EJB 3 IN ACTION

SECOND EDITION

Debu Panda
Reza Rahman
Ryan Cuprak

MEAP

MEAP Edition
Manning Early Access Program
EJB 3 in Action, Second Edition, version 3

Copyright 2012 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

Table of Contents

Part 1 Overview of the EJB Landscape

1 What's what in EJB 3

2 A first taste of EJB 3

Part 2 Working with EJB 3 Components

3 Building business logic with session beans

4 Messaging and developing MDBs

5 EJB runtime context, dependency injection, and aspect oriented programming

6 Transactions and security

7 Scheduling and timers

8 Exposing EJBs as SOAP and REST web services

Part 3 Using EJB 3 with JPA and CDI

9 JPA entities

10 Managing entities

11 Using CDI with EJB 3

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

Part 4 Putting EJB 3 into action

12 Packaging EJB 3 applications

13 EJB 3 testing

14 Designing EJB-based systems

15 EJB performance and scalability

16 EJB 3, Seam, and Spring

17 The future of EJB 3

Appendixes

Appendix A RMI Primer

Appendix B Migrating from EJB 2.1 to EJB 3

Appendix C Annotations reference

Appendix D Deployment descriptors reference

Appendix E Installing and configuring the Java EE 6 SDK

Appendix F EJB 3 developer certification exam

Appendix G EJB 3 tools support

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

Part 1

Overview of the EJB landscape

This book is about Enterprise Java Beans (EJB) 3.1, the newest member of the Enterprise Java Beans family. The goals of 3.1 are to continue to simplify the API - making EJB as easy to use as possible - and to implement much of the functionality requested by the user community after the release of 3.0. This resulted in a nice array of new toys, and we're going to share them with you.

Part 1 will present EJB 3 as a powerful, highly usable platform worthy of its place as the business component development standard for mission-critical enterprise development. We'll introduce the Java Persistence API (JPA 2), a Java EE technology that aims to standardize Java ORM and works hand-in-hand with EJB 3. We'll also take a quick look at Contexts and Dependency Injection for Java (CDI), the next-generation generic type-safe dependency injection technology for Java EE.

Chapter 1 is an introduction to the pieces that make up EJB 3, touching on the unique strengths EJB has as a development platform and the new features that promote productivity and ease of use. We'll even throw in a "Hello World" example.

In chapter 2 we'll provide more realistic code samples and introduce the Action Bazaar application, an imaginary enterprise system developed throughout the book. We'll try to give you a feel for how EJB 3 looks as quickly and easily as possible. Be ready for a lot of code!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

1
What's what in 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 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 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.

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. The following chapters will 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

In the Java world EJB is an important and uniquely influential technology radically transformed in version 3. We will 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. If you're curious about it, 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 the next. 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 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? You could—if deadlines and limited resources were not realities. Before anyone dreamed up EJB this is exactly what people did. The resulting long hours proved that you would 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 refers to server-side components that you can use to build the business component layer of your application. Some of us tend to 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 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.

In order 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 sub-divided into stateless session beans, stateful session beans and singletons. Each component type has a specialized purpose, scope, state, life-cycle and usage pattern in the business tier. We'll discuss these component types throughout the rest of the book, particularly in Part 2. Although they are not EJBs, entities are another type of component you will need to know about. Entities reside in the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

persistence tier and are part of JPA 2. We'll discuss Entities 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. CDI allows you to use dependency injection with all managed beans, including EJBs. We'll explore CDI and managed beans more 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 recognized components because they simply make a lot of sense for business tier components. These services include dependency injection, transactions, thread-safety and pooling. In order to use most services, you must declare that 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, particularly Part 2. 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.

