

# **EJB on JBoss<sup>®</sup>**

**Version 10.3.0**

**Instructor's Guide**

## Overview

EJB has seen a dramatic improvement in usability with the 3.0 release that's part of Java EE 5, and it looks as if this might bring a resurgence of interest. One thing to understand about this course is that it is geared primarily to the student with no prior EJB experience – and not so much to the EJB 2.x developer looking for a skills upgrade, although that is certainly a possible class of participant. A bit more on this in the Timeline section, but basically we're presenting EJB 3.0, fresh and unencumbered by its past, to experienced Java developers – preferably Java 5 since we'll use annotations so heavily, but Java-1.4 programmers shouldn't have any big trouble.

In fact the technology has become so simple to use in some areas that parts of our previous course have collapsed into smaller chapters or have gone away completely, making room for some new topics. Stateless session beans are very nearly trivial to implement, especially if they're bound into an enterprise application with the web tier that will invoke them. Message-driven beans that read from JMS queues are also very simple. Clearly the part that's still not so simple is persistence with what are now called "entities." ORM is of course a thorny problem, and even with the help of TopLink or Kodo there are complexities to confront, and odd little limitations and pitfalls.

Another area that requires attention, and is trickier than it might seem at first, is the much-ballyhooed dependency-injection feature of EJB 3.0. We've devoted a full chapter (and one of the longer ones actually) to this topic because we've found that while DI is indeed a powerful tool and well worth using, it is not so simple and magical as much of the current literature would have us believe. See the Chapter-7 notes for more on this.

This course is dedicated to EJB3 as practiced on JBoss 5.1. Happily, JBoss implements the specification – and Java EE 5 as a whole – just about perfectly. So there's not a lot of hand-waving about features that should be there or that work in some funky way or that compromise portability. We do observe a few small compliance failures (and unclear compliance requirements) toward the end of the course, and these should be instructive. But the theme is that wherever code portability is a concern, the EJB3 specs (and JBoss) are doing their jobs.

## Timeline

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

### Day 1

2 hours Chapter 1

2½ hours Chapter 2

3 hours (spans days) Chapter 3

### Day 2

2 hours Chapter 4

3 hours Chapter 5

### Day 3

1½ hours Chapter 6

3 hours Chapter 7

1½ hours Chapter 8

### Day 4

3½ hours Chapter 9

½ hour Chapter 10

2 hours Chapter 11

### Day 5

2 hours Chapter 12

2 hours Chapter 13

There should be no problem covering all the material in five days, given a group with the prerequisite experience. If there is interest, the instructor might consider setting up a related add-on module entitled “JSF, JPA, and EJB3.” This can be found on the Capstone website, in the technical library:

**<http://capcourse.com/Library>**

## Tools Deployed with the Lab Software

This course's software requires separate setups of JDK6, JBoss 5.1, the Crimson text editor, and/or Eclipse Galileo; otherwise it is self-contained, as the lab installer sets up not only the lab software but necessary tools – find these under **c:\Capstone\Tools** once the labs are installed.

For this course the only such tool is the **MySQL** RDBMS, deployed in **c:\Capstone\Tools\MySQL5.0**. This is not the full 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. Further documentation on how we use this tool is found in the coursebook itself – see Chapter 2 just prior to the first working example.

## 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\EJB\Ant**. Information here defines a routine for building Java EE targets including WARs, EJB JARs, EARs, application client JARs, and standalone Java applications.

## 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. The workspace and projects have been tested thoroughly with the core course material, 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, and for those interested in Java EE development using Eclipse.

While the coursebook contains no references to Eclipse and no IDE-specific demo or lab instructions, it is generally intuitive to map the coursebook instructions (“build,” “run,” “deploy,” etc.) to the appropriate actions in the IDE. Since the courseware is written to work with or without the IDE overlays, there are some things that are not as intuitive, and we’ve documented those in a set of files bundled into the workspace itself. Follow the pointers given in `c:\Capstone\EJB\Eclipse\ReadMe.html` (which appears front-and-center when the workspace is first opened) to find notes on how to use the Eclipse overlays for Capstone courses generally, and on the particulars of this workspace for this course. Please read through those documents in detail before jumping into using the workspace in class, and 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.

