

SAMPLE CHAPTER



EJB 3 IN ACTION

SECOND EDITION

Debu Panda
Reza Rahman
Ryan Cuprak
Michael Remijan



EJB 3 in Action

by Debu Panda
Reza Rahman
Ryan Cuprak
Michael Remijan

Chapter 1

Copyright 2014 Manning Publications

brief contents

PART 1 OVERVIEW OF THE EJB LANDSCAPE.....1

- 1 ■ What's what in EJB 3 3
- 2 ■ A first taste of EJB 25

PART 2 WORKING WITH EJB COMPONENTS.....47

- 3 ■ Building business logic with session beans 49
- 4 ■ Messaging and developing MDBs 93
- 5 ■ EJB runtime context, dependency injection, and crosscutting logic 117
- 6 ■ Transactions and security 160
- 7 ■ Scheduling and timers 196
- 8 ■ Exposing EJBs as web services 214

PART 3 USING EJB WITH JPA AND CDI251

- 9 ■ JPA entities 253
- 10 ■ Managing entities 294
- 11 ■ JPQL 321
- 12 ■ Using CDI with EJB 3 359

PART 4 PUTTING EJB INTO ACTION395

- 13 ■ Packaging EJB 3 applications 397
- 14 ■ Using WebSockets with EJB 3 427
- 15 ■ Testing and EJB 458

What's what in EJB 3

This chapter covers

- The EJB container and its role in Enterprise applications
- The different types of Enterprise Java Beans (EJBs)
- Closely related technologies such as the Java Persistence API (JPA)
- The different EJB runtime environments
- Innovations started with EJB 3
- New changes with EJB 3.2

One day, when God was looking over his creatures, he noticed a boy named Sadhu whose humor and cleverness pleased him. God felt generous that day and granted Sadhu three wishes. Sadhu asked for three reincarnations—one as a ladybug, one as an elephant, and the last as a cow. Surprised by these wishes, God asked Sadhu to explain himself. The boy replied, “I want to be a ladybug so that everyone in the world will admire me for my beauty and forgive the fact that I do no work. Being an elephant will be fun because I can gobble down enormous amounts of food without being ridiculed. I’ll like being a cow the best because I’ll be loved by all and

useful to mankind.” God was charmed by these answers and allowed Sadhu to live through the three incarnations. He then made Sadhu a morning star for his service to humankind as a cow.

EJB too has lived through three major incarnations. When it was first released, the industry was dazzled by its innovations. But like the ladybug, EJB 1 had limited functionality. The second EJB incarnation was almost as heavy as the largest of our beloved pachyderms. The brave souls who couldn't do without its elephant power had to tame the awesome complexity of EJB 2. And finally, in its third incarnation, EJB has become much more useful to the huddled masses, just like the gentle bovine that's sacred for Hindus and respected as a mother whose milk feeds us well.

A lot of hard work from a lot of good people made EJB 3 simple and lightweight without sacrificing Enterprise-ready power. EJB components can now be Plain Old Java Objects (POJOs) and look a lot like code in a “Hello World” program. In the following chapters we'll describe a star among frameworks with increasing industry adoption.

We've strived to keep this book practical without skimping on content. The book is designed to help you learn EJB 3 quickly and easily without neglecting the basics. We'll also dive into deep waters, sharing all the amazing sights we've discovered and warning about any lurking dangers.

In the Java world EJB is an important and uniquely influential technology radically transformed in version 3. We'll spend little time with EJB 2. You probably either already know earlier versions of EJB or are completely new to it. Spending too much time on previous versions is a waste of time. EJB 3 and EJB 2 have very little in common, and EJB 3.2 now makes support for EJB 2 optional. But if you're curious about EJB 2, we encourage you to pick up one of the many good books on the previous versions of EJB.

In this chapter we'll tell you what's what in EJB 3, explain why you should consider using it, and outline the significant improvements the newest version offers, such as annotations, convention-over-configuration, and dependency injection. We'll build on the momentum of this chapter by jumping into code in chapter 2. Let's start with a broad overview of EJB.

1.1 EJB overview

The first thing that should cross your mind while evaluating any technology is what it really gives you. What's so special about EJB? Beyond a presentation-layer technology like JavaServer Pages (JSP), JavaServer Faces (JSF), or Struts, couldn't you create your web application using the Java language and some APIs like Java Database Connectivity (JDBC) for database access? You could—if deadlines and limited resources weren't realities. Before anyone dreamed up EJB, this is exactly what people did. The resulting long hours proved that you'd spend a lot of time solving very common system-level problems instead of focusing on the real business solution. These experiences emphasized that there are common solutions for common development problems. This is exactly what EJB brings to the table. EJB is a collection of “canned” answers to common server application development problems, as well as a roadmap to common server component

patterns. These canned solutions or services are provided by the EJB container. To access these services, you build specialized components using declarative and programmatic EJB APIs and deploy them into the container.

1.1.1 EJB as a component model

In this book, EJBs refer to server-side components that you can use to build the business component layer of your application. Some developers associate the term *component* with developing complex and heavyweight CORBA or Microsoft COM+ code. In the brave new world of EJB 3, a component is what it ought to be—nothing more than a POJO with some special powers. More importantly, these powers stay invisible until they're needed and don't distract from the real purpose of the component. You'll see this firsthand throughout this book, especially starting in chapter 2.

To use EJB services, your component must be declared to be a recognized EJB component type. EJB recognizes two specific types of components: session beans and message-driven beans. Session beans are further subdivided into stateless session beans, stateful session beans, and singletons. Each component type has a specialized purpose, scope, state, lifecycle, and usage pattern in the business logic tier. We'll discuss these component types throughout the rest of the book, particularly in part 2. For data CRUD (create, read, update, delete) operations in the persistence tier, we'll talk about JPA entities and their relationship with EJBs in detail in part 3. As of EJB 3.1, all EJBs are managed beans. Managed beans are basically any generic Java object in a Java EE environment. Contexts and Dependency Injection (CDI) allows you to use dependency injection with all managed beans, including EJBs. We'll explore CDI and managed beans further in part 3.

1.1.2 EJB component services

As we mentioned, the canned services are the most valuable part of EJB. Some of the services are automatically attached to recognize components because they make a lot of sense for business logic-tier components. These services include dependency injection, transactions, thread safety, and pooling. To use most services, you must declare you want them using annotations/XML or by accessing programmatic EJB APIs. Examples of such services include security, scheduling, asynchronous processing, remoting, and web services. Most of this book will be spent explaining how you can exploit EJB services. We can't explain the details of each service in this chapter, but we'll briefly list the major ones in table 1.1 and explain what they mean to you.