Service	What It Means for You
Registry, dependency injection and look-up	Helps locate and glue together components, ideally through simple configuration. Let's you change component wiring for testing.
Life-cycle management	Let's you take appropriate actions when the life-cycle of a component changes such as when it is created and when it is destroyed.
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 itself is; EJB will make sure it is thread-safe.
Transactions	EJB automatically makes all of your components transactional that means you don't have to write any transaction code while using databases or messaging servers via JDBC, JPA or JMS.
Pooling	EJB 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 done servicing a client, it is 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

	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 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/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 *passivation*. When memory becomes available again and a passivated component is needed, EJB puts the component back into memory. This is called *activation*.
Messaging	EJB 3 allows you to write message processing components without having to deal with a lot of the mechanical details of the Java Messaging Service (JMS) API.
Security	EJB allows you to easily secure your components through simple configuration.
Scheduling	Let's 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 in a very lightweight, intuitive way using *interceptors*. This allows you to easily separate out crosscutting concerns such as logging, auditing, and do so on in a configurable way.
Web services	EJB 3 can transparently turn business components into SOAP or 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 DI.
Testing	You can easily unit and integration test any EJB component using embedded containers with frameworks like JUnit.

Table 1.1: EJB Services

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

1.1.3 Layered architectures and EJB

Enterprise applications are designed to solve a unique type of problem and so 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 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, 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 is not 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 today: traditional four-tier architectures and domain-driven design (DDD). Let's take a brief 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 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 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

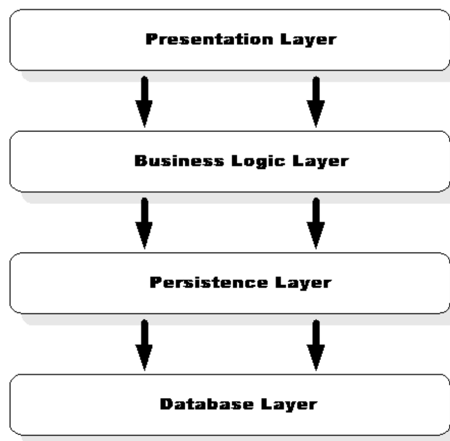

Figure 1.1 Most traditional enterprise applications have at least four layers. 1) The presentation layer is the actual user interface and can either be a browser or a desktop application. 2) The business logic layer defines the business rules. 3) The persistence layer deals with interactions with the database. 4) The database layer consists of a relational database such as Oracle that stores the persistent objects.

EJB is not 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.

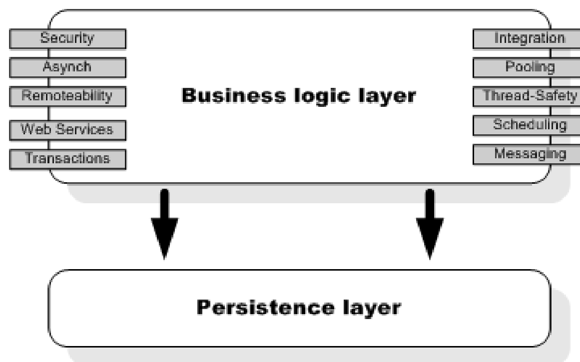

Figure 1.2 The component services offered by EJB 3 at supported application layer. Note that each service is independent of each other, so you are for the most part free to pick the features important for your application.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

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 rather 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 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 are not in stock. Being POJOs, JPA entities support OO features, such as inheritance or polymorphism. It's easy to implement a persistence object model with the JPA and easy 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 [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 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 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 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 avoidance of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

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), 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 and JavaServer Faces (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 are to adopt it. EJB 3 has an open, public API specification and compatibility test kit, which 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 and that 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 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

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.

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 up-front 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 remoting. 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 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 two types based on what they are 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

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 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 three types of session beans: *stateful, stateless* and *singleton*.

A stateful session bean automatically saves 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 life-cycle when the client requests it. 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. Singleton session beans maintain state, are shared by all clients and live for the duration of the application. You will use singleton beans to store shared state such as a discount processing component. Note 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, 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 HornetQ, ActiveMQ, 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.

