THE EXPERT'S VOICE® IN JAVA

Beginning Java EE 7

Antonio Goncalves

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

In today's business world, applications need to access data, apply business logic, add presentation layers, be mobile, use geolocalization, and communicate with external systems and online services. That's what companies are trying to achieve while minimizing costs, using standard and robust technologies that can handle heavy loads. If that's your case, you have the right book in your hands.

Java Enterprise Edition appeared at the end of the 1990s and brought to the Java language a robust software platform for enterprise development. Challenged at each new version, badly understood or misused, overengineered, and competing with open source frameworks, J2EE was seen as a heavyweight technology. Java EE benefited from these criticisms to improve and today focuses on simplicity.

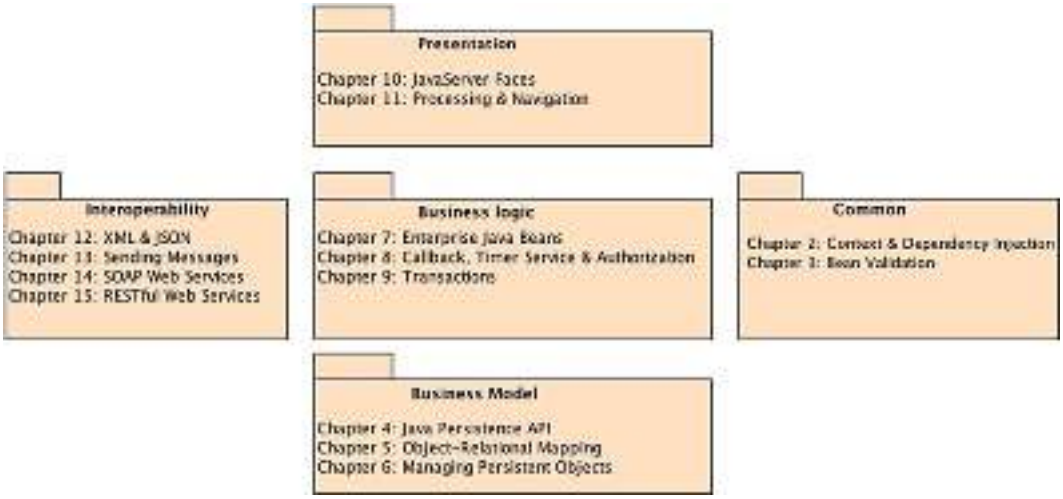
If you are part of the group of people who still think that "EJBs are bad, EJBs are evil," read this book, and you'll change your mind. EJBs (Enterprise Java Beans) are great, as is the entire Java EE 7 technology stack. If, on the contrary, you are a Java EE adopter, you will see in this book how the platform has found equilibrium through its ease of development and easy component model. If you are a beginner in Java EE, this is also the right book: it covers the most important specifications in a very understandable manner and is illustrated with a lot of code and diagrams to make it easier to follow.

Open standards are collectively one of the main strengths of Java EE. More than ever, an application written with JPA, CDI, Bean Validation, EJBs, JSF, JMS, SOAP web services, or RESTful web services is portable across application servers. Open source is another of Java EE's strengths. As you'll see in this book, most of the Java EE 7 Reference Implementations use open source licensing (GlassFish, EclipseLink, Weld, Hibernate Validator, Mojarra, OpenMQ, Metro, and Jersey).

This book explores the innovations of this new version, and examines the various specifications and how to assemble them to develop applications. Java EE 7 consists of nearly 30 specifications and is an important milestone for the enterprise layer (CDI 1.1, Bean Validation 1.1, EJB 3.2, JPA 2.1), for the web tier (Servlet 3.1, JSF 2.2, Expression Language 3.0), and for interoperability (JAX-WS 2.3 and JAX-RS 2.0). This book covers a broad part of the Java EE 7 specifications and uses the JDK 1.7 and some well-known design patterns, as well as the GlassFish application server, the Derby database, JUnit, and Maven. It is illustrated abundantly with UML diagrams, Java code, and screenshots.

How the Book Is Structured

This book concentrates on the most important Java EE 7 specifications and highlights the new features of this release. The structure of the book follows the architectural layering of an application.



Chapter 1 briefly presents Java EE 7 essentials and Appendix A highlights the tools used throughout the book and how to install them (JDK, Maven, JUnit, Derby, and GlassFish).

In the first chapters, I explain the common concerns discussed throughout the book. Chapter 2 describes Context and Dependency Injection 1.1 and Chapter 3 looks at Bean Validation 1.1.

Chapters 4 through 6 describe the persistent tier and focus on JPA 2.1. After a general overview with some hands-on examples in Chapter 4, Chapter 5 dives into object-relational mapping (mapping attributes, relationships, and inheritance), while Chapter 6 shows you how to manage and query entities, their life cycle, callback methods, and listeners.

To develop a transaction business logic layer with Java EE 7, you can naturally use EJBs. Chapters 7 through 9 describe this process. After an overview of the specification and its history, Chapter 7 concentrates on session beans and their programming model. Chapter 8 focuses on the life cycle of EJBs, the timer service, and how to handle authorization. Chapter 9 explains transactions and how JTA 1.2 brings transactions to EJBs as well as CDI Beans.

In Chapters 10 and 11 you will learn how to develop a presentation layer with JSF 2.2. After an overview of the specification, Chapter 10 focuses on how to build a web page with JSF and Facelets components. Chapter 11 shows you how to interact with an EJB back end with CDI backing beans and navigate through pages.

Finally, the last chapters present different ways to interoperate with other systems. Chapter 12 explains how to process XML (using JAXB and JAXP) and JSON (JSON-P 1.0). Chapter 13 shows you how to exchange asynchronous messages with the new JMS 2.0 and Message-Driven Beans. Chapter 14 focuses on SOAP web services, while Chapter 15 covers RESTful web services with the new JAX-RS 2.0.