Table 1.1 EJB services

Service	What it means for you
Registry, dependency injection, and lookup	Helps locate and glue together components, ideally through simple configuration. Lets you change component wiring for testing.
Lifecycle management	Lets you take appropriate actions when the lifecycle of a component changes, such as when it's created and when it's destroyed.

Table 1.1 EJB services (*continued*)

Service	What it means for you
Thread safety	EJB makes all components thread-safe and highly performant in ways that are completely invisible to you. This means that you can write your multi-threaded server components as if you were developing a single-threaded desktop application. It doesn't matter how complex the component is; EJB will make sure it's thread-safe.
Transactions	EJB automatically makes all of your components transactional, which means you don't have to write any transaction code while using databases or messaging servers via JDBC, JPA, or Java Message Service (JMS).
Pooling	EJB creates a pool of component instances that are shared by clients. At any point in time, each pooled instance can only be used by a single client. As soon as an instance is done servicing a client, it's returned to the pool for reuse instead of being frivolously discarded for the garbage collector to reclaim. You can also specify the size of the pool so that when the pool is full, any additional requests are automatically queued. This means that your system will never become unresponsive trying to handle a sudden burst of requests. Similar to instance pooling, EJB also automatically pools threads across the container for better performance.
State management	The EJB container manages the state transparently for stateful components instead of having you write verbose and error-prone code for state management. This means that you can maintain the state in instance variables as if you were developing a desktop application. EJB takes care of all the details of session/state maintenance behind the scenes.
Memory management	EJB steps in to optimize memory by saving less frequently used stateful components into the hard disk to free up memory. This is called <i>passivation</i> . When memory becomes available again and a passivated component is needed, EJB puts the component back into memory. This is called <i>activation</i> .
Messaging	EJB 3 allows you to write message processing components without having to deal with a lot of the mechanical details of the JMS API.
Security	EJB allows you to easily secure your components through simple configuration.
Scheduling	EJB lets you schedule any EJB method to be invoked automatically based on simple repeating timers or cron expressions.
Asynchronous processing	You can configure any EJB method to be invoked asynchronously if needed.
Interceptors	EJB 3 introduces AOP (aspect-oriented programming) in a lightweight, intuitive way using <i>interceptors</i> . This allows you to easily separate out crosscutting concerns such as logging and auditing, and to do so in a configurable way.
Web services	EJB 3 can transparently turn business components into Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) web services with minimal or no code changes.
Remoting	In EJB 3, you can make components remotely accessible without writing any code. In addition, EJB 3 enables client code to access remote components as if they were local components using dependency injection (DI).
Testing	You can easily unit- and integration-test any EJB component using embedded containers with frameworks like JUnit.

1.1.3 Layered architectures and EJB

Enterprise applications are designed to solve a unique type of problem and therefore share many common requirements. Most Enterprise applications have some kind of user interface, implement business processes, model a problem domain, and save data into a database. Because of these shared requirements, you can follow a common architecture or design principle for building Enterprise applications known as *patterns*.

For server-side development, the dominant pattern is *layered architectures*. In a layered architecture, components are grouped into tiers. Each tier in the application has a well-defined purpose, like a section of a factory assembly line. Each section of the assembly line performs its designated task and passes the remaining work down the line. In layered architectures, each layer delegates work to a layer underneath it.

EJB recognizes this fact and thus isn't a jack-of-all-trades, master-of-none component model. Rather, EJB is a specialist component model that fits a specific purpose in layered architectures. Layered architectures come in two predominant flavors: traditional four-tier architectures and domain-driven design (DDD). Let's take a look at each of these architectures and where EJB is designed to fit in them.

TRADITIONAL FOUR-TIER LAYERED ARCHITECTURE

Figure 1.1 shows the traditional four-tier server architecture. This architecture is pretty intuitive and enjoys a good amount of popularity. In this architecture, the *presentation layer* is responsible for rendering the graphical user interface (GUI) and handling user input. The presentation layer passes down each request for application functionality to the business logic layer. The *business logic layer* is the heart of the application and contains workflow and processing logic. In other words, business logic-layer components model distinct actions or processes that the application can perform, such as billing, search, ordering, and user account maintenance. The business logic layer retrieves data from and saves data into the database by utilizing the persistence

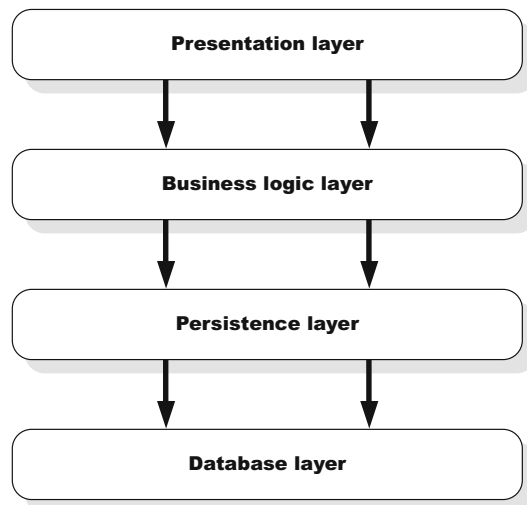


Figure 1.1 Most traditional Enterprise applications have at least four layers: the presentation layer is the actual user interface and can either be a browser or a desktop application; the business logic layer defines the business rules; the persistence layer deals with interactions with the database; and the database layer consists of a relational database such as Oracle database that stores the persistent objects.

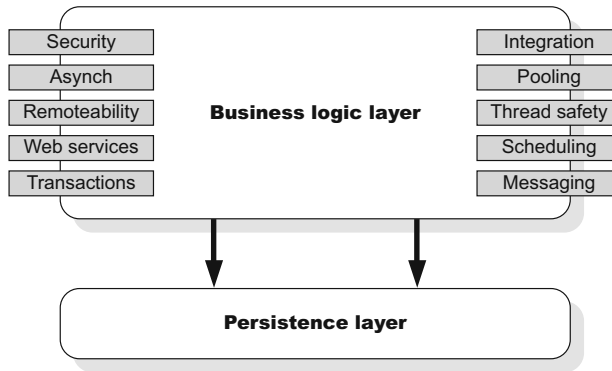


Figure 1.2 The component services offered by EJB 3 at the supported application layer. Note that each service is independent of the others, so you are (for the most part) free to pick the features important for your application.

tier. The *persistence layer* provides a high-level object-oriented (OO) abstraction over the database layer. The *database layer* typically consists of a relational database management system (RDBMS) like Oracle database, DB2 database, or SQL Server database.