1.3 Related Specifications

EJB has two very closely related specifications that we will cover in this book. The first is JPA which is the persistence standard for Java EE and CDI which 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. However, JPA has some specific runtime integration points with EJB since the specifications are so closely related. We'll say just a few things about JPA here since we've got chapters dedicated to it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

Persistence is the ability to have data contained in Java objects automatically stored into a relational database like Oracle, SQL Server, and DB2. 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. It relieves you of the task of writing low-level, boring, and complex JDBC code to persist objects into database.

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. Since JPA standardizes ORM frameworks for the Java platform, you can plug in ORM products 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 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.

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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

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 is that this is very limiting. You were not able to inject EJB into Struts or JUnit, and you couldn't inject non-EJB DAOs 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 will cover CDI in detail in a later chapter.

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 the previous section) 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 exist inside.

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 EJB. 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.

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. Oracle WebLogic Server, GlassFish, IBM WebSphere, JBoss Application Server, and Caucho Resin are examples of Java EE containers.

1.4.1 The Application Server

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. Other than supporting Java EE related technologies, some application servers can also function as production-quality HTTP servers. Yet others support modularity via technologies like OSGi.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

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 towards web applications. Web Profile APIs include JSF 2, CDI, EJB 3.1 Lite (discussed below), JPA 2, JTA and bean validation. At the time of writing, GlassFish and Resin provided Java EE 6 Web Profile offerings. Note Java EE 6 Web Profile implementations are free to add APIs as they wish. For example, Resin adds JMS as well as close to all of the EJB API including messaging, remoting and scheduling but not EJB 2 backwards compatibility. Figure 1.3 shows how the Web Profile compares with the complete Java EE platform.

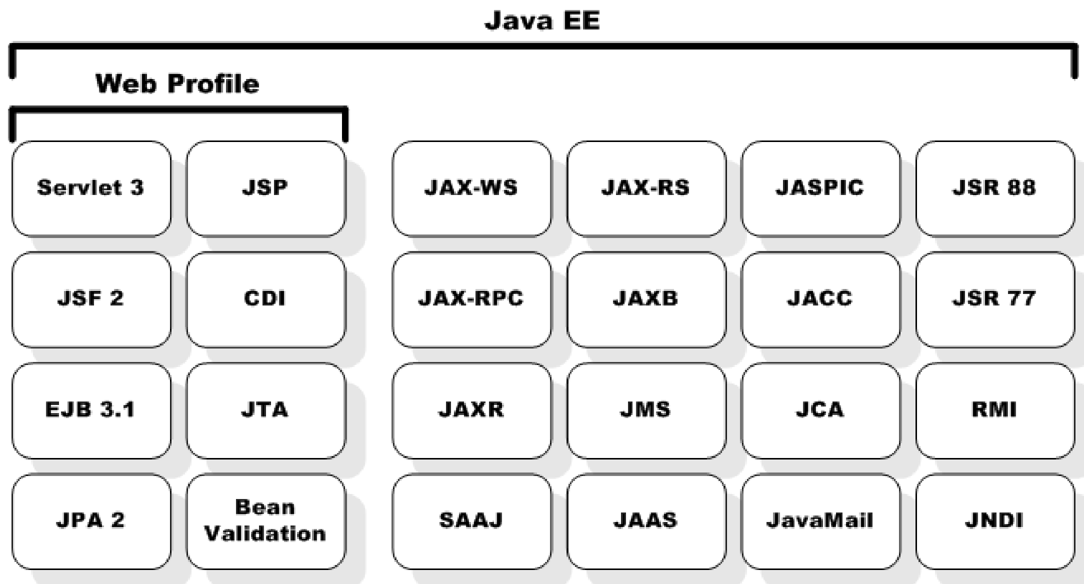

Figure 1.3 Java EE Web Profile vs. Full Java EE Platform

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 6 Web Profile, EJB 3.1 also comes in a scaled down, lighter-weight version called EJB 3.1 Lite. EJB Lite goes hand-in-hand with the Web Profile and is intended for web applications. Just as the Web Profile, any vendor implementing the EJB 3.1 Lite API is free to include EJB features as they wish. From a practical standpoint, the most important thing that EJB 3.1 Lite does is remove support for EJB 2 backwards compatibility. This means that an EJB container can be much more lightweight because it

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

does not have to implement the old APIs in addition to the light-weight EJB 3 model. Because EJB 3.1 Lite also does not include support for MDBs and remoting, it can mean a lighter weight server if you really don't need these features.