Again, this IDE layer of the courseware is an optional piece, and while we’ve taken care to test it fully, we can only offer complete technical support on the standard course. If a given exercise is giving trouble, please (a) check the troubleshooting notes both in the workspace itself and in this instructor’s guide, and (b) be certain to build and run it from the command line, using the simpler and more predictable Ant-based builds as prescribed in the coursebook, before contacting Capstone.

## Teaching Notes

### Chapter 1

Ordinarily we'd have a single chapter on a given technology entitled "Overview" or "Architecture," to define the motivation and introduce the main concepts. Here we have two such chapters: one is essentially about "why?" and the other is an introduction to "how?" This overview includes a lot of motivation material, as in look at all the wonderful things you get for free. It should go quite quickly, especially for those with any experience with Java web development or earlier EJB versions.

### Chapter 2

This chapter is the true architecture chapter: it starts with the assumption that, yes, we do want to build EJBs, and jumps into architecture and the beginnings of how-to. The centerpiece of this chapter is the initial example, **Wholesale/Step1**, which is introduced in several segments before instructions are finally given to build and test it. In this way you can touch down on working code as you move to each new concept: IoC, metadata, deployed structure, and DI.

The design exercise should be an opportunity for students to start thinking in EJB terms when designing a small application. Of course here there is not one precisely right answer, just a suggestion, and the discussion should focus on the big concepts and not so much on what is exactly the perfect design.

## Chapter 3

We start our detailed study of EJB with session beans for two reasons: they are the natural entry point into an EJB system; and they are probably the simplest of the three bean types to implement (with MDBs admittedly being a close second). The techniques in this chapter should be straightforward to present and for students to grasp. Probably the most interesting issue here is the stateful/stateless choice and the impacts of one or the other. SOA thinking has put something of a stigma on stateful EJBs, and many people more or less write them off before really considering a design. Remember that not all EJB systems are service-oriented architectures; some are traditional, tightly-coupled systems in which a stateful remote object may well be the best solution to a problem. Neither do stateful beans imply long-running transactions; they introduce certain concurrency issues but really those issues would arise in the analogous stateless service anyway.

It's worth noting that the pooling behavior of JBoss is different from the other application servers for which this course has been developed – namely GlassFish and WebLogic. Both of those products instantiate stateless beans as singletons, while JBoss creates instance pools. (Of course everybody pools stateful beans.) Deep discussion of the relative merits of the two approaches may not be worth a lot of class time, but this does provide one clear example of the value of standards, as vendors can pursue their own ideas about threading, memory usage, and performance, while adhering to a contract with the application developer.

(Also, this pooling behavior made a later lab much tougher to present, because that lab left a latent bug in the code that is exposed only for the singleton-policy servers, so that the starter code works on JBoss, and at first blush there is nothing to do! More on this in the notes for Chapter 7.)

## Chapter 4

This chapter introduces the Java Persistence API (JPA) part of the EJB 3 specification, which will become an independent specification in the next version. The defined JPA annotations make mapping classes to tables quite straightforward, made even easier by the “configuration by exception” approach. Since JPA is effectively a separate specification, the examples in this chapter show JPA used in both Java SE and Java EE applications.

Relationship mappings are saved for the next chapter, so this chapter's mappings are particularly simple. It is arguably more important in this chapter is illustrate how to acquire and use an entity manager to interact with the persistence provider.

## Chapter 5

Programs are never as simple as a single entity class mapped to a single database table. The issue in this chapter is how to deal with multiple classes and tables in all the standard relationships, including one-to-one, one-to-many, many-to-one, and many-to-many.

It is particularly important to emphasize the concept of the owner of a relationship, because the form of the mapping annotations is dependent upon which class maps to the table with the foreign key. This leads to the one truly strange part of the specification, which is the lack of an easy mapping for unidirectional one-to-many relationships. The specification recommends that this relationship be mapped using a link table, which is unusual to say the least. In this chapter we try to illustrate that problem along with a couple of possible approaches to solving it in a real application.