EJB isn't a presentation layer or persistence-layer technology. It's all about robust support for implementing business logic-layer components for Enterprise applications. Figure 1.2 shows how EJB supports these layers via its services.

In a typical Java EE-based system, JSF and CDI will be used at the presentation tier, EJB will be used in the business layer, and JPA and CDI will be used in the persistence tier.

The traditional four-tier layered architecture isn't perfect. One of the most common criticisms is that it undermines the OO ideal of modeling the business domain as objects that encapsulate both data and behavior. Because the traditional architecture focuses on modeling business processes instead of the domain, the business logic tier tends to look more like a database-driven procedural application than an OO one. Because persistence-tier components are simple data holders, they look a lot like database record definitions rather than first-class citizens of the OO world. As you'll see in the next section, DDD proposes an alternative architecture that attempts to solve these perceived problems.

DOMAIN-DRIVEN DESIGN

Figure 1.3 shows domain-driven architecture. The term *domain-driven design* may be relatively new but the concept is not (see *Domain-Driven Design: Tackling Complexity in the Heart of Software*, by Eric Evans [Addison-Wesley Professional, 2003]). DDD emphasizes that domain objects should contain business logic and shouldn't just be dumb replicas of database records. Domain objects can be implemented as entities in JPA. With DDD, a Catalog object in a trading application might, in addition to having all the data of an entry in the catalog table in the database, know not to return catalog entries that aren't in stock. Being POJOs, JPA entities support OO features, such as inheritance and polymorphism. It's easy to implement a persistence object model with the JPA and to add business logic to your entities. Now, DDD still utilizes a *service layer* or *application layer* (see *Patterns of Enterprise Application Architecture*, by Martin Fowler

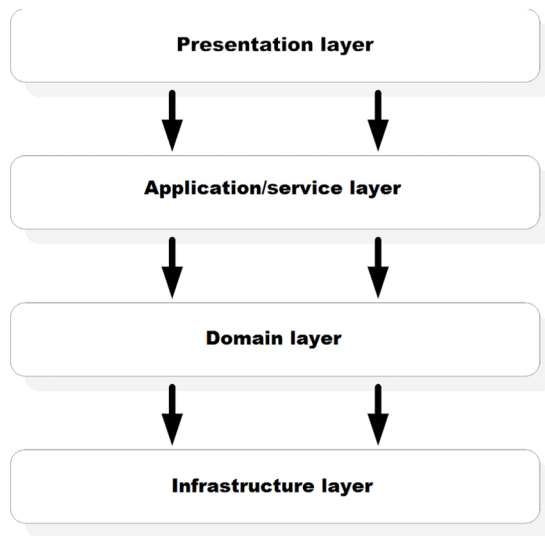


Figure 1.3 Domain-driven design typically has four or more layers. The presentation layer is responsible for the user interface and for interaction with the application/service layer. The application/service layer is typically very light and only allows communication between the presentation layer and the domain. The domain layer is the complex expression of your application data model consisting of entities, value objects, aggregates, factories, and repositories. The infrastructure layer gets to the database and other similar technology.

[Addison-Wesley Professional, 2002]). The application layer is similar to the business logic layer of the traditional four-tier architecture but much thinner. EJB works well as the service-layer component model. Whether you use the traditional four-tier architecture or a layered architecture with DDD, you can use entities to model domain objects, including modeling state and behavior. We'll discuss domain modeling with JPA entities in chapter 7.

Despite its impressive services and vision, EJB 3 isn't the only act in town. You can combine various technologies to more or less match EJB services and infrastructure. For example, you could use Spring with other open-source technologies such as Hibernate and AspectJ to build your application, so why choose EJB 3? Glad that you asked....

1.1.4 Why choose EJB 3?

At the beginning of this chapter, we hinted at EJB's status as a pioneering technology. EJB is a groundbreaking technology that raised the standards of server-side development. Just like Java itself, EJB changed things in ways that are here to stay and inspired many innovations. Up until a few years ago, the only serious competition to EJB came from the Microsoft .NET framework. In this section, we'll point out a few of the compelling EJB 3 features that we feel certain will have this latest version at the top of your short list.

EASE OF USE

Thanks to the unwavering focus on ease of use, EJB 3 is probably the simplest server-side development platform around. The features that shine the brightest are POJO programming, annotations in favor of verbose XML, heavy use of sensible defaults, and avoidance of complex paradigms. Although the number of EJB services is significant, you'll find them very intuitive. For the most part, EJB 3 has a practical outlook on things and doesn't demand that you understand the theoretical intricacies. In fact,

most EJB services are designed to give you a break from this mode of thinking so you can focus on getting the job done and go home at the end of the day knowing you accomplished something.

COMPLETE, INTEGRATED SOLUTION STACK

EJB 3 offers a complete stack of server-side solutions, including transactions, security, messaging, scheduling, remoting, web services, asynchronous processing, testing, dependency injection, and interceptors. This means that you won't have to spend a lot of time looking for third-party tools to integrate into your application. These services are also just there for you—you don't have to do anything to explicitly enable them. This leads to near-zero configuration systems.

In addition, EJB 3 provides seamless integration with other Java EE technologies, such as CDI, JPA, JDBC, JavaMail, Java Transaction API (JTA), JMS, Java Authentication and Authorization Service (JAAS), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), and so on. EJB is also guaranteed to seamlessly integrate with presentation-tier technologies like JSP, Servlets, and JSF. When needed, you can integrate third-party tools with EJB using CDI.

OPEN JAVA EE STANDARD

EJB is a critical part of the Java EE standard. This is an extremely important concept to grasp if you're to adopt EJB. EJB 3 has an open, public API specification and compatibility test kit that organizations are encouraged to use to create a container implementation. The EJB 3 standard is developed by the Java Community Process (JCP), consisting of a nonexclusive group of individuals driving the Java standard. The open standard leads to broader vendor support for EJB 3, which means you don't have to depend on a proprietary solution.

BROAD VENDOR SUPPORT

EJB is supported by a large and diverse variety of independent organizations. This includes the technology world's largest, most respected, and most financially strong names, such as Oracle and IBM, as well as passionate and energetic open-source groups like JBoss and Apache. Wide vendor support translates to three important advantages for you. First, you're not at the mercy of the ups and downs of a particular company or group of people. Second, a lot of people have concrete long-term interests in keeping the technology as competitive as possible. You can essentially count on being able to take advantage of the best-of-breed technologies both in and outside the Java world in a competitive timeframe. Third, vendors have historically competed against one another by providing value-added nonstandard features. All of these factors help keep EJB on the track of continuous healthy evolution.