For reference, here is a table comparing major EJB and EJB Lite features:

Feature	**EJB Lite**	**EJB**
Stateless beans	✔	✔
Stateful beans	✔	✔
Singleton beans	✔	✔
Message driven beans		✔
No interfaces	✔	✔
Local interfaces	✔	✔
Remote interfaces		✔
Web service interfaces		✔
Asynchronous invocation		✔
Interceptors	✔	✔
Declarative security	✔	✔
Declarative transactions	✔	✔
Programmatic transactions	✔	✔
Timer service		✔
EJB 2.x support		✔
CORBA interoperability		✔

Table 1.2: EJB and EJB Lite Feature Comparison

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 classpath of your application and automatically deploys any EJBs it can find. Figure 1.4 shows the architecture of an embedded EJB 3 container.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

Embeddable containers have been around for a while. OpenEJB, EasyBeans and Embedded JBoss are examples. EJB 3.1 finally makes them required for all vendors that must implement the EJB API. Note 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 the chapter on testing EJB 3.

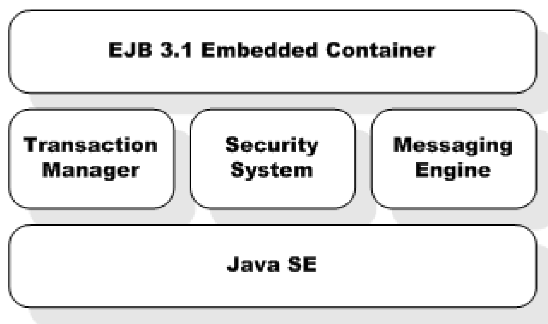

Figure 1.4 EJB 3.1 embedded containers run directly inside Java SE and provide all EJB services such as transactions, security and messaging

1.4.4 Using EJB3 in Tomcat

Apache Tomcat, the light-weight, popular Servlet container, does not support EJB 3. However, you can easily use EJB 3 on Tomcat through embedded containers. In fact, the Apache OpenEJB project has specific support for enabling EJB 3 on Tomcat. As shown in Figure 1.5 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 6 APIs on Tomcat if you wish.

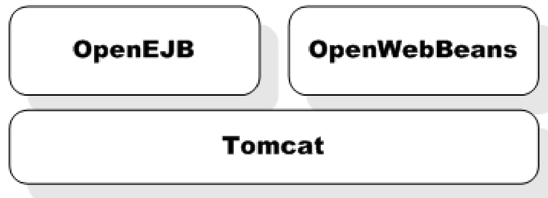

Figure 1.5 You can use OpenEJB and OpenWebBeans to enable both EJB and CDI on Tomcat

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

1.5 The Brave New Innovations

Starting from this point onward, let's start getting a little down and dirty and seeing how the brave new world of EJB 3 actually 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 is very well suited to introducing a technology as simply and plainly as possible.

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 (listing 1.1).

Listing 1.1 HelloUser Session bean

```
package ejb3inaction.example;
import javax.ejb.Stateless;
@Stateless                                                          #A
public class HelloUserBean {                                        #B
    public void sayHello(String name) {
        System.out.println("Hello " + name + " welcome to EJB 3.1!");
    }
}
```

#A HelloUserBean POJO
#B Stateless annotation

Listing 1.1 is a complete and working EJB! The bean class is a plain old Java object (POJO) (#A), 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. Easy to use, but you need to pick your public methods carefully. The exposeAllTheCompanysDirtySecrets() method should probably be private. The funny `@Stateless` symbol in listing 1.1 is a metadata annotation (#B) that converts the POJO to a full-powered stateless EJB. In effect, they are "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 `chapter1.zip` from www.manning.com/panda and follow the online instructions to deploy and run it in your favorite EJB container.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

We're going to be analyzing 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.