Downloading and Running the Code

The examples used in this book are designed to be compiled with the JDK 1.7, to be built with Maven 3, to be deployed on GlassFish v4 application server, and to store data in a Derby database. Appendix A shows you how to install all these software packages, and each chapter explains how to build, deploy, run, and test components depending on the technology used. The code has been tested on the Mac OS X platform (but should also work on Windows or Linux). The source code of the examples in the book is available from the Source Code page of the Apress web site at `www.apress.com`. You can also download the code straight from the public GitHub at `https://github.com/agoncal/agoncal-book-javaee7`.

Contacting the Author

If you have any questions about the content of this book, the code, or any other topic, please contact the author at `antonio.goncalves@gmail.com`. You can also visit his web site at `www.antoniogoncalves.org` and follow him on Twitter at @agoncal.

CHAPTER 1

Java EE 7 at a Glance

Enterprises today live in a globally competitive world. They need applications to fulfill their business needs, which are getting more and more complex. In this age of globalization, companies are distributed over continents, they do business 24/7 over the Internet and across different countries, have several datacenters, and internationalized systems which have to deal with different currencies and time zones—all that while reducing their costs, lowering the response times of their services, storing business data on reliable and safe storage, and offering several mobile and web interfaces to their customers, employees, and suppliers.

Most companies have to combine these complex challenges with their existing enterprise information systems (EIS) at the same time developing business-to-business applications to communicate with partners or business-to-customer systems using mobile and geolocalized applications. It is also not rare for a company to have to coordinate in-house data stored in different locations, processed by multiple programming languages, and routed through different protocols. And, of course, it has to do this without losing money, which means preventing system crashes and being highly available, scalable, and secure. Enterprise applications have to face change and complexity, and be robust. That's precisely why Java Enterprise Edition (Java EE) was created.

The first version of Java EE (originally known as J2EE) focused on the concerns that companies were facing back in 1999: distributed components. Since then, software applications have had to adapt to new technical solutions like SOAP or RESTful web services. The Java EE platform has evolved to respond to these technical needs by providing various ways of working through standard specifications. Throughout the years, Java EE has changed and became richer, simpler, easier to use, more portable, and more integrated.

In this chapter, I'll give you an overview of Java EE. After an introduction to its internal architecture, components, and services, I'll cover what's new in Java EE 7.

Understanding Java EE

When you want to handle collections of objects, you don't start by developing your own hashtable; you use the collection API (application programming interface). Similarly, if you need a simple Web application or a transactional, secure, interoperable, and distributed application, you don't want to develop all the low-level APIs: you use the Enterprise Edition of Java. Just as Java Standard Edition (Java SE) provides an API to handle collections, Java EE provides a standard way to handle transactions with Java Transaction API (JTA), messaging with Java Message Service (JMS), or persistence with Java Persistence API (JPA). Java EE is a set of specifications intended for enterprise applications. It can be seen as an extension of Java SE to facilitate the development of distributed, robust, powerful, and highly available applications.

Java EE 7 is an important milestone. Not only does it follow in the footsteps of Java EE 6 by focusing on an easier development model, but it also adds new specifications, as well as adding new functionalities to existing ones. Moreover, Context and Dependency Injection (CDI) is becoming the integration point between all these new specifications. The release of Java EE 7 coincides closely with the 13th anniversary of the enterprise platform. It combines the advantages of the Java language with experience gained over the last 13 years. Java EE profits from the dynamism of open source communities as well as the rigor of the JCP (Java Community Process) standardization process. Today Java EE is a well-documented platform with experienced developers, a large community, and many

deployed applications running on companies' servers. Java EE is a suite of APIs that can be used to build standard component-based multitier applications. These components are deployed in different containers offering a series of services.

Architecture

Java EE is a set of specifications implemented by different containers. Containers are Java EE runtime environments that provide certain services to the components they host such as life-cycle management, dependency injection, concurrency, and so on. These components use well-defined contracts to communicate with the Java EE infrastructure and with the other components. They need to be packaged in a standard way (following a defined directory structure that can be compressed into archive files) before being deployed. Java EE is a superset of the Java SE platform, which means Java SE APIs can be used by any Java EE components.

Figure 1-1 shows the logical relationships between containers. The arrows represent the protocols used by one container to access another. For example, the web container hosts servlets, which can access EJBs through RMI-IIOP.