CLUSTERING, LOAD BALANCING, AND FAILOVER

Features historically added by most application server vendors are robust support for clustering, load balancing, and failover. EJB application servers have a proven track record of supporting some of the largest high-performance computing (HPC)–enabled server farm environments. More importantly, you can use such support with

no changes to code, no third-party tool integration, and relatively simple configuration (beyond the inherent work in setting up a hardware cluster). This means that you can rely on hardware clustering to scale up your application with EJB 3 if you need to.

PERFORMANCE AND SCALABILITY

Enterprise applications have a lot in common with a house. Both are meant to last, often much longer than anyone expects. Being able to support high-performance, fault-tolerant, scalable applications is an upfront concern for the EJB platform instead of being an afterthought. Not only will you be writing good server-side applications faster, you can also expect your platform to grow as needed. You can support a larger number of users without having to rewrite your code; these concerns are taken care of by EJB container vendors via features like thread safety, distributed transactions, pooling, passivation, asynchronous processing, messaging, and remototing. You can count on doing minimal optimization or moving your application to a distributed, clustered server farm by doing nothing more than a bit of configuration.

We expect that by now you're getting jazzed about EJB and you're eager to learn more. So let's jump right in and see how you can use EJB to build the business logic tier of your applications, starting with the beans.

1.2 Understanding EJB types

In EJB-speak, a component is a *bean*. If your manager doesn't find the Java "coffee bean" play on words cute either, blame Sun's marketing department. Hey, at least you get to hear people in suits use the words "Enterprise" and "bean" in close sequence as if it were perfectly normal.

As we mentioned, EJB classifies beans into two types based on what they're used for:

- Session beans
- Message-driven beans

Each bean type serves a purpose and can use a specific subset of EJB services. The real purpose of bean types is to safeguard against overloading them with services that cross wires. This is kind of like making sure the accountant in the horn-rimmed glasses doesn't get too curious about what happens when you touch both ends of a car battery terminal at the same time. Bean classification also helps you understand and organize an application in a sensible way; for example, bean types help you develop applications based on a layered architecture. Let's start digging a little deeper into the various EJB component types, starting with session beans.

1.2.1 Session beans

A session bean is invoked by a client to perform a specific business operation, such as checking the credit history of a customer. The name *session* implies that a bean instance is available for the duration of a unit of work and doesn't survive a server crash or shutdown. A session bean can model any application logic functionality. There are three types of session beans: *stateful*, *stateless*, and *singleton*.

A stateful session bean automatically saves the bean state between invocations from a single, unique client without your having to write any additional code. The typical example of a state-aware process is the shopping cart for a web merchant like Amazon. Stateful session beans are either timed out or end their lifecycle when the client requests it. In contrast, stateless session beans don't maintain any state and model application services that can be completed in a single client invocation. You could build stateless session beans for implementing business processes such as charging a credit card or checking a customer's credit history. Singleton session beans maintain the state, are shared by all clients, and live for the duration of the application. You could use a singleton bean for a discount processing component since the business rules for applying discounts are usually fixed and shared across all clients. Note that singleton beans are a new feature added in EJB 3.1.

A session bean can be invoked either locally or remotely using Java RMI. A stateless or singleton session bean can also be exposed as a SOAP or REST web service.

1.2.2 Message-driven beans

Like session beans, message-driven beans (MDBs) process business logic. But MDBs are different in one important way: clients never invoke MDB methods directly. Instead, MDBs are triggered by messages sent to a messaging server, which enables sending asynchronous messages between system components. Some typical examples of messaging servers are HornetQ, ActiveMQ, IBM WebSphere MQ, SonicMQ, Oracle Advanced Queueing, and TIBCO. MDBs are typically used for robust system integration and asynchronous processing. An example of messaging is sending an inventory-restocking request from an automated retail system to a supply-chain management system. Don't worry too much about messaging right now; we'll get to the details later in this book.

1.3 Related specifications

EJB has two closely related specifications that we'll cover in this book. The first is JPA, which is the persistence standard for Java EE and CDI, and provides dependency injection and context management services to all Java EE components including EJB.

1.3.1 Entities and the Java Persistence API

EJB 3.1 saw JPA 2 moved from an EJB 3 API to a completely separate Java EE specification. But JPA has some specific runtime integration points with EJB because the specifications are so closely related. We'll say just a few things about JPA here because we have chapters dedicated to it.

Persistence is the ability to have data contained in Java objects automatically stored into a relational database like Oracle database, SQL server database, and DB2 database. Persistent objects are managed by JPA. It automatically persists Java objects using a technique called object-relational mapping (ORM). ORM is the process of mapping data held in Java objects to database tables using configuration or declaratively using annotations. It relieves you of the task of writing low-level, boring, and complex JDBC code to persist objects into a database.

An ORM framework performs transparent persistence by making use of ORM metadata that defines how objects are mapped to database tables. ORM isn't a new concept and has been around for a while. Oracle TopLink is probably the oldest ORM framework in the market; open-source framework JBoss Hibernate popularized ORM concepts within the mainstream developer community. Because JPA standardizes ORM frameworks for the Java platform, you can plug in an ORM product like JBoss Hibernate, Oracle TopLink, or Apache OpenJPA as the underlying JPA “persistence provider” for your application.

JPA isn't just a solution for server-side applications. Persistence is a problem that even a standalone Swing-based desktop application has to solve. This drove the decision to make JPA 2 a cleanly separated API in its own right that can be run outside an EJB 3 container. Much like JDBC, JPA is intended to be a general-purpose persistence solution for any Java application.

ENTITIES

Entities are the Java objects that are persisted into the database. While session beans are the “verbs” of a system, entities are the “nouns.” Common examples include an `Employee` entity, a `User` entity, and an `Item` entity. Entities are the OO representations of the application data stored in the database. Entities survive container crashes and shutdown. The ORM metadata specifies how the object is mapped to the database. You'll see an example of this in the next chapter. JPA entities support a full range of relational and OO capabilities, including relationships between entities, inheritance, and polymorphism.

ENTITYMANAGER

Entities tell a JPA provider how they map to the database, but they don't persist themselves. The `EntityManager` interface reads the ORM metadata for an entity and performs persistence operations. The `EntityManager` knows how to add entities to the database, update stored entities, and delete and retrieve entities from the database.

