SAMPLE CHAPTER

EJB3 IN ACTION

Debu Panda
Reza Rahman
Derek Lane

MANNING

EJB 3 in Action

by Debu Panda

Reza Rahman

Derek Lane

Sample Chapter 1

brief contents

1 What's what in EJB 3

This chapter covers

- EJB as component and framework
- Types of EJBs
- EJB containers and persistence provider
- Reasons for choosing EJB 3

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 will like being a cow the best because I will 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 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 just about as heavy as the largest of our beloved pachyderms. The brave souls who could not 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 is sacred for Hindus and respected as a mother whose milk feeds us well.

Many people have put in a lot of hard work to make EJB 3 as simple and lightweight as possible without sacrificing enterprise-ready power. EJB components are now little more than plain old Java objects (POJOs) that look a lot like code in a Hello World program. We hope you agree with us as you read through the next chapters that EJB 3 has all the makings of a star.

We've strived to keep this book as practical as possible without skimping on content. The book is designed to help you learn EJB 3 as quickly and easily as possible. At the same time, we won't neglect the basics where needed. We'll also dive into deep waters with you where we can, share with you all the amazing sights we've discovered, and warn you about any lurking dangers.

This book is about the radical transformation of an important and uniquely influential technology in the Java world. We suspect you are not picking this book up to learn too much about EJB 2. You probably either already know EJB or are completely new to the world of EJB. In either case, spending too much time on previous versions is a waste of your time—you won't be surprised to learn that EJB 3 and EJB 2 have very little in common. If you are curious about the journey that brought us to the current point, we encourage you to pick up one of the many good books on the previous versions of EJB.

Our goal in this chapter is to tell you what's what in EJB 3, explain why you should consider using it, and, for EJB 2 veterans, outline the significant improvements the newest version offers. We'll then jump right into code in the next chapter to build on the momentum of this one. With these goals in mind, let's now start with a broad overview of EJB.

1.1 EJB overview

In very straightforward terms, Enterprise JavaBeans (EJB) is a platform for building portable, reusable, and scalable business applications using the Java programming language. Since its initial incarnation, EJB has been touted as a component model or framework that lets you build enterprise Java applications without having to reinvent services such as transactions, security, automated persistence, and so on that you may need for building an application. EJB allows application developers to focus on building business logic without having to spend time on building infrastructure code.

From a developer's point of view, an EJB is a piece of Java code that executes in a specialized runtime environment called the *EJB container*, which provides a number of component services. The persistence services are provided by a specialized framework called the *persistence provider*. We'll talk more about the EJB container, persistence provider, and services in section 1.3.

In this section, you'll learn how EJB functions as both a component and a framework. We'll also examine how EJB lends itself to building layered applications. We'll round off this section by listing a few reasons why EJB might be right for you.

1.1.1 EJB as a component

In this book, when we talk about EJBs, we are referring to the server-side *components* that you can use to build parts of your application, such as the business logic or persistence code. Many of us tend to associate the term *component* with developing complex and heavyweight CORBA, 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 remain invisible until they are needed and don't distract from the real purpose of the component. You will see this firsthand throughout this book, especially starting with chapter 2.

The real idea behind a component is that it should effectively encapsulate application behavior. The users of a component aren't required to know its inner workings. All they need to know is what to pass in and what to expect back.

There are three types of EJB components: session beans, message-driven beans, and entities. Session beans and message-driven beans are used to implement business logic in an EJB application, and entities are used for persistence.

Components can be reusable. For instance, suppose you're in charge of building a website for an online merchant that sells technology books. You implement a module to charge the credit card as part of a regular Java object. Your company does fairly well, and you move on to greener pastures. The company then decides to diversify and begins developing a website for selling CDs and DVDs. Since the deployment environment for the new site is different, it can't be located on the same server as your module. The person building the new site is forced to duplicate your credit card module in the new website because there's no easy way to access your module. If you had instead implemented the credit card–charging module as an EJB component as shown in figure 1.1 (or as a web service), it would have been much easier for the new person to access it by simply making a call to it when she needed that functionality. She could have reused it without having to understand its implementation.

Given that, building a reusable component requires careful planning because, across enterprise applications within an organization, very little of the business logic may be reusable. Therefore, you may not care about the reusability of EJB components, but EJB still has much to offer as a framework, as you'll discover in the next section.

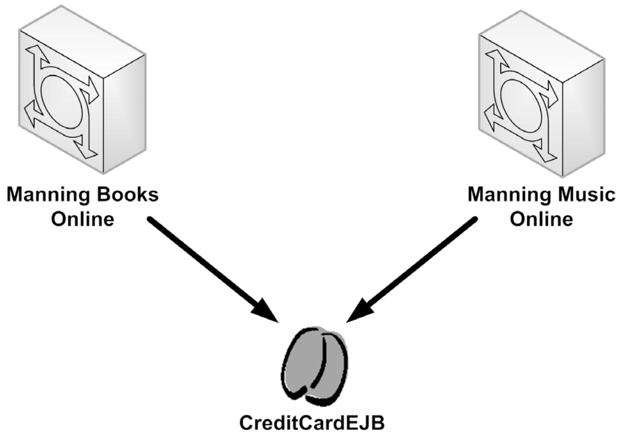

Figure 1.1
EJB allows development of reusable components. For example, you can implement the credit card–charging module as an EJB component that may be accessed by multiple applications.

1.1.2 EJB as a framework

As we mentioned, EJB components live in a container. Together, the components, or EJBs, and the container can be viewed as a framework that provides valuable services for enterprise application development.

Although many people think EJBs are overkill for developing relatively simple web applications of moderate size, nothing could be further from the truth. When you build a house, you don't build everything from scratch. Instead, you buy materials or even the services of a contractor as you need it. It isn't too practical to build an enterprise application from scratch either. Most server-side applications have a lot in common, including churning business logic, managing application state, storing and retrieving information from a relational database, managing transactions, implementing security, performing asynchronous processing, integrating systems, and so on.

As a framework, the EJB container provides these kinds of common functionality as out-of-the-box *services* so that your EJB components can use them in your applications without reinventing the wheel. For instance, let's say that when you built the credit card module in your web application, you wrote a lot of complex and error-prone code to manage transactions and security access control. You could have avoided that by using the declarative transaction and security services provided by the EJB container. These services, as well as many others you'll learn about in section 1.3, are available to the EJB components when they are deployed in the EJB container, as you can see in figure 1.2. This means writing high-quality, feature-rich applications much faster than you might think.