1.5.2 Annotations vs. XML

Annotations are a relatively new development that was introduced in the Java language in Java SE 5. Prior to that point, XML was the only logical choice for application configuration simply because there were no other viable options around except maybe 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 it separates the information about the configuration from the Java code that uses it, making maintenance more difficult. Collectively, these problems are named "XML Hell" and annotations are specifically designed to be the cure.

EJB 3.0 was the first mainstream Java technology to pave the way for annotations 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 our code example, 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. As Figure 1.6 depicts, an annotation basically transforms a simple POJO into an EJB, just as the `@Stateless` annotation in our example does.

Figure 1.6 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 cannot change. Configuration that has little to do with Java code is also not very well 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. In fact, you can use XML to override or augment annotation-based configuration.

Unless you have a very strong preference for XML, it is generally advisable to start with annotations and use XML overrides where they are really needed.

1.5.3 Intelligent Defaults vs. Explicit Configuration

EJB takes a different approach to default behavior than most frameworks such as Spring. With Spring for example, generally, if you don't ask you don't get. Whatever behavior you want to have in your Spring components you have to ask for. 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, our hello world component is automatically thread-safe, pooled and transactional without you having to do anything at all. Similarly, if you wanted scheduling, asynchronous processing, remoting or web services, all you need to do is add a few annotations to the component and that is it. There simply is no service that you will 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 vs. JNDI lookup

EJB 3 was re-engineered 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 while using CDI with EJB 3. For example, if you want to access the `HelloUser` EJB that we saw in listing 1.1 from another EJB, Servlet or JSF backing bean you could use code like this:

```
@EJB                                                          #A
private HelloUserBean helloUser;

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

#A EJB Injection

Isn't that great? The `@EJB` annotation (#A) 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 you can still use JNDI lookups where they are unavoidable. For example, to dynamically look-up our bean, we could use code like this:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

```
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 a later chapter.

1.5.5 CDI vs. 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 such as the EJB context. CDI is far more type-safe than EJB. Generally speaking, CDI is a super-set of EJB injection. For example, you can use CDI to inject our EJB as follows:

```
@Inject                                                        #A
private HelloUserBean helloUser;

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

#A EJB Injection via CDI

It might seem that CDI should be used for all Java EE injection, but it currently has a limitation. While 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.

Generally speaking, 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. Indeed, projects like Arquillian focus specifically on integrating JUnit with embedded containers. Listing 1.2 shows how Arquillian allows you to inject EJBs into JUnit tests:

Listing 1.2 EJB 3 Unit testing with Arquillian

```
@RunWith(Arquillian.class)                                              #A
public class HelloUserBeanTest {
    ...
    @Inject                                                             #B
    private HelloUserBean helloUser;

     @Test
     public void testSayHello() {
         helloUser.sayHello("Curious George");                          #C
    }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

```
}
```

#A Running JUnit with Arquillian
#B Injecting bean to be tested
#C Using EJB in unit test

We have dedicated an entire chapter of this book, chapter 13, to testing EJB components.

1.6 Changes in EJB 3.1

Although EJB 3.1 is only a point release, it does add a number of much requested features as well as further enhancements to maximize ease of use. In fact, EJB 3.1 probably pushes the boundaries for a major Java EE API point release. In this section we will briefly talk about the changes in EJB 3.1 in particular.

1.6.1 Changes we already discussed

You've probably noticed that we've talked about a few of the EJB 3.1 changes already. For review here are the changes we've already covered so far:

- EJBs have been redefined to be managed beans with additional services.
- EJBs are no longer required to have interfaces, although you should use this feature wisely.
- Singleton beans have been added to model shared data across the system.
- EJB Lite has been defined to be a smaller sub-set of the EJB API that focuses only on web applications. EJB Lite is intended for lightweight Java EE 6 Web Profile application servers.
- Support for embedded containers has been added primary to support integration testing.
- JPA 2 is now a separate specification from EJB 3.1, although there are well-defined integration points.
- EJB is tightly integrated with CDI, which focuses on dependency injection and context management.