JAVA PERSISTENCE QUERY LANGUAGE

JPA provides a specialized SQL-like query language called Java Persistence Query Language (JPQL) to search for entities saved into the database. With a robust and flexible API such as JPQL, you won't lose anything by choosing automated persistence instead of handwritten JDBC. In addition, JPA supports native, database-specific SQL, in the rare cases where it's worth using.

1.3.2 Contexts and dependency injection for Java EE

Java EE 5 had a basic form of dependency injection that EJB could use. It was called *resource injection* and allowed you to inject container resources, such as data sources, queues, JPA resources, and EJBs, using annotations like `@EJB`, `@Resource`, and `@PersistenceContext`. These resources could be injected into Servlets, JSF backing beans, and EJB. The problem was that this was very limiting. You weren't able to inject EJB into Struts or JUnit, and you couldn't inject non-EJB DAOs (data access objects) or helper classes into EJB.

CDI is a powerful solution to the problem. It provides EJB (as well as all other APIs and components in the Java EE environment) best-of-breed, next-generation, generic dependency injection and context management services. CDI features include injection, automatic context management, scoping, qualifiers, component naming, producers, disposers, registry/lookup, stereotypes, interceptors, decorators, and events. Unlike many older dependency injection solutions, CDI is completely type-safe, compact, futuristic, and annotation-driven. We'll cover CDI in detail in chapter 12.

1.4 **EJB runtimes**

When you build a simple Java class, you need a Java Virtual Machine (JVM) to execute it. In a similar way (as you learned in section 1.3), to execute session beans and MDBs you need an EJB container. In this section we give you a bird's-eye view of the different runtimes that an EJB container may contain inside.

Think of the container as an extension of the basic idea of a JVM. Just as the JVM transparently manages memory on your behalf, the container transparently provides EJB component services such as transactions, security management, remotng, and web services support. You might even think of the container as a JVM on steroids, of which the purpose is to execute EJB. In EJB 3, the container provides services applicable only to session beans and MDBs. The task of putting an EJB 3 component inside a container is called *deployment*. Once an EJB is successfully deployed in a container, it can be used in your applications.

In the Java world, containers aren't limited to the realm of EJB 3. You're probably familiar with a web container, which allows you to run web-based applications using Java technologies such as Servlets, JSP, and JSF. A *Java EE container* is an application server solution that supports EJB 3, a web container, and other Java EE APIs and services. Oracle WebLogic server, GlassFish server, IBM WebSphere application server, JBoss application server, and Caucho Resin are examples of Java EE containers.

1.4.1 **Application servers**

Application servers are where EJBs have been traditionally deployed. Application servers are Java EE containers that include support for all Java EE APIs, as well as facilities for administration, deployment, monitoring, clustering, load balancing, security, and so on. In addition to supporting Java EE-related technologies, some application servers can also function as production-quality HTTP servers. Others support modularity via technologies like OSGi. As of Java EE 6, application servers can also come in a scaled-down, lightweight *Web Profile* form. The Web Profile is a smaller subset of Java EE APIs specifically geared toward web applications. Web Profile APIs include JSF 2.2, CDI 1.1, EJB 3.2 Lite (discussed in section 1.4.2), JPA 2.1, JTA 1.2, and bean validation. At the time of writing, GlassFish and Resin provided Java EE 7 Web Profile offerings. Note that Java EE 7 Web Profile implementations are free to add APIs as they wish. For example, Resin adds JMS, as well as most of the EJB API including messaging, remotng, and scheduling (but not EJB 2 backward compatibility). Figure 1.4 shows how the Web Profile compares with the complete Java EE platform.

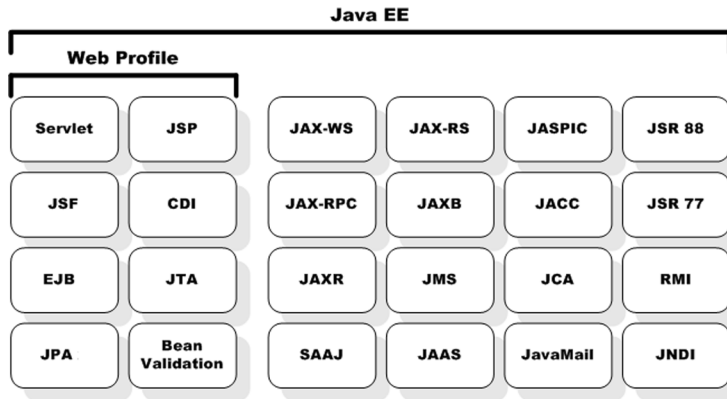


Figure 1.4 Java EE Web Profile versus full Java EE platform

The Web Profile defines a complete stack on which to build a modern web application. Web applications are now rarely written from the ground up using raw Servlets, but instead sit on top of JSF and make use of the various EE technologies.

1.4.2 EJB Lite

Similar to the idea of the Java EE 7 Web Profile, EJB 3.2 also comes in a scaled-down, lighter-weight version called EJB 3.2 Lite. EJB Lite goes hand-in-hand with the Web Profile and is intended for web applications. Just as with the Web Profile, any vendor implementing the EJB 3.2 Lite API is free to include EJB features as they wish. From a practical standpoint, the most important thing that EJB 3.2 Lite does is remove support for EJB 2 backward compatibility. This means that an EJB container can be much more lightweight because it doesn't have to implement the old APIs in addition to the lightweight EJB 3 model. Because EJB 3.2 Lite also doesn't include support for MDBs and remoting, it can mean a lighter-weight server if you don't need these features. For reference, table 1.2 compares the major EJB and EJB Lite features.

Table 1.2 EJB and EJB Lite feature comparison

Feature	EJB Lite	EJB
Stateless beans	✓	✓
Stateful beans	✓	✓
Singleton beans	✓	✓
Message-driven beans		✓
No interfaces	✓	✓
Local interfaces	✓	✓
Remote interfaces		✓
Web service interfaces		✓

Table 1.2 EJB and EJB Lite feature comparison (*continued*)

Feature	EJB Lite	EJB
Asynchronous invocation	✓	✓
Interceptors	✓	✓
Declarative security	✓	✓
Declarative transactions	✓	✓
Programmatic transactions	✓	✓
Timer service	✓	✓
EJB 2.x support		✓
CORBA interoperability		✓

1.4.3 Embeddable containers

Traditional application servers run as a separate process that you deploy your applications into. Embedded EJB containers, on the other hand, can be started through a programmatic Java API inside your own application. This is very important for unit testing with JUnit as well as using EJB 3 features in command-line or Swing applications. When an embedded container starts, it scans the class path of your application and automatically deploys any EJBs it can find. Figure 1.5 shows the architecture of an embedded EJB 3 container.

