

Introduction to EJB

Version 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 it was honestly a challenge to find interesting things to say. 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 Hibernate there are complexities to confront and odd little limitations and pitfalls. Hence the three chapters in the middle of the course on this part of the technology.

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.

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

2½ hours Chapter 4

2½ hours Chapter 5

Day 3

1½ hours Chapter 6

3 hours Chapter 7

1½ hours Chapter 8

There should be no problem covering all the material in three full training days, given a group with the prerequisite experience. If there is interest, the instructor might consider setting up one or two related add-on modules:

- JSF, JPA, and EJB3
- EJB3 on the JBoss Application Server

These can both 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 the Java EE 5.0 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 – find these under **c:\Capstone\Tools** once the labs are installed.

The **JSTL** deployment in **c:\Capstone\Tools\JSTL1.1** supports JSTs in various web applications used in the course exercises. You can pretty much ignore this, but if students ask, “where are those custom tags coming from?” this is the answer.

More visible to students will be 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. An obvious alternative would be the JavaDB (a/k/a Derby) product that's bundled with the EE SDK, but we like that the courseware illustrates more explicitly the need for things like JDBC data sources and connection pools, and that configuring the server for use with an external database product is necessary but also not so painful. 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 **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\EJB\Ant**. Information here defines a routine for building Java EE targets including WARs, EJB JARs, EARs, application client JARs, and standalone Java applications.

The Java EE SDK Verifier

All our builds that deploy EARs to the application server will run the EE SDK's **verifier** tool as the end of the build process, just before deployment. This tool can be quite helpful in sniffing out coding errors that a Java compiler won't catch – especially over metadata, including references in XML descriptors that can't be resolved to Java classes or methods, and annotations that likewise don't line up to their intended targets. Generally, see the file **build/Verifier/MyEAR.ear.txt** for warnings, errors, and failures, and follow up on any that you see as they are probably indications of coding errors.

However the tool also produces some spurious negative output, and you may have to soothe students' fears that they're somehow building a lab incorrectly when in fact everything's fine. We've observed a few specific "false negatives" with which you should be familiar before class time:

- The verifier always complains that there was an error opening the "ZIP file" – this can safely be ignored in all cases.

```
WARNING: DPL5400:Exception occurred : error in opening zip file.
```

- As it works it's way through the EAR file, it will often throw out what looks like a warning, as shown below. In fact you may see a couple dozen of these. They are completely benign and this is a documented bug in an open-source tool used by the verifier.

```
Visiting non-standard Signature object
```

- For any application that carries an EJB **persistence.xml**, the verifier will complain about the means of generating IDs for new database rows. This appears to be a matter of the verifier relying too automatically on the default RDBMS provider. The default provider (JavaDB) is consulted on a question that should be delegated to the actual configured (MySQL), and this yields a false negative. Fortunately the server gets this right on the actual deployment. So, you'll see a summary like this in the **asant** output ...

```
# of Failures : 1
# of Warnings : 0
# of Errors   : 0
```

... and the following details in the verifier's report:

```
Exception [TOPLINK-7144] (Oracle TopLink Essentials - 2006.8 (Build
060830)): oracle.toplink.essentials.exceptions.ValidationException
```

```
Exception Description: SEQ_GEN_SEQUENCE: platform DatabasePlatform
doesn't support NativeSequence.
```

Eclipse Overlays

Capstone Courseware provides an optional package of workspace and project files for Eclipse 3.2 for this course. (See the course Setup Guide for download URLs.) Instructors, use this package on your own initiative and at your own risk. You should have experience yourself with Eclipse before using the overlay package in the classroom. The workspace and projects have been tested lightly with the course but are not part of the standard product.

That said, this overlay should save a good deal of work for those who wish to use Eclipse instead of the text editor and command-line tools that are standard for the course. See the file `c:\Capstone\EJB\Eclipse\ReadMe.html` for general notes on how to use the Eclipse overlays for Capstone courses. Be prepared to walk students through the first few exercises in Eclipse; the notes in this file are for experienced Eclipse users, and will not be clear to many students on their own.

Notes on the overlay for this course:

- The source files in this course's projects are in a subdirectory **src** of the main project directory. To make Eclipse happy in attaching to this directory structure, we need the **JavaSource** folder that you see in each Eclipse project – this attaches to the **src** directory. But then, to see the remaining resources in the exercise, we need a second folder that attaches to the root directory of the exercise: this is the **Resources** folder. This is not intuitive, and students will need some guidance. Especially, the **Resources** folder is important because it gives access to the **build.xml** which can and should be used to build and deploy the project using Ant.
- The **JavaEE** folder is there to hold JARs which are then placed in the classpath.

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

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.

Try to avoid long digressions on Java-5 annotations, by the way: it's a challenge to deliver all this material in two days so generally it's important to stay on topic, and this is just one of a few tempting topics that may draw a lot of questions.

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.

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.

If you are teaching this on a longer timeline and/or to a group with great interest in database transactions, not that although we don’t have time or space to cover them in depth, the lab code does offer some support. The **Wholesale** database includes an overlaid unique key that asserts that no product can appear in the same shipment twice. This constraint can be triggered by using the web application to modify one of the shipments so that it has duplicate product names on two or more lines, and then trying to save the modified shipment. The error handling is ugly, but the more important point is that the system is naturally transactional; the shipment will be returned to a consistent state in this case. You might even start to place transaction attributes over the relevant EJB methods to show how the transaction support can be disabled, with bad results.

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.

The other relationships in this chapter are implemented pretty intuitively and shouldn't cause the students too many problems. There is one remaining oddity, described and addressed right at the end of Lab 5B. You may want to mention that this is a reported bug with TopLink's JPA implementation; in fact, with Hibernate configured as the JPA provider, we've seen firsthand that this problem goes away.