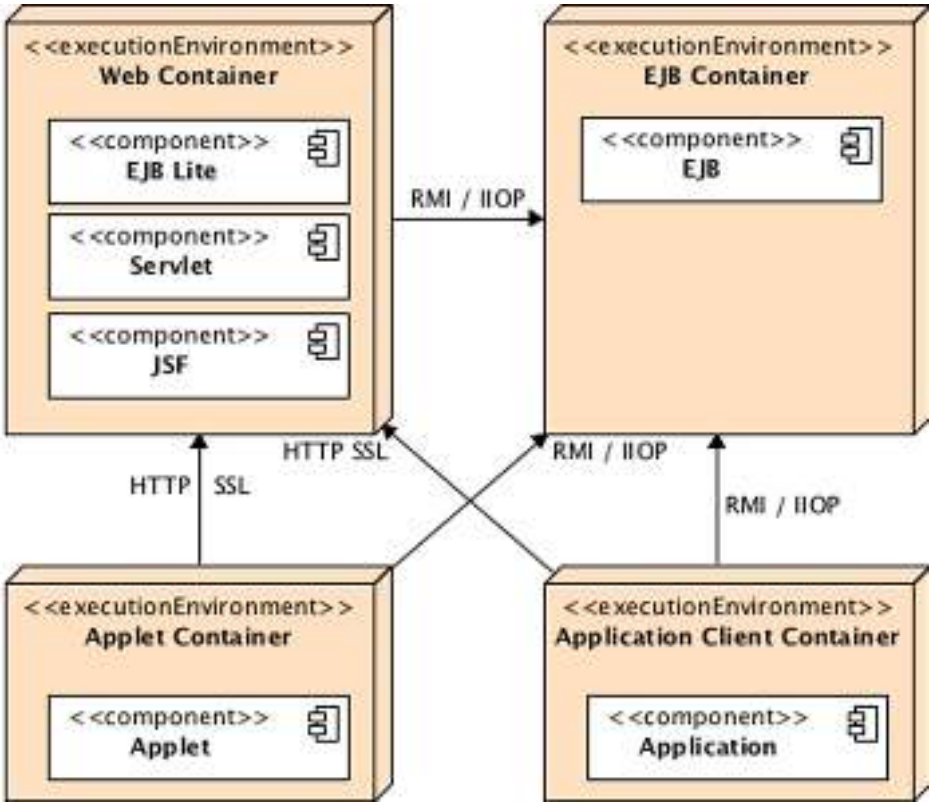


Figure 1-1. *Standard Java EE containers*

Components

The Java EE runtime environment defines four types of components that an implementation must support:

- *Applets* are GUI (graphic user interface) applications that are executed in a web browser. They use the rich Swing API to provide powerful user interfaces.
- *Applications* are programs that are executed on a client. They are typically GUIs or batch-processing programs that have access to all the facilities of the Java EE middle tier.
- *Web applications* (made of servlets, servlet filters, web event listeners, JSP and JSF pages) are executed in a web container and respond to HTTP requests from web clients. Servlets also support SOAP and RESTful web service endpoints. Web applications can also contain EJBs Lite (more on that in Chapter 7).
- *Enterprise applications* (made of Enterprise Java Beans, Java Message Service, Java Transaction API, asynchronous calls, timer service, RMI/IIOP) are executed in an EJB container. EJBs are container-managed components for processing transactional business logic. They can be accessed locally and remotely through RMI (or HTTP for SOAP and RESTful web services).

Containers

The Java EE infrastructure is partitioned into logical domains called containers (see Figure 1-1). Each container has a specific role, supports a set of APIs, and offers services to components (security, database access, transaction handling, naming directory, resource injection). Containers hide technical complexity and enhance portability. Depending on the kind of application you want to build, you will have to understand the capabilities and constraints of each container in order to use one or more. For example, if you need to develop a web application, you will develop a JSF tier with an EJB Lite tier and deploy them into a web container. But if you want a web application to invoke a business tier remotely and use messaging and asynchronous calls, you will need both the web and EJB containers. Java EE has four different containers:

- *Applet containers* are provided by most web browsers to execute applet components. When you develop applets, you can concentrate on the visual aspect of the application while the container gives you a secure environment. The applet container uses a sandbox security model where code executed in the “sandbox” is not allowed to “play outside the sandbox.” This means that the container prevents any code downloaded to your local computer from accessing local system resources, such as processes or files.
- The *application client container* (ACC) includes a set of Java classes, libraries, and other files required to bring injection, security management, and naming service to Java SE applications (swing, batch processing, or just a class with a `main()` method). The ACC communicates with the EJB container using RMI-IIOP and the web container with HTTP (e.g., for web services).
- The *web container* provides the underlying services for managing and executing web components (servlets, EJBs Lite, JSPs, filters, listeners, JSF pages, and web services). It is responsible for instantiating, initializing, and invoking servlets and supporting the HTTP and HTTPS protocols. It is the container used to feed web pages to client browsers.
- The *EJB container* is responsible for managing the execution of the enterprise beans (session beans and message-driven beans) containing the business logic tier of your Java EE application. It creates new instances of EJBs, manages their life cycle, and provides services such as transaction, security, concurrency, distribution, naming service, or the possibility to be invoked asynchronously.

Services

Containers provide underlying services to their deployed components. As a developer, containers allow you to concentrate on implementing business logic rather than solving technical problems faced in enterprise applications. Figure 1-2 shows you the services provided by each container. For example, web and EJB containers provide connectors to access EIS, but not the applet container or the ACCs. Java EE offers the following services:

- *Java Transaction API*: This service offers a transaction demarcation API used by the container and the application. It also provides an interface between the transaction manager and a resource manager at the Service Provider Interface (SPI) level.
- *Java Persistence API*: Standard API for object-relational mapping (ORM). With its Java Persistence Query Language (JPQL), you can query objects stored in the underlying database.
- *Validation*: Bean Validation provides class and method level constraint declaration and validation facilities.
- *Java Message Service*: This allows components to communicate asynchronously through messages. It supports reliable point-to-point (P2P) messaging as well as the publish-subscribe (pub-sub) model.
- *Java Naming and Directory Interface*: This API, included in Java SE, is used to access naming and directory systems. Your application uses it to associate (bind) names to objects and then to find these objects (lookup) in a directory. You can look up data sources, JMS factories, EJBs, and other resources. Omnipresent in your code until J2EE 1.4, JNDI is used in a more transparent way through injection.
- *JavaMail*: Many applications require the ability to send e-mails, which can be implemented through use of the JavaMail API.
- *JavaBeans Activation Framework*: The JAF API, included in Java SE, provides a framework for handling data in different MIME types. It is used by JavaMail.
- *XML processing*: Most Java EE components can be deployed with optional XML deployment descriptors, and applications often have to manipulate XML documents. The Java API for XML Processing (JAXP) provides support for parsing documents with SAX and DOM APIs, as well as for XSLT. The Streaming API for XML (StAX) provides a pull-parsing API for XML.
- *JSON processing*: New in Java EE 7 the Java API for JSON Processing (JSON-P) allows applications to parse, generate, transform, and query JSON.
- *Java EE Connector Architecture*: Connectors allow you to access EIS from a Java EE component. These could be databases, mainframes, or enterprise resource planning (ERP) programs.
- *Security services*: Java Authentication and Authorization Service (JAAS) enables services to authenticate and enforce access controls upon users. The Java Authorization Service Provider Contract for Containers (JACC) defines a contract between a Java EE application server and an authorization service provider, allowing custom authorization service providers to be plugged into any Java EE product. Java Authentication Service Provider Interface for Containers (JASPIC) defines a standard interface by which authentication modules may be integrated with containers so that these modules may establish the authentication identities used by containers.
- *Web services*: Java EE provides support for SOAP and RESTful web services. The Java API for XML Web Services (JAX-WS), replacing the Java API for XML-based RPC (JAX-RPC), provides support for web services using the SOAP/HTTP protocol. The Java API for RESTful Web Services (JAX-RS) provides support for web services using the REST style.