Embeddable containers have been around for a while. OpenEJB, EasyBeans, and Embedded JBoss are examples. Embeddable containers are only required to support EJB Lite, but most implementations are likely to support all features. For example, the embedded versions of GlassFish, JBoss, and Resin support all the features available on the application server. We'll discuss embedded containers in detail in chapter 15 on testing EJB 3.

1.4.4 Using EJB 3 in Tomcat

Apache Tomcat, the lightweight, popular Servlet container, doesn't support EJB 3, because unlike application servers, Servlet containers aren't required to support EJB. But you can easily use EJB 3 on Tomcat through embedded containers. The Apache

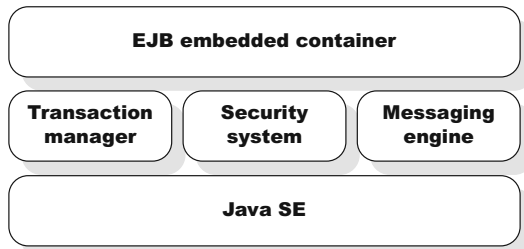


Figure 1.5 EJB 3.1 embedded containers run directly inside Java SE and provide all EJB services such as transactions, security, and messaging.

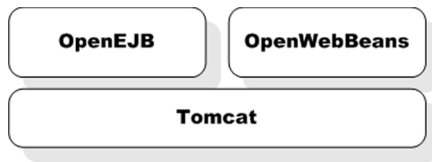


Figure 1.6 You can use OpenEJB and OpenWebBeans to enable both EJB and CDI on Tomcat.

OpenEJB project has specific support for enabling EJB 3 on Tomcat. As shown in figure 1.6, you can also enable CDI on Tomcat using Apache OpenWebBeans. OpenWebBeans and OpenEJB are closely related projects and work seamlessly together. In this way, you can use a majority of Java EE 7 APIs on Tomcat if you wish.

1.5 Brave new innovations

From this point onward, let's start getting a little down and dirty and seeing what the brave new world of EJB 3 looks like in code. We'll note the primary distinguishing features of EJB 3 along the way.

1.5.1 "Hello User" example

"Hello World" examples have ruled the world since they first appeared in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice Hall PTR, 1988). "Hello World" caught on and held ground for good reason. It's very well suited to introducing a technology as simply and plainly as possible. The code samples for this book will use Maven, and if you're using Eclipse, you'll find it useful to have the m2eclipse plug-in installed to integrate Eclipse with Maven.

In 2004, one of the authors, Debu Panda, wrote an article for the TheServerSide.com in which he stated that when EJB 3 was released, it would be so simple you could write a "Hello World" in it using only a few lines of code. Any experienced EJB 2 developer knows that this couldn't be done easily in EJB 2. You had to write a home interface, a component interface, a bean class, and a deployment descriptor. Well, let's see if Debu was right in his prediction, as shown in the following listing.

Listing 1.1 HelloUser session bean

```

package ejb3inaction.example;
import javax.ejb.Stateless;

@Stateless
public class HelloUserBean implements HelloUser {
    public String sayHello(String name) {
        return String.format("Hello %s welcome to EJB 3.1!", name);
    }
}
  
```

① Stateless annotation

② HelloUserBean POJO

This listing is a complete and working EJB! The bean class is a POJO ①, without even an interface. EJB 3.1 introduced the no-interface view. Before this, EJB required an interface to indicate which methods should be visible. The no-interface view essentially says that all public methods in your bean will be available for invocation. It's easy to

use, but you need to pick your public methods carefully. The `exposeAllTheCompanyDirtySecrets()` method should probably be private. The funny `@Stateless` symbol in listing 1.1 is a metadata annotation ❷ that converts the POJO to a full-powered stateless EJB. In effect, they're "comment-like" configuration information that can be added to Java code.

EJB 3 enables you to develop an EJB component using POJOs that know nothing about platform services. You can apply annotations to these POJOs to add platform services such as remoteability, web services support, and lifecycle callbacks as needed.

To execute this EJB, you have to deploy it to the EJB container. If you want to execute this sample, download the `actionbazaar-snapshot-2.zip` from <https://code.google.com/p/action-bazaar/>, extract "chapter1 project" to build, and install it using Maven and the embedded GlassFish server.

We're going to analyze a lot of code in this book—some just as easy as this. You could trigger the hello as a web service by simply adding the `@WebService` annotation. You could inject a resource, like a helper bean that will translate hello into foreign languages, with `@Inject`. What do you want to do with EJB? If you keep reading, we'll probably tell you how to do it.

1.5.2 Annotations versus XML

Prior to annotations (introduced in Java SE 5), XML was the only logical choice for application configuration because there were no other viable options around, except for tools like XDoclet, which was popular in many relatively progressive EJB 2 shops.

The problems with XML are myriad. XML is verbose, not that readable, and extremely error-prone. XML also takes no advantage of Java's unique strength in strong type safety. Lastly, XML configuration files tend to be monolithic and they separate the information about the configuration from the Java code that uses them, making maintenance more difficult. Collectively, these problems are named XML Hell and annotations are specifically designed to be the cure.

EJB 3 was the first mainstream Java technology to pave the way for annotation adoption. Since then, many other tools like JPA, JSF, Servlets, JAX-WS, JAX-RS, JUnit, Seam, Guice, and Spring have followed suit.

As you can see in the code example in listing 1.1, annotations are essentially property settings that mark a piece of code, such as a class or method, as having particular attributes. When the EJB container sees these attributes, it adds the container services that correspond to it. This is called *declarative-style programming*, where the developer specifies what should be done and the system adds the code to do it behind the scenes.

In EJB 3, annotations dramatically simplify development and testing of applications. Developers can declaratively add services to EJB components when they need to. As figure 1.7 depicts, an annotation basically transforms a simple POJO into an EJB, just as the `@Stateless` annotation does in the example.



Figure 1.7 EJBs are regular Java objects that may be configured using metadata annotations.

While XML has its problems, it can be beneficial in some ways. It can be easier to see how the system components are organized by looking at a centralized XML configuration file. You can also configure the same component differently per deployment or configure components whose source code you can't change. Configuration that has little to do with Java code is also poorly expressed in annotations. Examples of this include port/URL configuration, file locations, and so on. The good news is that you can use XML with EJB 3. You can even use XML to override or augment annotation-based configuration. Unless you have a very strong preference for XML, it's generally advisable to start with annotations and use XML overrides where they're really needed.