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.

Chapter 8

We really wanted to paint a complete picture of EJB3 in this course, even if at an introductory level. So we decided to get in a chapter on MDBs; these had been pushed off to the “Effective EJB” module in the 2.1 course. We feel like this should be in striking distance for a two-day course, given the great simplifications in the EJB 3.0 API and usage model. Still, it's clearly a discrete topic, not necessarily all that tied in to the general workings of session/entity systems that account for most EJB development. Treat this chapter as optional and note that it is pretty much pure cookbook: quick description of JMS and MDBs, simple example, simple lab. If time is very tight at the end but you want to cover something, skim the material and jump right to reviewing the lab answer, since it's a familiar system at that point and the minor addition of an MDB brings an interesting new dimension to the application.

Revision History

Version 3.0 treats EJB 3.0, and represents an overhaul of the entire course. There are some passages of text and some diagrams drawn from earlier versions, but most of the book and all of the lab code is entirely new.

Version 2.1.2 is a maintenance release, fixing minor typos in the book, and with no changes to the lab software.

Revision 2.1.1 re-fits the course software to the final release of the J2EE 1.4 reference implementation, including Sun's AppServer 8, PointBase, and IMQ. The most significant changes:

- Ant builds have been overhauled to work with AS8, and refactored a bit in the process. Happily, all administrative and development tasks can now be performed from within the project space using various Ant targets - no more manual editing of the server's config files.
- We now use PointBase instead of Cloudscape, and in the process have switched to embedded use, so there is no need to manage a separate server process. Generally, much less of the RDBMS administrative work is exposed.
- A new tool JNDIRport is included to check on available JNDI names.

Version 2.1 effects the following changes:

- This course version focuses on the EJB 2.1 specification. All software runs under the J2EE 1.4 reference implementation. Some discussion of differences between EJB 1.x and 2.x has been preserved (for CMP chapters especially).
- All code exercises have been simplified significantly by moving to the Ant build utility from the old system of DOS batch files.
- Introductory materials have been beefed up. The first two chapters have been consolidated into one large chapter that treats EJB architecture and development process more seamlessly. This chapter has a new demo and lab, a very simple "Hello, EJB" exercise that gives students an easier introduction to the unique aspects of EJB development.
- Old Chapters 3 and 4 have been consolidated into one chapter on BMP entities, and the following chapter is now a bit more symmetrical in treating CMP entities.
- The previous Chapter 6, on session beans, has been split into a chapter each on stateless and stateful beans. This and the other chapter-level reorganization has evened out the time spent per chapter, where time (lab time especially) had been heavily back-loaded.

Version 2.0 focuses on the EJB 2.0 specification. All software runs under the J2EE 1.3 server. Some discussion of 1.1 differences has been preserved, but the focus is on 2.0.

Version 1.1 effects the following changes:

- There is now much less focus on the interactive use of the J2EE's deploytool. We've retained the initial foray into use of this tool for deployment of the Love Is Blind application, because it does present a lot of useful elements of the deployment descriptor and process step-by-step and page-by-page. After this initial use, however, the GUI for the tool seems increasingly cumbersome, and its use is unacceptably error-prone. Instead, we've added deployment scripts to all example and lab steps, similar to those that were in use for the second module in the 1.0 release of the course. Instructions for labs and the remaining Love Is Blind demos are therefore much shorter and simpler, and lab times have been adjusted to this fact.
- A number of basic syntactic errors have been corrected in both the student guide and the code base

Version 1.0 was 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:

- Environment variables **CC_MODULE**, **JAVA_HOME**, **JAVA_EE_HOME**, 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.
- A **NullPointerException** in **EntityManagerFactory.createEntityManagerFactory** usually indicates either a missing **persistence.xml** file, a misplaced file, or a mismatch in the name of the persistence unit as referenced from the Java code.
- If a student is getting errors that seem to lack explanation when invoking an EJB, try peeking at the tail of the server log:
%JAVA_EE_HOME%/domains/domain1/logs/server.log.
- 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.
- Sometimes MySQL will have trouble starting because AS9 has already glommed up the port that it wants to use, which is 3306. AS9 gradually starts using up lots of local ports for various purposes, taking whatever it can find in a certain range. You can reconfigure MySQL to use another port, but usually it's best to just follow a startup sequence by which MySQL goes first, then AS9.
- The **JNDIReport** application mentioned in Chapter 7 can be a useful diagnostic tool for earlier in the course as well. Also the command **asant list** from any project directory will list EARs currently deployed to the server.
- See the earlier page about the Java EE SDK verifier – this tool can help sniff out problems, but also throws up some false negatives and you need to be aware of those.

Errata

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

- In Chapter 4, on page 92, the code listing for Product.java is missing a line- the **@GeneratedValue** annotation that's in the actual source file is not shown.
- Chapter 4, page 101, the command 'run cc.db.ListProducts' should be simply 'run ListProducts'.
- In Lab 4, Step 23, there is a note about any comments added during the session disappearing when you return to **summary.jsp**, and this note is incorrect. It seems that in addition to being held as a session-scope attribute, the proposal object is also held in the JPA provider's cache. Redeploying the application illustrates the key difference more effectively, which is that simple fields such as applicant name are truly persistent now, while comments won't be until next chapter.
- Lab 8, step 10, the phrase 'the message string "contributor"' often confuses students, who think this means the literal string "contributor" where what we mean is to call **getString** on the **MapMessage** object, passing "contributor" to get that value as wrapped in the JMS message. Same story for "comment". The result is that every time the student runs the **makeAComment** script, another comment appears in the web page, but with "contributor" and "comment" as the values, rather than the actual contributor and comment as provided on their command lines.

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.