use common Ant targets to hide most of this complexity, and in this chapter students get a feel for how they will go about building, deploying and testing their applications. The main demo in this chapter also gives students their first look at a running service and client, and at SOAP messages passing in real time.





Chapter 3

This chapter starts from scratch, in a way, working through the Web-services problem now with the assumption that Java technology will be used for implementation. From this point forward, it will be important to keep the student's eye on the dualities of JAXM+SAAJ/SOAP, JAX-RPC/WSDL, etc., and to highlight the mappings from Java code to SOAP messages and WSDL content. Primarily the chapter compares what we're calling low- and high-level strategies for implementing SOAP messaging: SAAJ/JAXM vs. JAX-RPC, essentially. This sets a framework for the remainder of the course: Chapters 4-6 cover low-level services and Chapters 7-13 are almost all about the high-level approach. Some knowledge about SOAP and SAAJ is useful to effective JAX-RPC development, too, so it's not a strict dichotomy.

Chapter 4

This chapter introduces SOAP 1.1 "from scratch." A brief discussion of Web services and the role of SOAP are intended to set the context for the module's work and to motivate the study of SOAP and SOAP tools. Then the protocol is presented in some detail: SOAP namespaces, envelope, header and body, etc. To enable students to learn to read and write raw SOAP messages, the SOAPPad application is provided. The latter part of the chapter goes into more depth on validating messages and some details about the SOAP Section 5 encoding, the latter mostly so that encoding constructs that appear in JAX-RPC messages won't seem completely foreign later on.

You will find that the SOAPPad tool is generally useful in testing services as they are developed in later labs, and for illustrating various details of how services and clients are working. It's a good idea to create a desktop shortcut to the **run.bat** file. Also, for purposes of screen presentation, the tool has a configurable font size – you may find this handy. The **run** script takes a single command-line argument, so 'run 20' will give you 20-point type and a larger initial window size.





Chapter 5

Now the module moves to the question of how to read and write SOAP messages using Java: SAAJ as a thin wrapper around the SOAP content model. There is a certain amount of rather dry coverage of the many SAAJ interfaces. The UML diagram that provides an overview of these interfaces, and the restated diagram of the SOAP content model itself, should help to keep students' eyes on the bigger picture and to see the connections between the various SAAJ interfaces, while learning some of the key details of how to create and use envelope, part, body, element, etc.

The labs in this and the next chapter are fairly code-intensive and error-prone – this is to some extent an illustration of what it's like to build low-level services! Be sure to plan adequate time for these, perhaps being ready to go beyond the suggested completion times if possible. This is especially important for student groups who are less than perfectly fluent in Java (and who, therefore, probably shouldn't be taking this course! but it happens).

Chapter 6

Now we get to the JAXM and how to actually send and receive SOAP messages built with SAAJ. The JAXM architecture is discussed, but most of the interesting use of JAXM boils down to **JAXMServlet** and some boilerplate client code, so the labs are as much about reinforcing SAAJ knowledge as about getting a JAXM service working. By the end of this chapter students should be comfortable building JAXM/SAAJ services and clients. Though we return to this skill set later in the course, the next chapter is a bit of a key change as we switch to WSDL, JAX-RPC, and rapid development of component-based services.

Chapter 7

The beginning of this chapter develops some motivation for use and study of WSDL, especially by comparison to major DOC infrastructures, each of which features an IDL of some sort. Then the language is developed component-by-component, working from abstract model to concrete, with comparisons to the SOAP content that is expected based on various WSDL descriptions. The labs are the only hands-on work in writing WSDL descriptors, and should be completed by all students if at all possible. (By contrast, learning to write SOAP by hand is not so critical – there it's just a matter of getting familiar with syntax, writing as reinforcement to reading, really. Skill in writing WSDL descriptors is of real, practical value.)