1.5.3 Intelligent defaults versus explicit configuration

EJB takes a different approach to default behavior than most frameworks such as Spring. With Spring, for example, if you don't ask, you don't get. You have to ask for any behavior you want to have in your Spring components. In addition to making the task of configuration easier via annotations, EJB 3 reduces the total amount of configuration altogether by using sensible defaults wherever possible. For example, the "Hello World" component is automatically thread-safe, pooled, and transactional without you having to do anything at all. Similarly, if you want scheduling, asynchronous processing, remoting, or web services, all you need to do is add a few annotations to the component. There's no service that you'll need to understand, explicitly enable, or configure—everything is enabled by default. The same is true of JPA and CDI as well. Intelligent defaulting is especially important when you're dealing with automated persistence using JPA.

1.5.4 Dependency injection versus JNDI lookup

EJB 3 was reengineered from the ground up for dependency injection. This means that you can inject EJBs into other Java EE components and inject Java EE components into EJBs. This is especially true when using CDI with EJB 3. For example, if you want to access the `HelloUser` EJB in listing 1.1 from another EJB, Servlet, or JSF backing bean, you could use code like this:

```
@EJB
private HelloUserBean helloUser;

void hello() {
    helloUser.sayHello("Curious George");
}
```

← 1 EJB injection

Isn't that great? The `@EJB` annotation ❶ transparently "injects" the `HelloUserBean` EJB into the annotated variable. The `@EJB` annotation reads the type and name of the

EJB and looks it up from JNDI under the hood. All EJB components are automatically registered with JNDI while being deployed. Note that you can still use JNDI lookups where they're unavoidable. For example, to dynamically look up your bean, you could use code like this:

```
Context context = new InitialContext();
HelloUserBean helloUser = (HelloUserBean)
    context.lookup("java:module/HelloUserBean");
helloUser.sayHello("Curious George");
```


We'll talk in detail about EJB injection and lookup in chapter 5.

1.5.5 **CDI versus EJB injection**

EJB-style injection predates CDI. Naturally, this means that CDI injection adds a number of improvements over EJB injection. Most importantly, CDI can be used to inject almost anything. EJB injection, on the other hand, can only be used with objects stored in JNDI, such as EJB, as well as some container-managed objects like the EJB context. CDI is far more type-safe than EJB. Generally speaking, CDI is a superset of EJB injection. For example, you can use CDI to inject the EJB as follows:

```
@Inject
private HelloUserBean helloUser;

void hello(){
    helloUser.sayHello("Curious George");
}
```



It might seem that CDI should be used for all Java EE injections, but it currently has a limitation. Although CDI can retrieve an EJB by type, it doesn't work with remote EJBs. EJB injection (`@EJB`) will recognize whether an EJB is local or remote and return the appropriate type. You should use CDI for injection when possible.

1.5.6 **Testable POJO components**

Because all EJBs are simply POJOs, you can easily unit test them in JUnit for basic component functionality. You can even use CDI to inject EJBs directly into unit tests, wire mock objects, and so on. Thanks to embedded containers, you can even perform full integration testing of EJB components from JUnit. Projects like Arquillian focus specifically on integrating JUnit with embedded containers. The following listing shows how Arquillian allows you to inject EJBs into JUnit tests.

Listing 1.2 EJB 3 unit testing with Arquillian

```
package ejb3inaction.example;

import javax.ejb.EJB;
import org.jboss.arquillian.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.Archive;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
```

```

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(Arquillian.class)
public class HelloUserBeanTest {
    @EJB
    private HelloUser helloUser;

    @Deployment
    public static Archive<?> createDeployment() {
        return ShrinkWrap.create(JavaArchive.class, "foo.jar")
            .addClasses(HelloUserBean.class);
    }

    @Test
    public void testSayHello() {
        String helloMessage = helloUser.sayHello("Curious George");
        Assert.assertEquals(
            "Hello Curious George welcome to EJB 3.1!", helloMessage);
    }
}

```

Running JUnit with Arquillian

Injecting bean to be tested

Using EJB in unit test

We've dedicated chapter 15 in its entirety to testing EJB components.

1.6 Changes in EJB 3.2

The goal of 3.2 is to continue to evolve the EJB specification to be a complete solution for all Enterprise business needs and to improve the EJB architecture by reducing its complexity from the developer's point of view. In this section we'll briefly talk about the particular changes in EJB 3.2.

1.6.1 Previous EJB 2 features now optional

Support for EJB 2 has been made optional by EJB 3.2. This means a fully compliant Java EE 7 application server no longer needs to support EJB 2-style entities beans. EJB QL and JAX-RPC have also been made optional.

1.6.2 Enhancements to message-driven beans

In EJB 3.2, MDBs have been given a major overhaul. The update to JMS 2.0 brings a simplified API as well as integration with advances to Java in other areas such as dependency injection with CDI. It also makes using the `javax.jms.MessageListener` interface optional, giving you the ability to create an MDB with a no-methods listener interface, which makes public methods of the class message listener methods. Here's a quick look at the simplified API for MDB.

Send a message:

```

@Inject @JMSConnectionFactory("jms/QueueConnectionFactory")
private JMSContext context;

@Resource(lookup="jms/MessageQueue")
private Queue queue;

```

Injects JMSContext

Injects destination

```
public void sendMessage(String txtToSend) {
    context.createProducer().send(queue, txtToSend);
}
```

← Uses simplified API
to send message

Receive a message:

```
@MessageDriven(mappedName="jms/BidQueue")
public class BidMdb implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;

    public void onMessage(Message inMessage) {
        // handle message
    }
}
```

← Annotated as an MDB for the
container, listen on "jms/BidQueue"

← Class implements **MessageListener**
to receive JMS messages

← Injects a **MessageDrivenContext**
if needed

← Method implements **MessageListener**
and handles message

1.6.3 Enhancements to stateful session beans

In EJB 3.2, session beans haven't changed dramatically, at least not as dramatically as MDBs. A few enhancements have been made to stateful session beans concerning passivation and transaction support. Here's a quick look at these enhancements.