- *Dependency Injection*: Since Java EE 5, some resources (data sources, JMS factories, persistence units, EJBs . . .) can be injected in managed components. Java EE 7 goes further by using CDI as well as the DI (Dependency Injection for Java) specifications.
- *Management*: Java EE defines APIs for managing containers and servers using a special management enterprise bean. The Java Management Extensions (JMX) API is also used to provide some management support.
- *Deployment*: The Java EE Deployment Specification defines a contract between deployment tools and Java EE products to standardize application deployment.

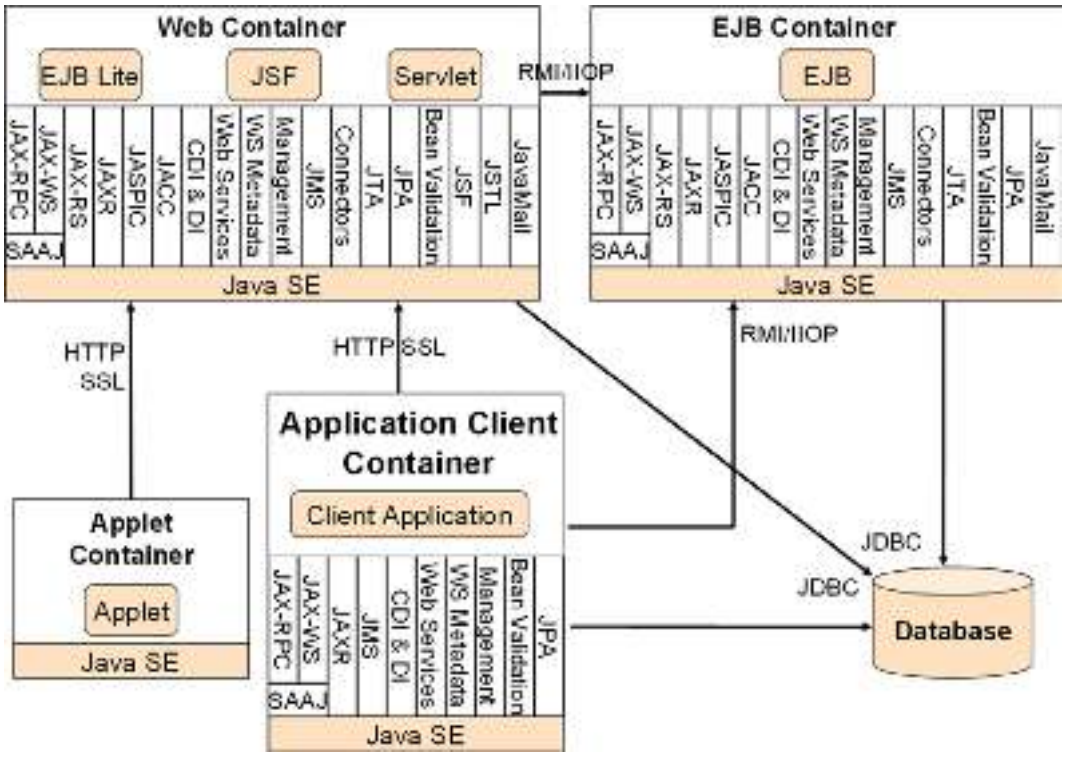


Figure 1-2. *Services provided by containers*

Network Protocols

As shown in Figure 1-2, components deployed in containers can be invoked through different protocols. For example, a servlet deployed in a web container can be called with HTTP as well as a web service with an EJB endpoint deployed in an EJB container. Here is the list of protocols supported by Java EE:

- *HTTP*: HTTP is the Web protocol and is ubiquitous in modern applications. The client-side API is defined by the `java.net` package in Java SE. The HTTP server-side API is defined by servlets, JSPs, and JSF interfaces, as well as SOAP and RESTful web services.
- *HTTPS* is a combination of HTTP and the Secure Sockets Layer (SSL) protocol.

- *RMI-IIOP*: Remote Method Invocation (RMI) allows you to invoke remote objects independently of the underlying protocol. The Java SE native RMI protocol is Java Remote Method Protocol (JRMP). RMI-IIOP is an extension of RMI used to integrate with CORBA. Java interface description language (IDL) allows Java EE application components to invoke external CORBA objects using the IIOP protocol. CORBA objects can be written in many languages (Ada, C, C++, Cobol, etc.) as well as Java.

Packaging

To be deployed in a container, components have first to be packaged in a standard formatted archive. Java SE defines Java Archive (jar) files, which are used to aggregate many files (Java classes, deployment descriptors, resources, or external libraries) into one compressed file (based on the ZIP format). As seen in Figure 1-3, Java EE defines different types of modules that have their own packaging format based on this common jar format.