The container provides the services to the EJB components in a rather elegant new way: metadata annotations are used to preconfigure the EJBs by specifying the type of services to add when the container deploys the EJBs. Java 5 introduced

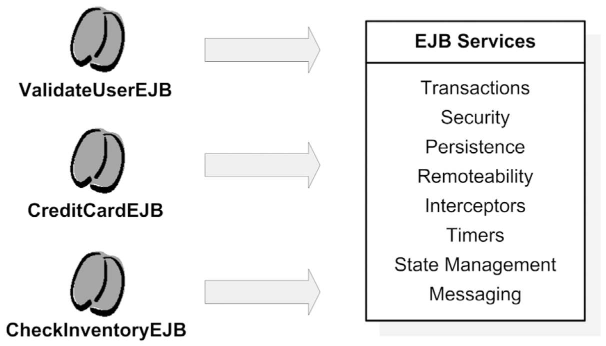

Figure 1.2
EJB as a framework provides services to EJB components.

metadata annotations, which are property settings that mark a piece of code, such as a class or method, as having particular attributes. This is a declarative style of programming, in which the developer specifies what should be done and the system adds the code to do it.

In EJB, metadata annotations dramatically simplify development and testing of applications, without having to depend on an external XML configuration file. It allows developers to declaratively add services to EJB components as and when they need. As figure 1.3 depicts, an annotation transforms a simple POJO into an EJB.

As you'll learn, annotations are used extensively throughout EJB, and not only to specify services. For example, an annotation can be used to specify the type of the EJB component.

Although it's sometimes easy to forget, enterprise applications have one more thing 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 up-front concern for the EJB platform instead of being an afterthought. Not only will you be writing good server-side applications faster, but also you can expect your platform to grow with the success of your application. When the need to support a larger number of users becomes a reality, you won't have to rewrite your code. Thankfully these concerns are taken care of by EJB container vendors. You'll be able to count on moving your application to a distributed, clustered server farm by doing nothing more than a bit of configuration.

Last, but certainly not least, with a world that's crazy about service-oriented architecture (SOA) and interoperability, EJB lets you turn your application into a web services powerhouse with ease when you need to.

The EJB framework is a standard Java technology with an open specification. If it catches your fancy, you can check out the real deal on the Java Community Process (JCP) website at www.jcp.org/en/jsr/detail?id=220. EJB is supported by a large number of companies and open source groups with competing but compatible implementations. On the one hand, this indicates that a large group of people will work hard to keep EJB competitive. On the other hand, the ease of portability

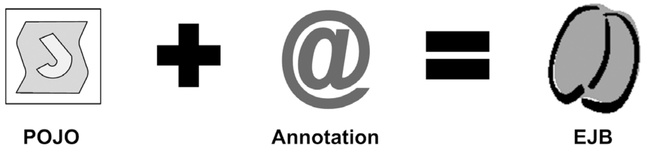

Figure 1.3
EJBs are regular Java objects that may be configured using metadata annotations.

means that you get to choose what implementation suits you best, making your application portable across EJB containers from different vendors.

Now that we've provided a high-level introduction to EJB, let's turn our attention to how EJB can be used to build layered applications.

1.1.3 Layered architectures and EJB

Most enterprise applications contain a large number of components. Enterprise applications are designed to solve a unique type of customer problem, but they share many common characteristics. For example, most enterprise applications have some kind of user interface and they implement business processes, model a problem domain, and save data into a database. Because of these commonalities, you can a 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, sort of like a profession but more 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 allows you to build applications using two different layered architectures: the traditional four-tier architecture and domain-driven design (DDD). Let's take a brief look at each of these architectures.

Traditional four-tier layered architecture

Figure 1.4 shows the traditional four-tier server architecture. This architecture is pretty intuitive and enjoys a wide 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 the application can perform, such as billing, searching, ordering, and user account maintenance. The business logic layer retrieves data from and saves data into the database by utilizing the persistence 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, DB2, or SQL Server.

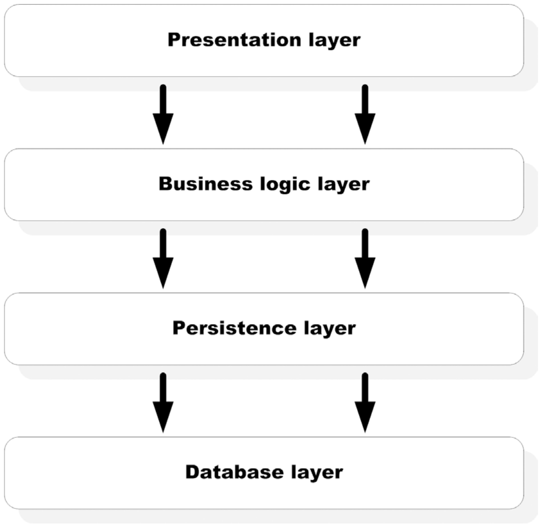

Figure 1.4
Most traditional enterprise applications have at least four layers. 1) The presentation layer is the actual user interface and can either be a browser or a desktop application. 2) The business logic layer defines the business rules. 3) The persistence layer deals with interactions with the database. 4) The database layer consists of a relational database such as Oracle that stores the persistent objects.

EJB is obviously not a presentation layer technology. EJB is all about robust support for implementing the business logic and persistence layers. Figure 1.5 shows how EJB supports these layers via its services.

In section 1.3 we'll go into more detail on EJB services. And in section 1.2 we'll explore EJB bean types. For now, just note that the bean types called session beans and message-driven beans (MDBs) reside in and use the services in the

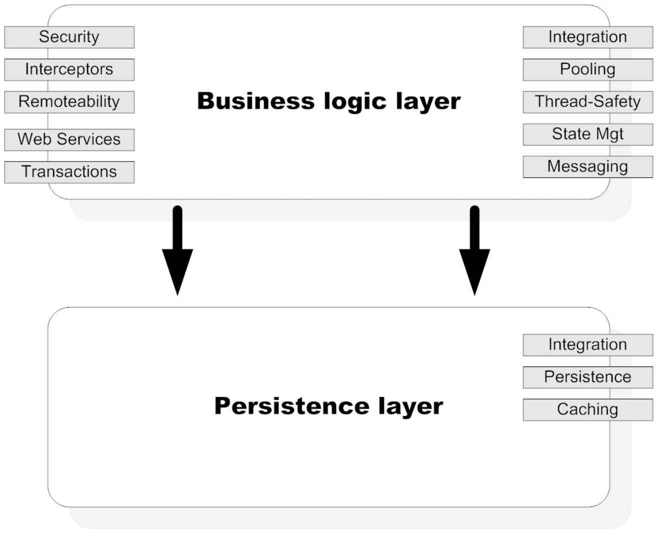

Figure 1.5
The component services offered by EJB 3 at each supported application layer. Note that each service is independent of the other, so you are for the most part free to pick the features important for your application. You'll learn more about services in section 1.3.

business logic tier, and the bean types called entities reside in and use services in the persistence tier.