DISABLE PASSIVATION

Prior to EJB 3.2, stateful beans needed all objects in them to implement `Serializable` so that the EJB container could passivate a bean without error. If one object in your stateful bean wasn't serializable, passivation would fail and the EJB container would destroy the bean, losing the state. Although it's highly advisable to make sure your stateful beans are serializable so the EJB container can provide you with services like passivation and clustered failover, sometimes it's not possible. To prevent the container from attempting to passivate a stateful bean you don't want to passivate, use the `passivationCapable` element:

```
@Stateful(passivationCapable=false)
public class BiddingCart {
}
```

← Prevents EJB container from
passivating stateful bean

TRANSACTION SUPPORT TO LIFECYCLE CALLBACKS

Prior to EJB 3.2, stateful session beans had lifecycle callback methods, but it was undefined how transactions were supported during these method calls. So the new API adds transactional support to the lifecycle callback methods by introducing the ability to annotate the lifecycle callback methods with `@TransactionalAttribute(REQUIRES_NEW)`. `REQUIRES_NEW` is the only valid value for stateful lifecycle callback methods:

```
@Stateful
public class BiddingCart {
    @PostConstruct
    @TransactionalAttribute(REQUIRES_NEW)
    public void lookupDefaults() {}
}
```

← Stateful
session bean

← **@PostConstruct** lifecycle
method `lookupDefaults` is
run in its own transaction

1.6.4 Simplifying local interfaces for stateless beans

Prior to 3.2, if interfaces weren't marked as `@Local` or `@Remote`, then the implementing bean was forced to define them. Here's what the pre-3.2 interface and bean looked like:

```
public interface A {}
public interface B {}

@Stateless
@Local({A.class, B.class})
public class BidServicesBean implements A, B {}
```

Interfaces A and B don't specify
`@Local` or `@Remote`

❶ Prior to EJB 3.2,
`@Local` must be used
to define both A and B
as local interfaces

Now EJB 3.2 has more intelligent defaults. By default, all interfaces that don't specify `@Local` or `@Remote` automatically become local interfaces by the container. So you can skip line ❶ and rewrite the code like this:

```
@Stateless
public class BidServicesBean implements A, B {}
```

Container will now default A
and B to local interfaces

1.6.5 Enhancements in TimerService API

The `TimerService` API has been enhanced to expand the scope of where and how timers may be retrieved. Prior to EJB 3.2, the `Timer` and `TimerHandler` objects could only be accessed by the bean that owned the timer. This restriction has been lifted, and a new API method called `getAllTimers()` has been added that will return a list of all active timers in the EJB module. This allows any code to view all timers and have the ability to alter them.

1.6.6 Enhancements in EJBContainer API

For EJB 3.2, a couple of changes have been made to the embeddable `EJBContainer` API. First, the API now implements `AutoCloseable` so it may be used with a `try-with-resources` statement:

```
try (EJBContainer c = EJBContainer.createEJBContainer();) {
    // work with container
}
```

Second, the embeddable `EJBContainer` object is required to support the EJB Lite group of the EJB API. EJB API groups will be discussed further in the next section.

1.6.7 EJB API groups

Because EJB technology is the backbone of Enterprise Java development, EJBs need to be able to provide a large number of services to fulfill business needs. These services include but aren't limited to transactions, security, remote access, synchronous and asynchronous execution, and state tracking, and the list goes on. Not all Enterprise solutions require all the services EJBs are able to provide. To help streamline usage, EJB API groups were created for EJB 3.2. EJB API groups are well-defined subsets of the capabilities of EJBs created for specific purposes. The groups defined in the EJB 3.2 specification are these:

- EJB Lite
- Message-driven beans
- EJB 3.x Remote
- Persistent EJB timer services
- JAX-WS Web Service endpoints
- Embeddable EJB container (optional)
- EJB 2.x API
- Entity beans (optional)
- JAX-RPC Web Service endpoints (optional)

Except for the few groups that are optional, a full EJB container is required to implement all of the groups. The most important of these is the EJB Lite group. The EJB Lite group consists of the minimum number of EJB features that's still powerful enough to handle the majority of business transactions and security needs. This makes an EJB Lite implementation ideal to embed into a Servlet container like Tomcat to give the container some Enterprise features, or you can embed it into your Android tablet application to handle its data needs.

Now that we've looked at some of the new features and changes made to EJB 3.2, let's see how EJB technology compares with other frameworks in the marketplace that are also attempting to provide solutions for Enterprise Java software development.

1.7 Summary

You should now have a good idea of what EJB 3 is, what it brings to the table, and why you should consider using it to build server-side applications. We gave you an overview of the new features in EJB 3, including these important points:

- EJB 3 components are POJOs that are configurable through simplified meta-data annotations.
- Accessing EJB from client applications and unit tests has become very simple using dependency injection.
- EJB provides a powerful, scalable, complete set of Enterprise services out-of-the-box.

We also provided a taste of code to show how EJB 3 addresses development pain points. Armed with this essential background, you're probably eager to look at more code. We aim to satisfy this desire, at least in part, in the next chapter. Get ready for a whirlwind tour of the EJB 3 API that shows just how easy the code really is.

EJB 3 IN ACTION, Second Edition

Panda • Rahman • Cuprak • Remijan

The EJB 3 framework provides a standard way to capture business logic in manageable server-side modules, making it easier to write, maintain, and extend Java EE applications. EJB 3.2 provides more enhancements and intelligent defaults and integrates more fully with other Java technologies, such as CDI, to make development even easier.

EJB 3 in Action, Second Edition is a fast-paced tutorial for Java EE business component developers using EJB 3.2, JPA, and CDI. It tackles EJB head-on through numerous code samples, real-life scenarios, and illustrations. Beyond the basics, this book includes internal implementation details, best practices, design patterns, performance tuning tips, and various means of access including Web Services, REST Services, and WebSockets.

What's Inside

- Fully revised for EJB 3.2
- POJO persistence with JPA 2.1
- Dependency injection and bean management with CDI 1.1
- Interactive application with WebSocket 1.0

Readers need to know Java. No prior experience with EJB or Java EE is assumed.

Debu Panda, Reza Rahman, Ryan Cuprak, and Michael Remijan are seasoned Java architects, developers, authors, and community leaders. Debu and Reza coauthored the first edition of *EJB 3 in Action*.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/EJB3inActionSecondEdition

“The ultimate tutorial for EJB 3.”

—Luis Peña, HP

“Reza sits on the EJB 3.2 Expert Group and is a great instructor. That’s all you need to know.”

—John Griffin, Progrexion ASG

“Thoroughly and clearly explains how to leverage the full power of the JEE platform.”

—Rick Wagner, Red Hat, Inc.

“Provides a rock-solid foundation to EJB novice and expert alike.”

—Jeet Marwah, gen-E

“If you have EJB troubles, this is your cure.”

—Jürgen De Commer, Imtech ICT



ISBN 13: 978-1-935182-99-3
ISBN 10: 1-935182-99-4



9 781935 182993