Though it probably won't come to light during the normal run of this class, a quick note about a JPA portability issue here. In the Wholesale application, starting with Step2, the **removeItem** method on **RecordsImpl** had to be tweaked in order to work with Hibernate. Simply calling **em.remove** is enough on TopLink or Kodo (the default JPA provider for WebLogic); but Hibernate would instead throw an **EntityNotFoundException**, with the message "deleted entity passed to persist: [cc.sales.Item#<null>]". The line of code immediately before it, that manually removes the item from its shipment's collection of items, was added to make Hibernate happy. This seems to be a cascading failure, and a JPA compliance failure as well, but it's probably more detail than you want to get into with students just learning JPA.

## Chapter 6

The concept of an object-oriented SQL is new to many students, who may even question the need for such a construct. Hopefully students with enough Java experience to be interested in EJBs at all will find the simplicity of JPQL appealing. The lack of explicit joins makes writing JPQL queries much easier, and the dot notation for following associations is a natural way of thinking for Java developers.

The lab at the end of this chapter really drives the concept home, because in order to write proper JPQL queries the students will need to think in terms of the class structure rather than the database schema. If the students are struggling, a rough UML class diagram sketched on the board can help them visualize the forms of the queries.

## Chapter 7

Most articles, tutorials, and even textbooks treat dependency injection like some magic incantation: slap the **@EJB** annotation on an interface reference and, presto! you get connected to the EJB of your desires. Sometimes this simple usage works, and sometimes it's even the best practice; but in many cases it's either insufficient or inadvisable. With this chapter we're trying to demystify DI and make clear the various ways in which a container can connect two components – and which approaches it will try first, second, last. It is indeed an excellent system, but it seems that having promoted EJB 3 as being dramatically unlike EJB 2.1, its proponents then were a little shy about talking about the one feature that really does call for an XML deployment descriptor. Try to make the case that while many of the EJB annotations make good sense as elements of a Java source file, when we get to DI we're talking about metadata that transcends the scope of one component, and therefore really belongs in an external document such as a deployment descriptor – even if we believe that all other metadata is best expressed in the source files.

Another little point, not made in the book but visible in some of the lab work, is that using dependency injection as opposed to the EJB-2 style of JNDI lookup allows for build-time name checking: for example if I call for a specific **beanName**, or even give my reference its own **name** for purposes of explicit wiring by a descriptor, the EJB verifier can assert that the **beanName** or **name** is resolved. JNDI lookups can't be checked in that way – even the slick new ones that pass through the EJB context interface, although those have their place in EJB3 development.

To follow up on a point from the Chapter 3 notes, Lab 7A is a little longer and more interesting in this JBoss version of the course than in others. The gist of this lab is that some intentionally sloppy coding – using **@EJB** to inject a stateful bean into a stateless one – causes concurrency failures. And so it does, in an obvious way, on GlassFish and on WebLogic – but not on JBoss! Since JBoss pools even stateless beans, the error is not exposed in casual testing.

We've tried to make a virtue of this by leading students through a quick exploration of JBoss-specific annotations to control pooling policies. By doing so, they are able to simulate – on a single machine with a couple of local browser instances – the very failure that would come to light in production eventually. Then they can fix the actual code problem, just as we usually do in simpler versions of this lab in other course variants.

## **Chapter 8**

Of course the big new concept for this chapter is asynchronous messaging. Students may already be familiar with messaging, and perhaps with JMS; or this may all be new territory. So we give a brief backgrounder in JMS before tackling MDBs proper. If more depth on JMS is needed, consider adding chapters from Course 120.

## **Chapter 9**

We circle around to persistence coding again, now to consider database transactions. In earlier chapters we've more or less assumed a single user and valid inputs, but of course real systems must plan transactional behavior carefully, and now we're in a good position to address that. So we talk about JPA transactions, and then EJB container-managed transactions, and eventually on to application-managed transactions.

There are a couple other important concepts for those who wish to understand how their persistence code is really working, and to code effectively. One is persistence contexts, and we discuss that (and confront it in a lab) in this chapter. The other is exception handling, which we consider in the following chapter.

## Chapter 10

This mini-chapter is like the previous chapter in that it goes back and explains what's been happening all along in the course exercises. We've always had transactions; we've encountered various exceptions, and now we understand the rules by which the container has been handling those exceptions, especially vis-à-vis transactions.

We do get a look at a new behavior, which is what the container does (and doesn't do) with application exceptions. This by way of a single brief example, and the chapter has no lab exercises.