The traditional four-tier layered architecture is not 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. Since 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

The term *domain-driven design* (DDD) 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 should not just be a dumb replica of database records. Domain objects are known as entities in EJB 3 (see section 1.2 for a discussion on entities). With DDD, the `Catalog` and `Customer` objects in a trading application are typical examples of entities, and they may contain business logic.

In his excellent book *POJOs in Action* (Manning, 2006), author Chris Richardson points out the problem in using domain objects just as a data holder.

> Some developers still view persistent objects simply as a means to get data in and out of the database and write procedural business logic. They develop what Fowler calls an "anemic domain model".... Just as anemic blood lacks vitality, anemic object models only superficially model the problem and consist of classes that implement little or no behavior.

Yet, even though its value is clear, until this release of EJB, it was difficult to implement DDD. Chris goes on to explain how EJB 2 encouraged procedural code:

> … J2EE developers write procedural-style code [because] it is encouraged by the EJB architecture, literature, and culture, which place great emphasis on EJB components. EJB 2 components are not suitable for implementing an object model.

Admittedly, implementing a real domain model was almost impossible with EJB 2 because beans were not POJOs and did not support many OO features. such as inheritance and polymorphism. Chris specifically targets entity beans as the problem:

EJB 2 entity beans, which are intended to represent business objects, have numerous limitations that make it extremely difficult to use them to implement a persistent object model.

The good news is that EJB 3 enables you to easily follow good object-oriented design or DDD. The entities defined by EJB 3 Java Persistence API (JPA) support OO features, such as inheritance or polymorphism. It's easy to implement a persistence object model with the EJB 3 JPA. More importantly, you can easily add business logic to your entities, so that implementing a rich domain model with EJB 3 is a trivial task.

Note, though, that many people don't like adding complex business logic in the domain object itself and prefer creating a layer for procedural logic referred to as the *service layer* or *application layer* (see *Patterns of Enterprise Application Architecture,* by Martin Fowler [Addison-Wesley Professional, 2002]). The application layer is similar to the business logic layer of the traditional four-tier architecture, but is much thinner. Not surprisingly, you can use session beans to build the service layer. 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 is not 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 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 has raised the standards of server-side development. Just like Java itself, EJB has changed things in ways that are here to stay and inspired many innovations. In fact, 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 JPA, all of which you will be learning about in this book. Although the number of EJB services is significant, you'll find them very intuitive. For the most part, EJB 3 has a practical outlook 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.

Integrated solution stack

EJB 3 offers a complete stack of server solutions, including persistence, messaging, lightweight scheduling, remoting, web services, dependency injection (DI), 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. In addition, EJB 3 provides seamless integration with other Java EE technologies, such as JDBC, JavaMail, Java Transaction API JTA (JTA), Java Messaging Service (JMS), Java Authentication and Authorization Service (JAAS), Java Naming and Directory Interface (JNDI), Java Remote Method Invocation (RMI), and so on. EJB is also guaranteed to seamlessly integrate with presentation-tier technologies like JavaServer Pages (JSP), servlets, JavaServer Faces (JSF), and Swing.

Open Java EE standard

EJB is a critical part of the Java EE standard. This is an extremely important concept to grasp if you are to adopt EJB 3. EJB 3 has an open, public API specification, which organizations are encouraged to use to create a container or persistence provider 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 Geronimo.

Wide vendor support translates to three important advantages for you. First, you are 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 to keep 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.

Stable, high-quality code base

Although EJB 3 is a groundbreaking step, most application server implementations will still benefit from a relatively stable code base that has lived through some of the most demanding enterprise environments over a prolonged period of time. Most persistence provider solutions like JDO, Hibernate, and TopLink are also stable products that are being used in many mission-critical production environments. This means that although EJB 3 is very new, you can expect stable implementations relatively quickly. Also, because of the very nature of standards-based development, the quality of EJB 3 container implementations is generally not taken lightly by vendors. To some degree, this helps ensure a healthy level of inherent implementation quality.

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 leverage 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.

EJB 3 is a compelling option for building enterprise applications. In the following sections, we explain more about EJB types and how to use them. We also discuss containers and persistence providers and explore the services they provide. By the time you finish reading sections 1.2 and 1.3, you'll have a good idea of what EJBs are and where they run, and what services they offer. So let's get started!

1.2 Understanding EJB types

If you're like most developers, you always have a tight deadline to meet. Most of us try to beg, borrow, or steal reusable code to make our lives easier. Gone are those days when developers had the luxury to create their own infrastructure when building a commercial application. While several commercial and open source frameworks are available that can simplify application development, EJB is a compelling framework that has a lot to offer.

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

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 we 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 three types, based on what they are used for:

- Session beans
- Message-driven beans
- Entities

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 akin to 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.

As we've briefly mentioned, session beans and message-driven beans (MDBs) are used to build business logic, and they live in the *container*, which manages these beans and provides services to them. Entities are used to model the persistence part of an application. Like the container, it is the *persistence provider* that manages entities. A persistence provider is pluggable within the container and is abstracted behind the Java Persistence API (JPA). This organization of the EJB 3 API is shown in figure 1.6.

We'll discuss the container and the persistence provider in section 1.3. For the time being, all you need to know is that these are separate parts of an EJB implementation, each of which provide support for different EJB component types.

Let's start digging a little deeper into the various EJB component types, starting with session beans.