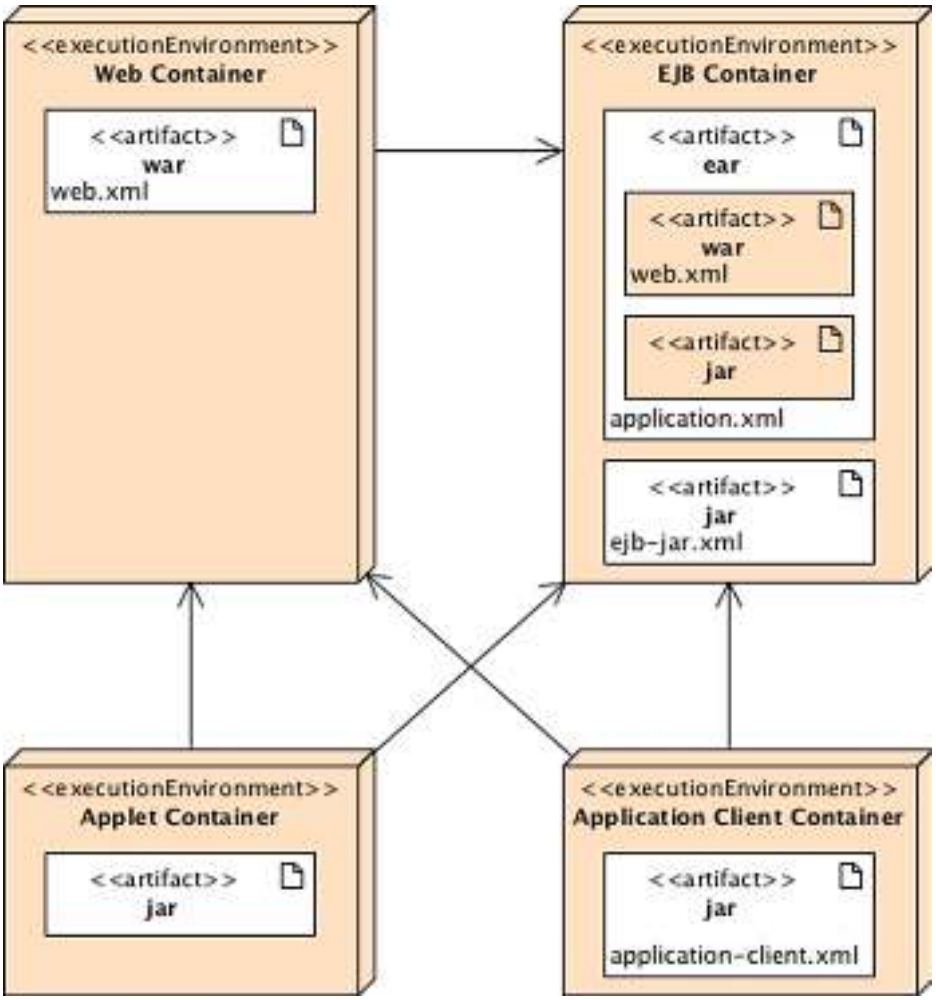


Figure 1-3. *Archives in containers*

- An application client module contains Java classes and other resource files packaged in a jar file. This jar file can be executed in a Java SE environment or in an application client container. Like any other archive format, the jar file contains an optional `META-INF` directory for meta information describing the archive. The `META-INF/MANIFEST.MF` file is used to define extension- and package-related data. If deployed in an ACC, the deployment descriptor can optionally be located at `META-INF/application-client.xml`.
- An EJB module contains one or more session and/or message-driven beans (MDBs) packaged in a jar file (often called an EJB jar file). It contains an optional `META-INF/ejb-jar.xml` deployment descriptor and can be deployed only in an EJB container.
- A web application module contains servlets, JSPs, JSF pages, and web services, as well as any other web-related files (HTML and XHTML pages, Cascading Style Sheets (CSS), Java-Scripts, images, videos, and so on). Since Java EE 6, a web application module can also contain EJB Lite beans (a subset of the EJB API described in Chapter 7). All these artifacts are packaged in a jar file with a `.war` extension (commonly referred to as a war file, or a Web Archive). The optional web deployment descriptor is defined in the `WEB-INF/web.xml` file. If the war contains EJB Lite beans, an optional deployment descriptor can be set at `WEB-INF/ejb-jar.xml`. Java `.class` files are placed under the `WEB-INF/classes` directory and dependent jar files in the `WEB-INF/lib` directory.
- An enterprise module can contain zero or more web application modules, zero or more EJB modules, and other common or external libraries. All this is packaged into an enterprise archive (a jar file with an `.ear` extension) so that the deployment of these various modules happens simultaneously and coherently. The optional enterprise module deployment descriptor is defined in the `META-INF/application.xml` file. The special `lib` directory is used to share common libraries between the modules.

Annotations and Deployment Descriptors

In programming paradigm, there are two approaches: imperative programming and declarative programming. Imperative programming specifies the algorithm to achieve a goal (*what has to be done*), whereas declarative programming specifies how to achieve this goal (*how it has to be done*). In Java EE, declarative programming is done by using metadata, that is, annotations or/and deployment descriptors.

As you've seen in Figure 1-2, components run in a container and this container gives the component a set of services. Metadata are used to declare and customize these services and associates additional information along with Java classes, interfaces, constructors, methods, fields or parameters.

Since Java EE 5, annotations have been proliferating in the enterprise platform. They decorate your code (Java classes, interfaces, fields, methods . . .) with metadata information. Listing 1-1 shows a POJO (Plain Old Java Object) that declares certain behavior using annotations on the class and on an attribute (more on EJBs, persistence context and annotations in the coming chapters).