Chapter 8

This chapter takes a “how does it work?” approach to JAX-RPC, seeing how a minimal body of WSDL and/or Java code can be parleyed into a fully-functional Web service or client. Where the previous chapter explicitly compares Web services to COM, CORBA, and EJB, in this chapter we're a little more detail-oriented; it may be interesting and help tie the lecture together to compare JAX-RPC to RMI and EJB, especially if students have any exposure to the latter technology.

If students are not satisfied with seeing only the WSDL in the Java-to-WSDL demo, and you want to show them the generated Java files, build and deploy the service and then look under `%J2EE_HOME%\domains\domain1\generated\ejb\j2ee-apps\PigLatin\JavaToWSDL\cc\language`. All the classes that are checked in the page that compares this demo to the WSDL-to-Java one will appear in this directory. You can do the same thing for LoveIsBlind in Chapter 9.

Chapter 9

The rubber hits the road in this chapter and the next, and these two are lighter on lecture and heavier on lab time. Here we start to look at the details of Java-to-XML/WSDL mapping, and also address the common development problem of converting or enhancing an existing Web application to provide SOAP-based Web service. (A working title for the chapter was “Adding a SOAP Interface to Java Code.”) This is of practical interest to many developers and also makes it easy to concentrate on what's distinct about Web-service development by assuming that domain and persistence code exist (and are well-organized and correctly decoupled).

Lab 9 is meant to be a fairly in-depth exercise not only in mapping/generating from Java, but in the sorts of domain changes or adaptations one often must perform in adding SOAP service to a codebase. This provides good depth, but does involve some low-level and error-prone coding. We've tried to minimize the risk of going off the tracks by providing complete new source files in some cases – it's a good idea to point these out in advance.

You will probably find in most classrooms that the group is either more interested in Java-to-WSDL development or WSDL-to-Java. There should be some good discussion of which is preferable in this span of chapters (8-11), but there will usually be a preference, even if it's purely practical. As such, the course provides more lab work in these two chapters than will probably fit, with the intention that either Lab 9 or Labs 10C and maybe 10B will be skipped. You might want to sound students out about this earlier in the week and plan time accordingly.



Chapter 10

Now we do a 180 and look at WSDL-to-Java development. Again, we walk through the mappings in detail, this time considering issues specific to generation from WSDL. It will be good to emphasize that WSDL-to-Java can be used to create services and not just clients. Although this approach will be of less practical use in the short term (compared, say, to converting existing Web applications by generating from Java code), in the abstract it is preferable and in the long term it will be a more common scenario. The arguments to this effect have already been made in Chapter 3, but some of these are emphasized in this chapter and in Chapter 11, now that more details about the mappings and process have been explained: client-side validation, strong typing in XML Schema, etc.

Again, it is not expected that students will do all of Labs 9, 10A, 10B and 10C – see notes under Chapter 9.

Chapter 11

With both paths for JAX-RPC development considered, this chapter wraps up discussion of basic JAX-RPC use by highlighting some issues and practical considerations. The initial discussion of which path to use in development is usually a spirited one, and helps to get at the real implications of using JAX-RPC in a J2EE environment. The interoperability section is really just an extension of that discussion, pointing out the basic difference between portability and interoperability and why Java-to-WSDL can be called interoperable without providing reliable portability or even binary compatibility.

The polymorphism example should be an eye-opener, especially as we've been comparing JAX-RPC to RMI and EJB for a few chapters, now. The rest of the chapter is a bit fragmented, covering things like DII and type mapping for completeness, but the last section on passing objects should wrap things up nicely.

Chapter 12

We've been assuming a servlet endpoint model pretty much since the introductory chapters, and now it's time to look at the broader range of implementation options for services and clients, especially EJB and JSP. The chapter assumes very little for purposes of discussions, but some previous understanding of EJB will be required to follow much of the EJB examples, and similarly students will need to know some JSP in order to do the lab.