## Chapter 11

This and the remaining chapters of the course are more in the way of independent topics; they don't build from chapter to chapter as we've been doing up to now. Security matters to some enterprises, and where it matters, frankly, a lot of it must be solved external to the EJB container and in ways that are not standardized. But we cover the EJB perspective on security in this chapter, which boils down to authorization policies, either declarative (annotations and XML are both covered) or programmatic.

## Chapter 12

Another take-it-or-leave-it technique: EJB interceptors, which give EJB applications the option of a filter-chain mechanism similar to servlet filters or JAX-WS message handlers. The basic mechanism is pretty simple to grasp and to implement.

This is an interesting case study in the question that's been bubbling along all through the course: which is better, annotations or XML? Interceptors seem to present a metadata problem that's best solved by a mixed approach. Interceptor classes are generally best annotated rather than externally declared; but bindings are by their nature external to both the interceptor and the target, much like dependency injection via ejb-links and JNDI lookups.

## Chapter 13

Finally we look at EJB timers – another feature that’s great if you need it, but may not come up at all in some sorts of EJB application development. There are a number of parallels here to JMS and MDBs, especially in how creating/canceling timers and calling timeout methods relate to container transactions.

One compliance note: WebLogic won’t allow a **private** timeout method – this is incorrect; see the EJB 3.0 Core specification, section 18.2.2.

One nice addition to the demo in this chapter is quickly to process several different orders: choose one, click the button, go back, choose another, click the button, etc. A separate timer will be set for each request, and this will help to show that the serializable state associated with timers really does travel in isolation from other timers – and, as I like to joke, that our System.out reporting approach is not thread-safe.

## Revision History

**Version 5.1** is the initial public release. It is based on Course 163, version 3.0.1.

## 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:

- Environment variables **CC\_MODULE**, **JAVA\_HOME**, **JBOSS\_HOME**, **JBOSS\_CLASSPATH** and the executable **PATH** must be set up as described in the coursebook (see Chapter 2 and the initial working example **Wholesale/Step1**). Most build failures having to do with not finding necessary JARs can be traced to one of these settings.
- If a student is getting errors that seem to lack explanation when invoking an EJB, try peeking at the tail of the server console, as errors sometimes appear there that are not reported through to a remote client.
- Be sure to run the **setup** script as directed for various exercises. If you miss this and move on to build and deploy an application that uses a database, the deployment will fail because the references in the EAR to a datasource will be irresolvable. In this case, use the **teardown** script to clean up – this undeploys the application and cleans up the database and any JDBC resources so you can start fresh.

- When running Java EE application clients (the Hello remote client in Chapter 3, the **readAMeter** and **flush** scripts in Chapter 8, and so on), it's important to understand that JBoss requires that application clients be deployed to the server in order to be launched. They are JNDI-named, and the most common error students will see if they try to test without building and deploying the client is a failure to find a certain name in the JNDI repository – for instance ending with “HelloClient” or “EnergyClient”. The Ant builds are set up to minimize the likelihood of this, as clients are deployed automatically during the build. But still it can come up, and when using the Eclipse workspace it's much more likely because there students must explicitly deploy and undeploy the EAR and the application client separately, so that it's easy enough to attempt a test without the client application being on the server yet. (See also extensive comments in the workspace notes on this issue.)
- Watch for timing surprises when testing JMS and MDBs. Specifically, we've observed a sequence in the Energy/JMS example where, even though the operator runs **readAMeter** to completion and then runs **flush**, the latter shows no output, and then a second run of the **flush** script will show the message data and remove the message from the queue. This is harmless, and it isn't any sort of compliance failure, as JMS explicitly disavows any timing guarantees between a producer and a consumer. Still, it's surprising! and suggests that JBoss's queue implementation may take a few seconds to complete a reliable delivery after the JMS client has completed its send and gone away.
- The **JNDIReport** application mentioned in Chapter 7 can be a useful diagnostic tool for earlier in the course as well.
- Also the command **ant list** from any project directory will list EARs and client applications currently deployed to the server, data sources deployed to the server, and databases currently installed on the RDBMS. It does this somewhat on the cheap, by showing filtered directory listings from the JBoss auto-deploy directory and MySQL directory tree, so the output could be prettier, but still it's a quick and easy way to check what's deployed and what's not.

## Errata

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

- No errata 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.