Listing 1-1. An EJB with Annotations

```
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {

  @PersistenceContext(unitName = "chapter01PU")
  private EntityManager em;

  public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }
}
```

The other manner of declaring metadata is by using deployment descriptors. A deployment descriptor (DD) refers to an XML configuration file that is deployed with the component in the container. Listing 1-2 shows an EJB deployment descriptor. Like most of the Java EE 7 deployment descriptors, it defines the http://xmlns.jcp.org/xml/ns/javaee namespace and contains a version attribute with the version of the specification.

Listing 1-2. An EJB Deployment Descriptor

```
<ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee" ➥
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
                             http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd" ➥
         version="3.2">

  <enterprise-beans>
    <session>
      <ejb-name>ItemEJB</ejb-name>
      <remote>org.agoncal.book.javaee7.ItemRemote</remote>
      <local>org.agoncal.book.javaee7.ItemLocal</local>
      <local-bean/>
      <ejb-class>org.agoncal.book.javaee7.ItemEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Deployment descriptors need to be packaged with the components in the special META-INF or WEB-INF directory to be taken in account. Table 1-1 shows the list of the Java EE deployment descriptors and the related specification (more on that in the coming chapters).

Table 1-1. *Deployment Descriptors in Java EE*

File	Specification	Paths
application.xml	Java EE	META-INF
application-client.xml	Java EE	META-INF
beans.xml	CDI	META-INF or WEB-INF
ra.xml	JCA	META-INF
ejb-jar.xml	EJB	META-INF or WEB-INF
faces-config.xml	JSF	WEB -INF
persistence.xml	JPA	META-INF
validation.xml	Bean Validation	META-INF or WEB-INF
web.xml	Servlet	WEB-INF
web-fragment.xml	Servlet	WEB-INF
webservices.xml	SOAP Web Services	META-INF or WEB-INF

Since Java EE 5 most deployment descriptors are optional and you can use annotations instead. But you can also use the best of both for your application. The biggest advantage of annotations is that they significantly reduce the amount of code a developer needs to write, and by using annotations you can avoid the need for deployment descriptors. On the other hand, deployment descriptors are external XML files that can be changed without requiring modifications to source code and recompilation. If you use both, then the metadata are overridden by the deployment descriptor (i.e., XML takes precedence over annotations) when the application or component is deployed.

■ **Note** In today's development annotations are preferred over deployment descriptors in Java EE. That is because there is a trend to replace a dual language programming (Java + XML) with only one (Java). This is also true because it's easy to analyze and prototype an application, when everything (data, methods, and metadata with annotations) is in one place.

Java EE uses the notion of Programming by Exception (a.k.a. Convention over Configuration) so that most of the common behavior doesn't need to be declared with metadata ("programming metadata is the exception, the container takes care of the defaults"). Which means that with only a small amount of annotations or XML the container can give you a default set of services with default behavior.

Standards

Java EE is based on standards. This means that Java EE goes through the standardizing process of the JCP and is described in a specification. In fact, Java EE is called an *umbrella specification* because it bundles together a number of other specifications (or Java Specification Requests). You might ask why standards are so important, as some of the most successful Java frameworks are not standardized (Struts, Spring, etc.). Throughout history, humans have created standards to ease communication and exchange. Some notable examples are language, currency, time, navigation, measurements, tools, railways, electricity, telegraphs, telephones, protocols, and programming languages.

In the early days of Java, if you were doing any kind of web or enterprise development, you were living in a proprietary world by creating your own frameworks or locking yourself to a proprietary commercial framework. Then came the days of open source frameworks, which are not always based on open standards. You can use open source and be locked to a single implementation, or use open source that implements standards and be portable. Java EE provides open standards that are implemented by several commercial (WebLogic, Websphere, MQSeries, etc.) or open source (GlassFish, JBoss, Hibernate, Open JPA, Jersey, etc.) frameworks for handling transactions, security, stateful components, object persistence, and so on. Today, more than ever in the history of Java EE, your application can be deployed to any compliant application server with very few changes.

JCP

The JCP is an open organization, created in 1998 by Sun Microsystems, that is involved in the definition of future versions and features of the Java platform. When the need for standardizing an existing component or API is identified, the initiator (a.k.a. specification lead) creates a JSR and forms a group of experts. This group, made of companies' representatives, organizations, universities, or private individuals, is responsible for the development of the JSR and has to deliver:

- One or more specifications that explain the details and define the fundamentals of the JSR (Java Specification Request),
- A *Reference Implementation* (RI), which is an actual implementation of the specification,
- *Compatibility Test Kit* (a.k.a. *Technology Compatibility Kit*, or TCK), which is a set of tests every implementation needs to pass before claiming to conform to the specification.

Once approved by the executive committee (EC), the specification is released to the community for implementation.

Portable

From its creation, the aim of Java EE was to enable the development of an application and its deployment to any application server without changing the code or the configuration files. This was never as easy as it seemed. Specifications don't cover all the details, and implementations end up providing nonportable solutions. That's what happened with JNDI names, for example. If you deployed an EJB to GlassFish, JBoss, or WebLogic, the JNDI name was different because it wasn't part of the specification, so you had to change your code depending on the application server you used. That particular problem, for example, was fixed in Java EE by specifying a syntax for JNDI names.

Today, the platform has introduced more portable configuration properties than ever, thus increasing portability. Despite having deprecated some APIs (pruning), Java EE applications keep their backward compatibility, letting you migrate your application to newer versions of an application server without too many problems.

Programming Model

Most of the Java EE 7 specifications use the same programming model. It's usually a POJO with some metadata (annotations or XML) deployed into a container. Most of the time the POJO doesn't even implement an interface or extend a superclass. Thanks to the metadata, the container knows which services to apply to this deployed component.

In Java EE 7, servlets, JSF backing beans, EJBs, entities, SOAP and REST web services are annotated classes with optional XML deployment descriptors. Listing 1-3 shows a JSF backing bean that turns out to be a Java class with a single CDI annotation.

Listing 1-3. A JSF Backing Bean