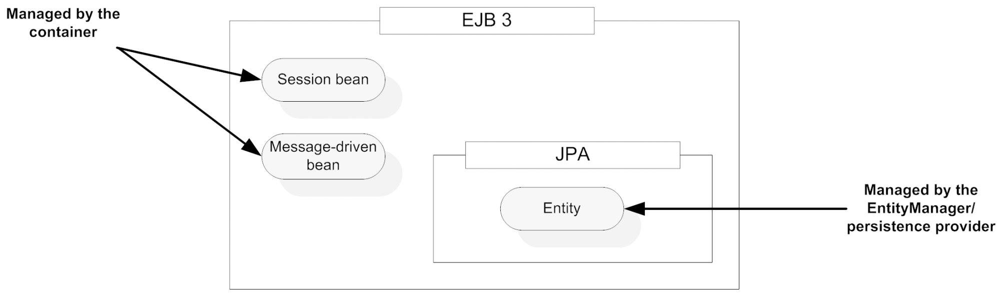

Figure 1.6 Overall organization of the EJB 3 API. The Java persistence API is completely separable from the EJB 3 container. The business logic processing is carried out by through two component types: session beans and message-driven beans. Both components are managed by the container. Persistence objects are called entities, which are managed by the persistent provider through the EntityManager interface.

1.2.1 Session beans

A session bean is invoked by a client for the purpose of performing a specific business operation, such as checking the credit history for a customer. The name *session* implies that a bean instance is available for the duration of a "unit of work" and does not survive a server crash or shutdown. A session bean can model any application logic functionality. There are two types of session beans: *stateful* and *stateless*.

A stateful session bean automatically saves bean state between client invocations without your having to write any additional code. A typical example of a state-aware process is the shopping cart for a web merchant like Amazon. In contrast, stateless session beans do not 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 customer credit history.

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

1.2.2 Message-driven beans

Like session beans, MDBs process business logic. However, 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 IBM WebSphere MQ, SonicMQ, Oracle Advanced Queueing, and TIBCO. MDBs are typically used for robust system integration or 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.

Next we'll explain the concept of persistence and describe how object-relational frameworks help enable automated persistence.

1.2.3 Entities and the Java Persistence API

One of the exciting new features of EJB 3 is the way it handles persistence. We briefly mentioned persistence providers and the JPA earlier, but now let's delve into the details.

Persistence is the ability to have data contained in Java objects automatically stored into a relational database like Oracle, SQL Server, and DB2. Persistence in EJB 3 is managed by the JPA. It automatically persists the Java objects using a technique called object-relational mapping (ORM). ORM is essentially the process of mapping data held in Java objects to database tables using configuration. It relieves you of the task of writing low-level, boring, and complex JDBC code to persist objects into a database.

The frameworks that provide ORM capability to perform automated persistence are known as ORM frameworks. As the name implies, an ORM framework performs transparent persistence by making use of object-relational mapping metadata that defines how objects are mapped to database tables. ORM is not 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 among the mainstream developer community.

In EJB 3 terms, a persistence provider is essentially an ORM framework that supports the EJB 3 Java Persistence API (JPA). The JPA defines a standard for

- The creation of ORM configuration metadata for mapping entities to relational tables
- The EntityManager API—a standard API for performing CRUD (create, read, update, and delete)/persistence operations for entities
- The *Java Persistence Query Language* (JPQL), for searching and retrieving persisted application data

Since JPA standardizes ORM frameworks for the Java platform, you can plug in ORM products like JBoss Hibernate, Oracle TopLink, or BEA Kodo as the underlying JPA "persistence provider" for your application.

It may occur to you that automated persistence is something you'll find useful for all kinds of applications, not just server-side applications such as those built with EJB. After all, JDBC, the grandfather of JPA, is used in everything from large-scale real-time systems to desktop-based hacked-up prototypes. This is exactly why JPA is completely separate from the rest of EJB 3 and usable in plain Java SE environments.

Entities are the session bean and MDB equivalent in the JPA world. Let's take a quick glance at them next, as well as the `EntityManager` API and the Java Persistence Query Language (JPQL).

Entities

If you're using JPA to build persistence logic of your applications, then you have to use entities. Entities are the Java objects that are persisted into the database. Just as session beans model processes, entities model lower-level application concepts that high-level business processes manipulate. While session beans are the "verbs" of a system, entities are the "nouns." Examples include an `Employee` entity, a `User` entity, an `Item` entity, and so on. Here's another perfectly valid (and often simpler-to-understand) way of looking at entities: they are the OO representations of the application data stored in the database. In this sense, entities survive container crashes and shutdown. You must be wondering how the persistence provider knows where the entity will be stored. The real magic lies in the ORM metadata; an entity contains the data that specifies how it 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.

The EntityManager

The JPA `EntityManager` interface manages entities in terms of actually providing persistence services. While entities tell a JPA provider how they map to the database, they do not 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. In addition, the JPA provides the ability to handle lifecycle management, performance tuning, caching, and transaction management.

The Java Persistence Query Language

JPA provides a specialized SQL-like query language called the 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 is worth using.

At this point, you should have a decent high-level view of the various parts of EJB. You also know that you need an EJB container to execute session beans and MDBs as well as a persistence provider to run your entities, so that these components can access the services EJB 3 provides. The container, the persistence provider, and the services are the central concepts in EJB 3, and we'll address them next.

1.3 Getting inside EJB

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 the previous section) to execute session beans and MDBs you need an EJB container, and to run your entities you need a persistence provider. In this section we give you a bird's-eye view of containers and persistence providers and explain how they are related.

In the Java world, containers aren't just 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, or JSF. A *Java EE container* is an application server solution that supports EJB 3, a web container, and other Java EE APIs and services. BEA WebLogic Server, Sun Microsystems's GlassFish, IBM WebSphere, JBoss Application Server, and Oracle Application Server 10*g* are examples of Java EE containers. The relationship between the Java EE container, web container, EJB container, and JPA persistence provider is shown in figure 1.7.

If you install a Java EE–compliant application server such as GlassFish, it will contain a preconfigured web container, EJB container, and a JPA provider. However, some vendors and open source projects may provide only a web container such as Tomcat or an EJB 3–compliant persistence provider such as Hibernate. These containers provide limited functionality compared to what you get with a complete Java EE 5 container.

In this section, we'll focus on how the EJB container and the persistence provider work, and we'll finish with a more complete discussion of EJB services. First, let's tackle the EJB container.