1.6.2 Asynchronous Processing

Sometimes people just get tired of waiting. If the customer on your web page has to wait too long for an order to process she's likely to either lose interest, or click again, either of which may have bad results. An Asynchronous process is one that returns control to the calling process immediately (your customer) and kicks off a separate background process to actually complete the request. EJB 3.1 adds support for extremely intuitive asynchronous processing via the `@Asynchronous` annotation. For example, adding the annotation on the EJB method below automatically makes it asynchronous:

```
@Stateless
public class OrderBillingServiceBean {
    ...
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

```
    @Asynchronous                                              #A
    public void billOrder(Order order) {
        ...
        bill(order);
        ...
    }

    ...
}
```

#A EJB asynchronous method

1.6.3 EJB Timer Service Enhancements

EJB has had a very simple timer service for a while, but it was limited to a simple programmatic API for interval/delay based timers only. What many developers have requested over the years in a cron-style declarative timer for EJB. EJB 3.1 adds such support via the `@Schedule` annotation. The `@Schedule` annotation is simple, intuitive and meets a vast majority of enterprise scheduling needs. For example, you can schedule an EJB method to generate a newsletter like this:

```
@Stateless
public class NewsLetterGeneratorBean {

    @Schedule(dayOfMonth="1")                                  #A
    public void generateMonthlyNewsLetter() {
        ... Code to generate the monthly news letter goes here...
    }
}
```

#A EJB cron schedule

The `@Schedule` (#A) annotation states that the EJB method to generate the monthly newsletters should be automatically invoked on the first day of every month.

1.6.4 Stripped Down EJB Packaging

Standard EJB packaging is modularized, separating the presentation layer and the business logic/persistence layer into two separate archives. The pieces of the presentation layer go into a Web Archive file (war); the EJB and JPA go into a jar file, and then into your ear (not your ear - the Enterprise Archive file). Larger projects with lots of components should probably stick to the traditional packaging. This is shown in figure 1.7

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

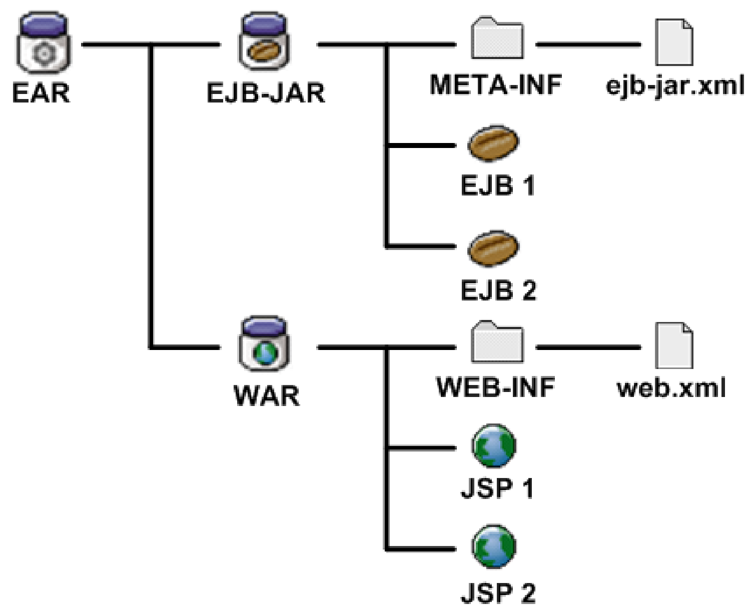

Figure 1.7 Standard EJB Packaging

For smaller projects 3.1 introduced a simpler model, with a war file containing CDI, EJB, JSF, JPA and whatever other resources you need. This simplification goes a surprisingly long way in making EJB easier to use for developers new to Java EE. This packaging is shown in figure 1.8.

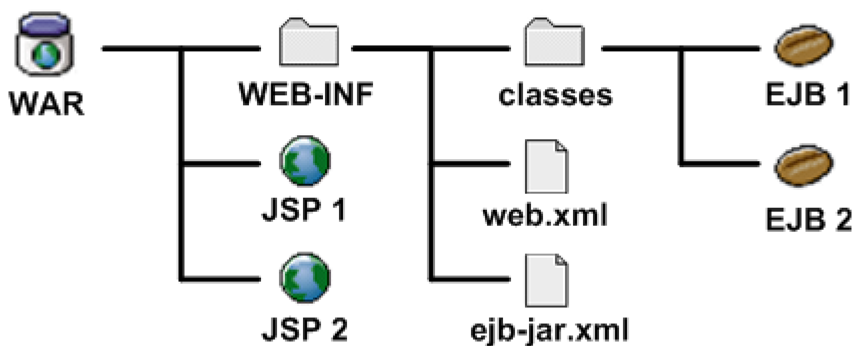

Figure 1.8 Simplified Packaging

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

1.6.5 Standardized JNDI Naming

Each EJB is given a name in the central JNDI registry of an application server. For example, let's take a bean packaged in an EJB-JAR named `creditprocess.jar` inside an EAR file named `actionbazaar.ear`:

```
@Stateless

public class CreditCheckBean {
    ...
}
```

Before standardization, you wouldn't know what exactly what the bean is named in JNDI until you read the vendor's documentation. Every vendor assigned beans names differently. For example one vendor may name the bean "`action-bazaar/CreditCheckBean/local`" while another may simply name it "`CreditCheckBean`". This inconsistency was a portability challenge and was quite confusing.

In EJB 3.1 each EJB is assigned a well-defined name in the global, app and module scopes. For example, the EJB in the example will have a global name of "`java:global/actionbazaar/creditprocess/CreditCheckBean`". We'll discuss JNDI naming in detail in a later chapter but let's next take a look at Seam and Spring and how they relate to Enterprise Java Beans.

1.7 EJB 3, Seam and Spring

EJB 3 is not a middleware technology on an island on its own. In addition to EJB 3, Spring and Seam are two non-standard middleware technologies available to developers. It is important to understand how Spring and Seam can be used with EJB 3. In fact, we have dedicated an entire chapter, chapter 16, of this book to this topic. In this section, we will briefly discuss the topic.

1.7.1 EJB 3 and Seam

Seam and EJB 3 have always been very closely related technologies. In fact, Seam creator Gavin King was instrumental in the development of EJB 3 and led the CDI specification. Seam was created to add the functionality missing in EJB 3.0 such as dependency injection, more powerful interceptors, conversations as well as better JSF integration. Seam also integrated a number of non-standard technologies with EJB 3.

CDI now contains a superset of the core Seam 2 dependency injection and context management features. Indeed, the CDI reference implementation, Weld is a JBoss project split from the Seam 2 codebase. As a result, Seam 3 now does not contain any dependency injection or context management features at all. Rather, Seam 3 is simply a collection of discrete pluggable modules for integrating third-party tools to into any CDI container (and not just Weld). At the time of writing, Seam 3 has modules for Drools, JSF enhancements, internationalization/localization, security, XML bean configuration, Seam 2 backwards compatibility, Spring integration, jBPM, document management, JMS, JBoss ESB, GWT and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

Quartz. Because CDI transparently integrates with EJB 3, this means that EJB 3 can take advantage of all these Seam 3 modules. Figure 1.9 shows the relationship between EJB 3, CDI and Seam 3.

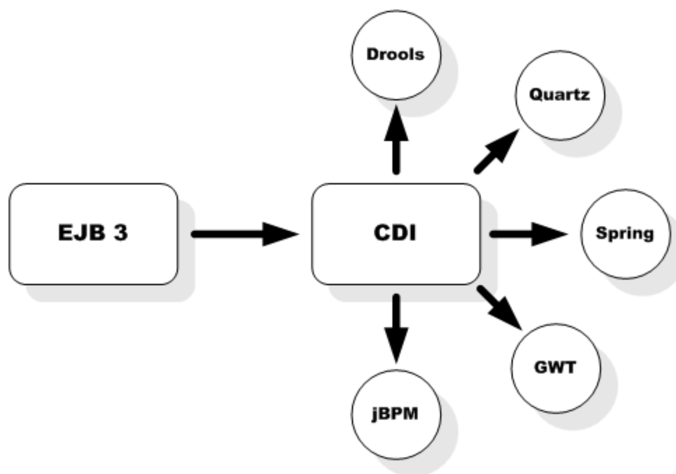

Figure 1.9 Seam provides portable extensions on top of CDI that can be accessed from the EJB container. CDI and EJB have a two-way relationship which each leveraging the other.

1.7.2 EJB 3 and Spring

Spring and EJB 3 have been historically seen as bitterly competing technologies. In reality however, there are a lot of sound technical reasons why these technologies can be thought of as highly complementary. As we noted, EJB 3 is an excellent choice for the business logic tier of your application. It is easy to use-and-configure, scalabale, provides a powerful set of container services and seamless integration with standard APIs such as JPA and JSF. Not counting CDI and Seam 3, Spring by comparison has much more powerful support for generic dependency injection, feature-rich (but more complex) aspect-oriented programming (AOP) support; a number of simple abstractions such as `JDBCTemplate` and `JMSTemplate` for utilizing common usage patterns of low-level Java EE APIs as well as a rich set of third-party integration for popular tools such as JUnit, iBATIS and Quartz.

Fortunately, it is very possible to fully leverage the strengths of both of these technologies and create a powerful hybrid solution by integrating EJB 3 and Spring. This should be good news if you have a significant investment in Spring but want to utilize the benefits of EJB 3 or are planning on using EJB 3 but would like to utilize Spring features and APIs. We'll talk about Spring/EJB 3 integration in much more detail in chapter 13. Let's take a quick look at some possibilities now.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

1.7.3 Injecting EJBs into Spring beans

It is possible to enable both the Java EE @EJB and @Resource injection annotations in Spring. This means that you can inject any EJB or JNDI managed object into a Spring bean. This comes in very handy to inject EJBs into unit tests as well as Spring beans at the presentation tier. The example below shows Spring injecting an EJB into a JUnit test:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/applicationContext.xml" })
public class AccountServiceTest {
       @EJB(mappedName = "DefaultAccountServiceLocal")
       private AccountService accountService;
```