```
@Named
public class BookController {

  @Inject
  private BookEJB bookEJB;

  private Book book = new Book();
  private List<Book> bookList = new ArrayList<Book>();

  public String doCreateBook() {
    book = bookEJB.createBook(book);
    bookList = bookEJB.findBooks();
    return "listBooks.xhtml";
  }

  // Getters, setters
}
```

EJBs also follow the same model. As shown in Listing 1-4, if you need to access an EJB locally, a simple annotated class with no interface is enough. EJBs can also be deployed directly in a war file without being previously packaged in a jar file. This makes EJBs the simplest transactional component that can be used from simple web applications to complex enterprise ones.

Listing 1-4. A Stateless EJB

```
@Stateless
public class BookEJB {

  @Inject
  private EntityManager em;

  public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }

  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }
}
```

RESTful web services have been making their way into modern applications. Java EE 7 attends to the needs of enterprises by improving the JAX-RS specification. As shown in Listing 1-5, a RESTful web service is an annotated Java class that responds to HTTP actions (more in Chapter 15).

Listing 1-5. A RESTful Web Service

```
@Path("books")
public class BookResource {

  @Inject
  private EntityManager em;

  @GET
  @Produces({"application/xml", "application/json"})
  public List<Book> getAllBooks() {
    Query query = em.createNamedQuery("findAllBooks");
    List<Book> books = query.getResultList();
    return books;
  }
}
```

Throughout the chapters of this book you will come across this kind of code where components only contain business logic and where metadata are represented by annotations (or XML) to ensure that the container applies the right services.

Java Standard Edition 7

It's important to stress that Java EE is a superset of Java SE. This means that all the features of the Java language are available in Java EE as well as the APIs.

Java SE 7 was officially released on July 2011. It was developed under JSR 336 and brought many new features as well as continuing the ease of development introduced by Java SE 5 (autoboxing, annotations, generics, enumeration, etc.) and Java SE 6 (diagnosing, managing, and monitoring tools, JMX API, simplified execution of scripting languages in the Java Virtual Machine). Java SE 7 aggregates the JSR 334 (better known under the name of Project Coin), JSR 292 (InvokeDynamic, or support of dynamic languages in the JVM), JSR 203 (the new API I / O, commonly called NIO.2) and several updates of existing specifications (such as JDBC 4.1 (JSR 221). Even if this book does not explicitly cover Java SE 7, some of these enhancements will be used throughout the book samples so I just want to give you a quick overview of what the samples could look like.

String Case

Before Java SE 7 only numbers (byte, short, int, long, char) or enumerations could be used in switch cases. It is now possible to use a switch on a Strcompare alphanumerical values. This avoids long lists of if/then/else and makes the code more readable. Listing 1-6 shows you what you can now write in your applications.

Listing 1-6. A String Case

```
String action = "update";
switch (action) {
  case "create":
    create();
    break;
  case "read":
    read();
    break;
  case "udpate":
    udpate();
    break;
```

```
  case "delete":
    delete();
    break;
  default:
    noCrudAction(action);
}
```

Diamond

Generics arrived with Java SE 5 with a rather verbose syntax. Java SE 7 brought a slightly lighter notation, called diamond, which does not repeat the declaration in the instantiation of an object. Listing 1-7 gives an example of declaring generics both with and without the diamond operator.

Listing 1-7. Declaring Generics with and Without Diamond

```
// Without diamond operator
List<String> list = new ArrayList<String>();
Map<Reference<Object>, Map<Integer, List<String>>> map =
    new HashMap<Reference<Object>, Map<Integer, List<String>>>();

// With diamond operator
List<String> list = new ArrayList<>();
Map<Reference<Object>, Map<Integer, List<String>>> map = new HashMap<>();
```

Try-with-Resources

In several Java APIs, closing resources have to be managed manually, usually by a call to a close method in a finally block. This is the case for resources managed by the operating system such as files, sockets, or JDBC connections. Listing 1-8 shows how it is necessary to put the closing code in a finally block with exception handling, which decreases the readability of the code.

Listing 1-8. Closing Input/Output Streams in Finally Blocks

```
try {
  InputStream input = new FileInputStream(in.txt);
  try {
    OutputStream output = new FileOutputStream(out.txt);
    try {
      byte[] buf = new byte[1024];
      int len;
      while ((len = input.read(buf)) >= 0)
        output.write(buf, 0, len);
    } finally {
      output.close();
    }
  } finally {
    input.close();
  }
} catch (IOException e) {
  e.printStrackTrace();
}
```

The try-with-resources overcomes this readability problem via a new simpler syntax. It allows the resources in the `try` to be automatically released at the end of the block. This notation described in Listing 1-9 can be used for any class that implements the new interface `java.lang.AutoCloseable`. This interface is now implemented by multiple classes (`InputStream`, `OutputStream`, `JarFile`, `Reader`, `Writer`, `Socket`, `ZipFile`...) and interfaces (`java.sql.ResultSet`).

Listing 1-9. Closing Input/Output Streams with Try-with-Resources

```
try (InputStream input = new FileInputStream(in.txt);
     OutputStream output = new FileOutputStream(out.txt)) {
  byte[] buf = new byte[1024];
  int len;
  while ((len = input.read(buf)) >= 0)
    output.write(buf, 0, len);
} catch (IOException e) {
  e.printStrackTrace();
}
```

Multicatch Exception

Until Java SE 6 the catch block could handle only one type of exception at a time. You therefore had to accumulate several catches to perform a specific action for each type of exception. And as shown in Listing 1-10 you often have to perform the same action for each exception.

Listing 1-10. Using Several Catch Exception Clauses