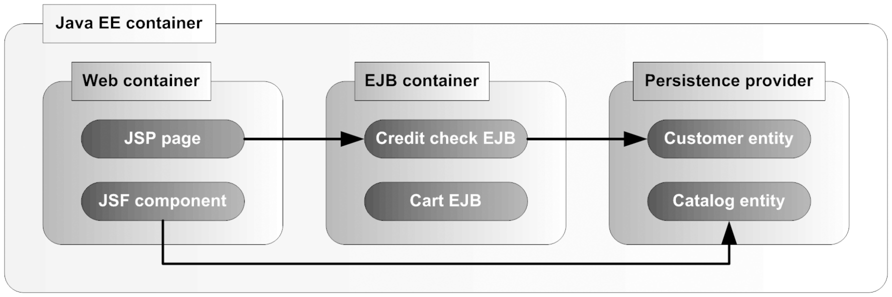

Figure 1.7 Java EE container typically contains web and EJB containers and a persistence provider. The stateless session bean (Credit Check EJB) and stateful session bean (Cart EJB) are deployed and run in the EJB container. Entities (Customer and Catalog) are deployed and run within an EJB persistence provider and can be accessed by either web or EJB container components.

1.3.1 Accessing EJB services: the EJB container

Think of the container as simply 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, remoting, and web services support. As a matter of fact, you might even think of the container as a JVM on steroids, whose purpose is to execute EJBs. In EJB 3, the container provides services applicable to session beans and MDBs only. 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.

The persistence provider is the container counterpart in JPA. We'll briefly talk about it next.

1.3.2 Accessing JPA services: the persistence provider

In section 1.2.3, we mentioned that the persistence provider's job is to provide standardized JPA services. Let's explore how it does that. Instead of following the JVM-like container model, JPA follows a model similar to APIs, like JDBC. JPA provides persistence services such as retrieving, adding, modifying, and deleting JPA entities when you explicitly ask for them by invoking `EntityManager` API methods.

The "provider" terminology comes from APIs such as JDBC and JNDI too. If you've worked with JDBC, you know that a "provider" is essentially the vendor implementation that the JDBC API uses under the covers. Products that provide JPA implementation are *persistence providers* or *persistence engines*. JBoss Hibernate and Oracle TopLink are two popular JPA providers.

Since JPA is completely pluggable and separable, the persistence provider and container in an EJB 3 solution need not come from the same vendor. For example, you could use Hibernate inside a BEA WebLogic container if it suits you better, instead of the Kodo implementation WebLogic ships with.

But without services, what good are containers? In the next section, we explore the services concept critical to EJB.

1.3.3 Gaining functionality with EJB services

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 JSP, JSF, or Struts, couldn't you create your web application using just the Java language and maybe some APIs like JDBC for database access? The plain answer is that you could—if deadlines and cutthroat competition were not realities. Indeed, before anyone dreamed up EJB this is exactly what people did. What the resulting long hours proved is that you tend to spend a lot of time solving very common system-level problems instead of focusing on the real business solution. These bitter experiences emphasized the fact that there are common solutions that can be reused to solve common development problems. This is exactly what EJB brings to the table.

EJB is a collection of "canned" solutions to common server application development problems as well as a roadmap to common server component patterns. These "canned" solutions, or *services*, are provided by either the EJB container or the persistence provider. To access those services, you build the application components and deploy them into the container. Most of this book will be spent explaining how you can exploit EJB services.

In this section, we briefly introduce some of the services EJB offers. Obviously, we can't explain the implementation details of each service in this section. Neither is it necessary to cover every service EJB offers right now. Instead, we briefly list the major EJB 3 services in table 1.1 and explain what they mean to you from a practical perspective. This book shows you how to use each of the services shown in table 1.1 in your application.

Despite its robust features, one of the biggest beefs people had with EJB 2 was that it was too complex. It was clear that EJB 3 had to make development as simple as possible instead of just continuing to add additional features or services. If you have worked with EJB 2 or have simply heard or read that it is complex, you should be curious as to what makes EJB 3 different. Let's take a closer look.

Table 1.1 Major EJB 3 component services and why they are important to you. The persistence services are provided by the JPA provider.

Service	Applies To	What It Means for You
Integration	Session beans and MDBs	Helps glue together components, ideally through simple configuration instead of code. In EJB 3, this is done through dependency injection (DI) as well as lookup.
Pooling	Stateless session beans, MDBs	For each EJB component, the EJB platform creates a pool of component instances that are shared by clients. At any point in time, each pooled instance is only allowed to be used by a single client. As soon as an instance is finished servicing a client, it is returned to the pool for reuse instead of being frivolously discarded for the garbage collector to reclaim.
Thread-safety	Session beans and MDBs	EJB makes all components thread-safe and highly performant in ways that are completely invisible. This means that you can write your server components as if you were developing a single-threaded desktop application. It doesn't matter how complex the component itself is; EJB will make sure it is thread-safe.
State management	Stateful session beans	The EJB container manages state transparently for stateful components instead of having you write verbose and error-prone code for state management. This means that you can maintain state in instance variables as if you were developing a desktop application. EJB takes care of all the details of session maintenance behind the scenes.
Messaging	MDBs	EJB 3 allows you to write messaging-aware components without having to deal with a lot of the mechanical details of the Java Messaging Service (JMS) API.
Transactions	Session beans and MDB	EJB supports declarative transaction management that helps you add transactional behavior to components using simple configuration instead of code. In effect, you can designate any component method to be transactional. If the method completes normally, EJB commits the transaction and makes the data changes made by the method permanent. Otherwise the transaction is rolled back.
Security	Session beans	EJB supports integration with the Java Authentication and Authorization Service (JAAS) API, so it is very easy to completely externalize security and secure an application using simple configuration instead of cluttering up your application with security code.
Interceptors	Session beans and MDBs	EJB 3 introduces AOP in a very lightweight, accessible manner using *interceptors*. This allows you to easily separate out crosscutting concerns such as logging, auditing, and so on in a configurable way.
Remote access	Session beans	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 DI.

continued on next page

Table 1.1 Major EJB 3 component services and why they are important to you. The persistence services are provided by the JPA provider. *(continued)*

Service	Applies To	What It Means for You
Web services	Stateless session beans	EJB 3 can transparently turn business components into robust web services with minimal code change.
Persistence	Entities	Providing standards-based, 100 percent configurable automated persistence as an alternative to verbose and error-prone JDBC/SQL code is a principal goal of the EJB 3 platform.
Caching and performance	Entities	In addition to automating persistence, JPA transparently provides a number of services geared toward data caching, performance optimization, and application tuning. These services are invaluable in supporting medium to large-scale systems.