Chapter 13

This chapter primarily focuses on managing SOAP headers using JAX-RPC handlers, but on the way there it seems worth a quick discussion of service lifecycle, especially now that J2EE has formalized the lives of JAX-RPC services in either servlet or EJB containers. Students should see at this point that Web services really are first-class J2EE components, and that to get past the tinkertoy stage they will need to think like component developers: look up references to other components, databases, environment entries and so on.

The UML diagrams are meant to establish a theme, which is that from three disparate starting points – servlet endpoint class, EJB endpoint class, message handler – the developer can navigate to the same SOAP message context object, and from there can manage and share information, about the message and about how it is to be processed. If time permits, the instructor may want to expand on the fairly simple exercise in Lab 6 by introducing the possibility of faults – this complicates the possible message paths between handlers and endpoint considerably, and is of course a practical consideration.

The closing pages on session management are included for their practical value in the near term. It would be good to emphasize that reliance on HTTP sessions (and particularly use of the HTTP session object) is not going to be the best practice in the long term. Those who may need to implement session management in upcoming projects might be encouraged to isolate and to wrap their session management logic in a layer that can be redirected to SOAP-header-based sessions, when JAX-RPC and J2EE support those.

In the session-management demo, we have gotten reports of the FrontDesk client failing to send the session cookie back to the server, and thus triggering an exception on the second remote call. We haven't been able to reproduce the problem; however, instructors are urged to test this demo carefully beforehand. If necessary, the best workaround is probably to build the service and only to test it with SOAPPad, as described, but to leave FrontDesk out of it if it isn't performing well.





Chapter 14

This chapter is perhaps the most cookbook-style of the course. The idea of attachments is fundamentally simple, and adds an important dimension to SOAP messaging for Java applications. So without a lot of embroidery we plough through the relevant SAAJ classes, with a quick into of the JAF. (There is a great deal more to learn about MIME, JAF, and runtime support for various content types, but to go much further than the current chapter content would require a lot more time. You might want to expand on this material informally.)

On the JAX-RPC front we have similar simplicity for a different reason, which is that JAX-RPC doesn't really have the attachment question solved all that well just yet – or perhaps because RPCs and attachments really don't mix well. Nonetheless, given the popularity of JAX-RPC over JAXM at the time of this release, it seems appropriate to focus the lab work on JAX-RPC.

Chapter 15

This final chapter on Web-service security only scratches the surface of what will become a massive topic. It provides an overview of the sorts of issues Web services will face, and of the areas of technology from which solutions will emerge: mostly this means J2EE (via the JCP) and XML/SOAP (via the W3C, OASIS, the WS-I, etc.). Most of the real practice is in securing URLs – a traditional Web-application technique.





Revision History

Revision 1.5.3 is a maintenance release with fixes for typos and other minor defects. 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.

Some specific issues that often arise:

- Make sure the **JAVA_HOME**, **J2EE_HOME** and **CC_MODULE** variables are correctly set.
- Make sure that executable path includes the **bin** directory from J2SE and J2EE, in that order.
- You will see Java-5.0 compiler warnings in many of the projects when you build them. Some of these are just older application code that has yet to be updated to Java-5.0 best practice; some is unavoidable since the SAAJ 1.2 API doesn't work with generics. All such warnings are benign.
- We've seen some odd behavior, very hard to reproduce, in the **SOAPPad** tool: it sometimes seems to fail quietly on the first request after being started. If you don't see the result you were expecting (or don't see any result at all), always try clicking **Send** a second time. If it continues to fail, then you probably have a real problem with your SOAP message or with the service, but that first-request failure seems to be a regressive bug in SOAPPad itself.
- A few projects use JSTL 1.0 and each will trigger a single error in the J2EE verifier. You'll see a message to the effect that it can't find the tag-library validator class. These errors are benign.
- The setup guide requires that the J2EE 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**: **AS8.2.properties** has the admin username, and **AS8.2.password** has the admin password.





Errata

The following issues have been reported since the release of this version of the course:

- In Lab 5B, Step 14 says to run 'asant build-clients'. The correct command is 'asant compile'.





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.