This technique is shown in figure 1.10.

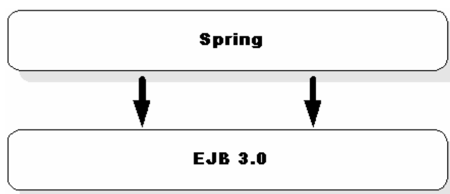

Figure 1.10 A Spring/EJB integration strategy. It is possible to inject EJB 3 beans into Spring beans.

This enables you to use the complementary strengths of both technologies.

1.7.4 Injecting Spring Beans into EJB 3

From the EJB 3 side, Spring beans can be injected into EJB 3 beans using the `SpringBeanAutowiringInterceptor` interceptor. This interceptor enables the `@Autowired` Spring injection annotation in EJBs. This is very useful for injecting Spring managed DAOs into EJBs in the business logic tier:

```
@Stateless
@Interceptors(SpringBeanAutowiringInterceptor.class)
public class BidService {
    @Autowired
    private BidDao bidDao;
```

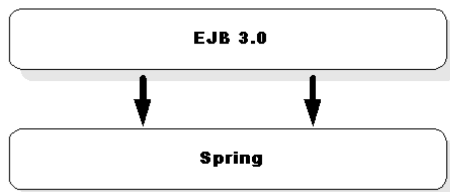

Figure 1.11 Leveraging Spring from EJB

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731

With this approach, depicted in Figure 1.11, you can reuse your existing Spring beans an EJB application. This opens up many interesting opportunities that we will discuss later.

1.8 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 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, and we took a brief look at how EJB 3 can be used with Spring and Seam.

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 EJB 3 API that shows just how easy the code really is.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=731