1.4 Renaissance of EJB

Software is organic. Much like carbon-based life forms, software grows and evolves. Features die. New features are born. Release numbers keep adding up like the rings of a healthy tree. EJB is no exception to the rule of software evolution. In fact, as far as technologies go, the saga of EJB is more about change than it is about stagnation. Only a handful of other technologies can boast the robust metamorphosis and continuous improvements EJB has pulled off.

It's time to catch a glimpse of the new incarnation of EJB, starting with an example of a simple stateless session bean and then revealing the features changes that make EJB an easy-to-use development tool.

To explore the new features of EJB 3, we'll be pointing out some of the problems associated with EJB 2. If you are not familiar with EJB 2, don't worry—the important thing to remember is how the problems have been resolved in EJB 3.

The problems associated with EJB 2 have been widely discussed. In fact, there have been entire books, such as *Bitter EJB* (Manning Publications, 2003) written about this topic. Chris Richardson in *POJOs in Action* rightfully identified the amount of sheer code you had to write to build an EJB:

> You must write a lot of code to implement an EJB—You must write a home interface, a component interface, the bean class, and a deployment descriptor, which for an entity bean can be quite complex. In addition, you must write a number of boilerplate bean class methods that are never actually called but that are required by the interface the bean class implements. This code isn't conceptually difficult, but it is busywork that you must endure.

In this section, we'd like to walk through some of those points and show you how they have been resolved in EJB 3. As you will see, EJB 3 specifically targets the thorniest issues in EJB 2 and solves them primarily through bold adoption and clever adaptation of the techniques widely available in popular open source solutions such as Hibernate and Spring. Both of which have passed the "market incubation test" without getting too battered. In many ways, this release primes EJB for even further innovations by solving the most immediate problems and creating a buffer zone for the next metamorphosis.

But first, let's look at a bit of code. You will probably never use EJB 2 for building simple applications such as Hello World. However, we want to show you a simple EJB implementation of the ubiquitous Hello World developed using EJB 3. We want you to see this code for a couple reasons: first, to demonstrate how simple developing with EJB 3 really is, and second, because this will provide context for the discussions in the following sections and make them more concrete.

1.4.1 HelloUser 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 is very well suited to introducing a technology as simply and plainly as possible. While almost every technology book starts with a Hello World example, to keep things lively and relevant we plan to deviate from that rule and provide a slightly different example.

In 2004, one of the authors, Debu, 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 accomplished easily in EJB 2. You had to write a home interface, a component interface, a bean class, and a deployment descriptor. Well, now that EJB 3 has been finalized, let's see if Debu was right in his prediction (listing 1.1).

Listing 1.1 HelloUser Session bean

```
 package ejb3inaction.example;
public interface HelloUser {    <-❶ HelloUser POJI
    public void sayHello(String name);
}

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

```
@Stateless     ◁─❷ Stateless annotation
public class HelloUserBean implements HelloUser { ◁─❸ HelloUserBean POJO
    public void sayHello(String name) {
        System.out.println("Hello " + name + " welcome to EJB 3!");
    }
}
```

Listing 1.1 is indeed a complete and self-contained example of a working EJB! Note that for simplicity we have kept both the interface and class as part of the same listing. As you can see, the EJB does not look much more complex than your first Java program. The interface is a plain old Java interface (POJI) ❶ and the bean class is a plain old Java object (POJO) ❸. The funny `@Stateless` symbol in listing 1.1 is a metadata annotation ❷ that converts the POJO to a full-powered stateless EJB. If you are not familiar with metadata annotations, we explore them in chapter 2. In effect, they are "comment-like" configuration information that can be added to Java code.

To execute this EJB, you have to deploy it to the EJB container. If you want to execute this sample, download the zip containing code examples from www.manning.com/panda and follow the online instructions to deploy and run it in your favorite EJB container.

However, don't worry too much about the details of this code right now; it's just a simple illustration. We'll dive into coding details in the next chapter. Our intent for the Hello World example is to use it as a basis for discussing how EJB 3 addresses the thorniest issues that branded EJB 2 as ponderous.

Let's move on now and take a look at what has transformed the EJB elephant into the EJB cow.

1.4.2 Simplified programming model

We heartily agree with Chris Richardson's quote: one of the biggest problems with EJB 2 was the sheer amount of code you needed to write in order to implement an EJB.

If we had attempted to produce listing 1.1 as an EJB 2 example, we would have had to work with several classes and interfaces just to produce the simple one-line output. All of these classes and interfaces had to either implement or extend EJB API interfaces with rigid and unintuitive constraints such as throwing `java.rmi.RemoteException` for all methods. Implementing interfaces like `javax.ejb.SessionBean` for the bean implementation class was particularly time consuming since you had to provide an implementation for lifecycle callback methods like `ejbCreate`, `ejbRemove`, `ejbActivate`, `ejbPassivate`, and `setSessionContext`, whether or not you actually used them. In effect, you were forced to deal with several mechanical steps to accomplish very little. IDE tools like JBuilder, JDeveloper, and WebSphere Studio helped matters a bit by automating some of these steps. However, in general, decent tools with robust support were extremely expensive and clunky.

As you saw in listing 1.1, EJB 3 enables you to develop an EJB component using POJOs and POJIs that know nothing about platform services. You can then apply configuration metadata, using annotations, to these POJOs and POJIs to add platform services such as remoteability, web services support, and lifecycle callbacks only as needed.

The largely redundant step of creating home interfaces has been done away with altogether. In short, EJB service definitions have been moved out of the type-safe world of interfaces into deploy and runtime configurations where they are suited best. A number of mechanical steps that were hardly ever used have now been automated by the platform itself. In other words, *you do not* have to write a lot of code to implement an EJB!

1.4.3 Annotations instead of deployment descriptors

In addition to having to write a lot of boilerplate code, a significant hurdle in managing EJB 2 was the fact that you still had to do a lot of XML configuration for each component. Although XML is a great mechanism, the truth is that not everyone is a big fan of its verbosity, poor readability, and fragility.

Before the arrival of Java 5 metadata annotations, we had no choice but to use XML for configuration. EJB 3 allows us to use metadata annotations to configure a component instead of using XML deployment descriptors. As you might be able to guess from listing 1.1, besides eliminating verbosity, annotations help avoid the monolithic nature of XML configuration files and localize configuration to the code that is being affected by it. Note, though, you can still use XML deployment descriptors if they suit you better or simply to supplement annotations. We'll talk more about this in chapter 2.

In addition to making the task of configuration easier, EJB 3 reduces the total amount of configuration altogether by using sensible defaults wherever possible. This is especially important when you're dealing with automated persistence using ORM, as you'll see in chapters 7, 8, 9, and 10.

1.4.4 Dependency injection vs. JNDI lookup

One of the most tedious parts of EJB 2 development was writing the same few lines of boilerplate code many times to do a JNDI lookup whenever you needed to access an EJB or a container-managed resource, such as a pooled database connection handle. In *POJOs in Action*, Chris Richardson sums it up well:

> A traditional J2EE application uses JNDI as the mechanism that one component uses to access another. For example, the presentation tier uses a JNDI lookup to obtain a reference to a session bean home interface. Similarly, an EJB uses JNDI to access the resources that it needs, such as a JDBC `DataSource`. The trouble with JNDI is that it couples application code to the application server, which makes development and testing more difficult.

In EJB 3, JNDI lookups have been turned into simple configuration using metadata-based dependency injection (DI). For example, if you want to access the `HelloUser` EJB that we saw in listing 1.1 from another EJB or servlet, you could use code like this:

```
...
@EJB
private HelloUser helloUser;

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

Isn't that great? The `@EJB` annotation transparently "injects" the `HelloUser` EJB into the annotated variable. EJB 3 dependency injection essentially gives you a simple abstraction over a full-scale enterprise JNDI tree. Note you can still use JNDI lookups where they are unavoidable.

1.4.5 Simplified persistence API

A lot of the problems with the EJB 2 persistence model were due to the fact that it was applying the container paradigm to a problem for which it was ill suited. This made the EJB 2 entity bean programming model extremely complex and unintuitive. Enabling remote access was one of the prime motivators behind making entity beans container-managed. In reality, very few clients made use of this feature because of performance issues, opting to use session beans as the remote access point.

Undoubtedly entity beans were easily the worst part of EJB 2. EJB 3 solves the problem by using a more natural API paradigm centered on manipulating metadata-based POJOs through the `EntityManager` interface. Moreover, EJB 3 entities do not carry the unnecessary burden of remote access.

Another limitation with EJB 2 was that you couldn't send an EJB 2 entity bean across the wire in different tiers. EJB developers discovered an anti-pattern for this problem: adding another layer of objects—the data transfer objects (DTOs). Chris sums it up nicely:

> *You have to write data transfer objects*—A data transfer object (DTO) is a dumb data object that is returned by the EJB to its caller and contains the data the presentation tier will display to the user. It is often just a copy of the data from one or more entity beans, which cannot be passed to the presentation tier because they are permanently attached to the database. Implementing the DTOs and the code that creates them is one of the most tedious aspects of implementing an EJB.

Because they are POJOs, entities can be transferred between different tiers without having to resort to anti-patterns such as data transfer objects.

The simplification of the persistence API leads to several other benefits, such as standardization of persistence frameworks, a separable persistence API that can be used outside EJB container, and better support of object-oriented features such as inheritance and polymorphism. We'll see EJB 3 persistence in action in chapter 2, but now let's take a close look at some of the main features of the persistence API.

Standardized persistence

One of the major problems with EJB 2 entity beans was that ORM was never standardized. EJB 2 entity beans left the details of database mapping configuration to the provider. This resulted in entity beans that were not portable across container implementations. The EJB 2 query mechanism, EJB-QL, had a similar unfinished feel to it. These standardization gaps have in effect given rise to highly divergent alternative ORM paradigms like Hibernate, Oracle TopLink, and JDO.

A major goal of JPA is to close the standardization gaps left by EJB 2. EJB 3 solidifies automated persistence with JPA in three distinct ways. First, it provides a robust ORM configuration set capable of handling most automated persistence complexities. Second, the Java Persistence Query Language (JPQL) significantly improves upon EJB-QL, standardizing divergent OR query technologies. Third, the `EntityManager` API standardizes ORM CRUD operations. But standardization isn't the only benefit of the simplified API: another great feature is that it can run outside the container.

The cleanly separated Java Persistence API

As we touched on in section 1.2.3, API persistence 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 is the realization that drove the decision to make JPA 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. This is a remarkably positive step in expanding the scope of EJB 3 outside the traditional realm of server applications.

Better persistence-tier OO support

Because EJB 2 entity beans were record oriented, they didn't support rich OO features like inheritance and polymorphism, and they didn't permit the mixing of persistent state and domain logic. As you saw in section 1.1.3, this made it impossible to model the domain layer in DDD architecture.

EJB 3 entities have robust OO support, not just because they are POJOs but also because the JPA ORM mapping scheme is designed with OO in mind. JPQL has robust support for OO as well. Getting impatient to learn more about JPA? Stick with us and we'll have many discussions on JPA throughout the book; part 3 is devoted to discussions on JPA.

Test-driven development has become quite popular because it can dramatically improve performance of software applications. Let's see how EJB 3 improves the testability of applications.

1.4.6 Unit-testable POJO components

Being able to unit-test component state or logic in response to simulated input is a critical technique in increasing code quality. In EJB 2, only functional testing of components was possible since components had to be deployed to the container to be executed. While functional testing simulating user interactions with the system is invaluable, it is not a good substitute for lower-level unit testing.

Because all EJB 3 components are POJOs, they can easily be executed outside the container. This means that it is possible to unit-test all component business logic using testing frameworks such as JUnit or TestNG.

These are just the primary changes to EJB 3; there are many more that we'll cover throughout the book.

Just in case you thought you had to choose between Spring and EJB 3, we thought we'd mention why they don't necessarily need to be regarded as competing technologies.

1.4.7 EJB 3 and Spring

As we mentioned earlier, EJB 3 and Spring are often seen as competitors; however, if you look more closely, you can see that they can also be complementary. Spring has some particularly strong points: support for inversion of control (IoC) for components with simple lifecycles such as singletons; feature-heavy (but slightly more complex) aspect-oriented programming (AOP) support; a number of simple interfaces such as `JDBCTemplate` and `JMSTemplate` utilizing common usage patterns of low-level Java EE APIs; and so on.

EJB 3, on the other hand, provides better support for transparent state management with stateful session beans, pooling, thread-safety, robust messaging support with MDBs, integrated support for distributed transaction management, standardized automated persistence through JPA, and so on.

From a levelheaded, neutral point of view, EJB 3 and Spring can be complementary technologies. The good news is that parts of both the Spring and Java EE communities are working diligently to make Spring/EJB 3 integration a reality. This is particularly good news if you have a significant investment in Spring but want to utilize the benefits of EJB 3. We'll talk about Spring/EJB 3 integration in more detail in chapter 16. However, we'd like to list the possibilities now.

Treating EJB 3 business-tier components as Spring beans

It is possible to treat EJB 3 business-tier components as Spring beans. This translates into an architecture shown in figure 1.8. In this architecture, Spring is used for gluing together the application that contains EJB 3 business-tier components.

The Spring Pitchfork project, part of Spring 2, is meant to make such an integration scenario completely transparent. The Spring framework plans to

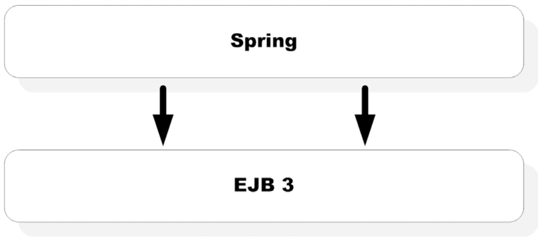

Figure 1.8
Spring/EJB 3 integration strategy. It is possible to use EJB 3 business-tier components as if they were Spring beans. This allows you to use the complementary strengths of both technologies in a "hybrid" fashion.

support EJB 3 annotation metadata specifying stateless session beans, interceptors, resource injection, and so on.

Integrating the JPA into Spring

Suppose that you find Spring is a good fit for your business-tier needs and you simply want to standardize your persistence layer. In this case, it is easy to integrate JPA directly into Spring, much like Spring/Hibernate or Spring/JDO integration. This scheme is shown in figure 1.9.

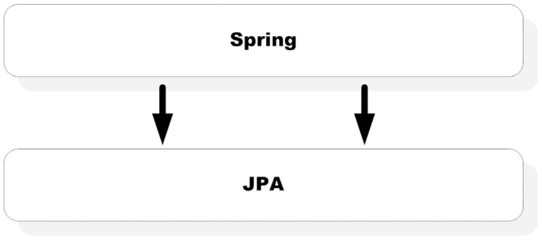

Figure 1.9
Spring/JPA integration. Because JPA is a cleanly separable API, you can integrate Spring with JPA just as you would integrate Hibernate.

In addition to using Spring with JPA, you may find yourself in a situation where you would like to use both Spring and EJB 3 session beans together. Let's examine the possibilities of such integration.

Using Spring interfaces inside EJB 3 components

Yet another interesting idea is to use some of the Spring interfaces like `JDBCTemplate` and `JMSTemplate` or even Spring beans inside EJB 3 components. You can do this either through direct instantiation or access through the Spring application context. Container vendors like JBoss, Oracle, and BEA are working to provide seamless support for integrating Spring beans into session beans and MDBs. This kind of integration is visualized in figure 1.10. We'll discuss combining the power of EJB 3 and Spring in chapter 16.

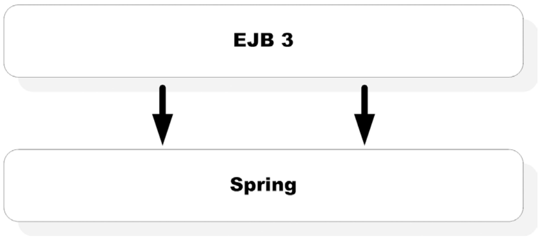

Figure 1.10
In certain cases, it might be a good idea to use Spring from EJB 3. Although it is possible to do so today, such support is likely to be much better in the future.

1.5 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 configurable through simplified metadata annotations.
- Accessing EJBs from client applications has become very simple using dependency injection.
- EJB 3 standardizes the persistence with the Java Persistence API, which defines POJO entities that can be used both inside and outside the container.

We also provided a taste of code to show how EJB 3 addresses development pain points that were inherent with EJB 2, and we took a brief look at how EJB 3 can be used with Spring.

Armed with this essential background, you are 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 entire EJB 3 API that shows just how easy the code really is.

JAVA

EJB 3 IN ACTION

Debu Panda, Reza Rahman, Derek Lane

EJB 2 is widely used but it comes at a cost—procedural, redundant code. EJB 3 is a different animal. By adopting a POJO programming model and Java 5 annotations, it dramatically simplifies enterprise development. A cool new feature, its Java Persistence API, creates a standard for object-relational mapping. You can use it for any Java application, whether inside or outside the EJB container. With EJB 3 you will create true object-oriented applications that are easy to write, maintain and extend.

EJB 3 in Action is a fast-paced tutorial for both novice and experienced Java developers. It will help you learn EJB 3 and the JPA quickly and easily. This comprehensive, entirely new EJB 3 book starts with a tour of the EJB 3 landscape. It then moves quickly into core topics like building business logic with session and message-driven beans. You'll find four full chapters on the JPA along with practical code samples, design patterns, performance tuning tips, and best practices for building and deploying scalable applications.

What's Inside

- Dependency Injection and Interceptors
- Domain modeling and persisting entities with the JPA and its query language
- Using EJB 3 across application tiers and web services
- Integrating with Spring
- Migrating from EJB 2, JDBC, and other O-R frameworks

Debu Panda, a member of the EJB 3.0 Expert Group, is a lead product manager of the Oracle Application Server development team focused on EJB. **Reza Rahman** is a JavaEE5 architect who works with EJB, Spring and Hibernate. **Derek Lane** is the CTO of a technology company and the founder of the Dallas and Oklahoma City Java User Groups.

"... this technical book is surprisingly entertaining."
—King Y. Wang, Oracle Canada

"... well written, easy, and fun."
—Patrick Dennis
Management Dynamics Inc.

"... this is *the* [EJB] book to read. Don't miss its practical advice!"
—Jeanne Boyarsky
JavaRanch.com

"Great book—covers everything relating to EJB 3.0."
—Awais Bajwa
Expert Group Member
JSR 243 Java Data Objects

For more information, code samples, and ebook visit
www.manning.com/EJB3inAction

ISBN-10: 1-933988-34-7
ISBN-13: 978-1-933988-34-4
54499
9 781933 988344

MANNING $44.99 US/$58.99 Canada