```
try {
  // Do something
} catch(SAXException e) {
  e.printStackTrace();
} catch(IOException e) {
  e.printStackTrace();
} catch(ParserConfigurationException e) {
  e.printStackTrace();
}
```

With Java SE 7 if the handling of each exception is identical, you can add as many exception types as you want, separated by a pipe character as shown in Listing 1-11.

Listing 1-11. Using Multicatch Exception

```
try {
  // Do something
} catch(SAXException | IOException | ParserConfigurationException e) {
  e.printStackTrace();
}
```

NIO.2

If like many Java developers you struggle each time you have to read or write a file, Java SE 7 came to your rescue by introducing a new IO package: `java.nio`. With a more expressive syntax, its goal is to replace the existing `java.io` package to allow:

- A cleaner exception handling.
- Full access to the file system with new features (support of specific operating system attributes, symbolic links, etc.).
- The addition of the notion of `FileSystem` and `FileStore` (e.g., a partition disk).
- Utility methods (move/copy files, read/write binary or text files, path, directories, etc.).

Listing 1-12 shows you the new `java.nio.file.Path` interface (used to locate a file or a directory in a file system) as well as the utility class `java.nio.file.Files` (used to get information about the file or to manipulate it). From Java SE 7 onward it is recommended to use the new NIO.2 even if the old `java.io` package has not been deprecated. The code in Listing 1-12 gets some information about the `source.txt` file, copies it to the `dest.txt` file, displays its content, and deletes it.

Listing 1-12. Using the New IO Package

```
Path path = Paths.get("source.txt");
boolean exists = Files.exists(path);
boolean isDirectory = Files.isDirectory(path);
boolean isExecutable = Files.isExecutable(path);
boolean isHidden = Files.isHidden(path);
boolean isReadable = Files.isReadable(path);
boolean isRegularFile = Files.isRegularFile(path);
boolean isWritable = Files.isWritable(path);
long size = Files.size(path);

// Copies a file
Files.copy(Paths.get("source.txt"), Paths.get("dest.txt"));
// Reads a text file
List<String> lines = Files.readAllLines(Paths.get("source.txt"), UTF_8);
for (String line : lines) {
  System.out.println(line);
}
// Deletes a file
Files.delete(path);
```

Java EE Specifications Overview

Java EE is an umbrella specification that bundles and integrates others. Today, an application server has to implement 31 specifications in order to be compliant with Java EE 7 and a developer has to know thousands of APIs to make the most of the container. Even if there are many specifications and APIs to know, Java EE 7 focuses on bringing simplicity to the platform by introducing a simple programming model based on POJO, a Web profile, and pruning some outdated technologies.

A Brief History of Java EE

Figure 1-4 summarizes 14 years of Java EE evolution. Java EE formerly called J2EE. J2EE 1.2, was first developed by Sun, and was released in 1999 as an umbrella specification containing ten JSRs. At that time people were talking about CORBA, so J2EE 1.2 was created with distributed systems in mind. Enterprise Java Beans (EJBs) were introduced with support for remote stateful and stateless service objects, and optional support for persistent objects (entity beans). They were built on a transactional and distributed component model using RMI-IIOP (Remote Method Invocation–Internet Inter-ORB Protocol) as the underlying protocol. The web tier had servlets and JavaServer Pages (JSPs), and JMS was used for sending messages.

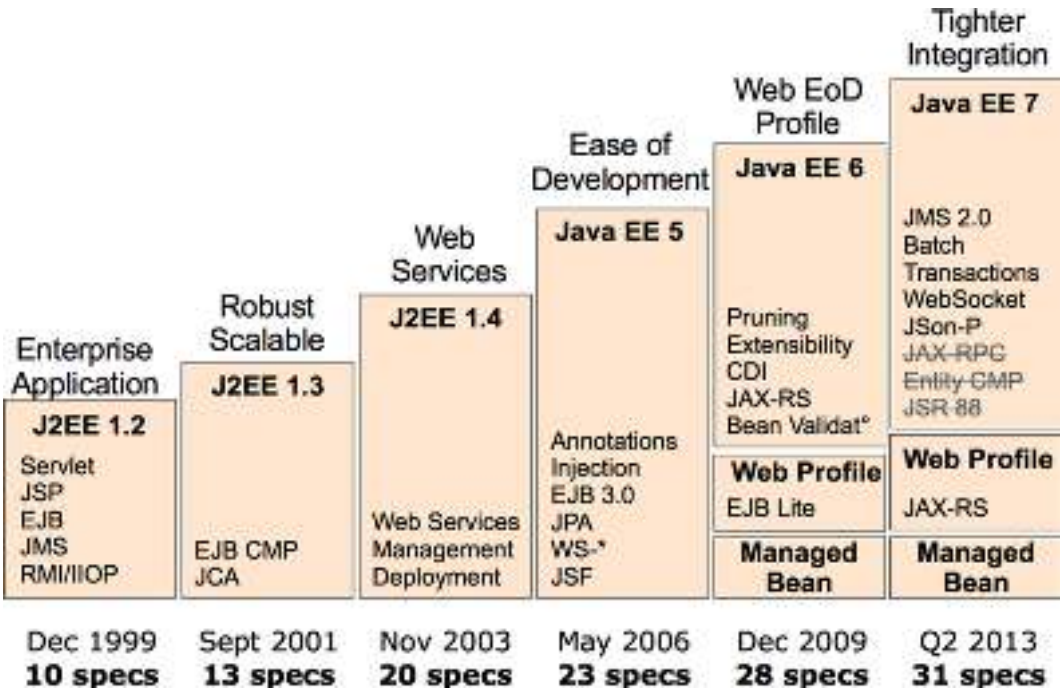


Figure 1-4. *History of J2EE/Java EE*

Starting with J2EE 1.3, the specification was developed by the Java Community Process (JCP) under the JSR 58. Support for entity beans was made mandatory, and EJBs introduced XML deployment descriptors to store metadata (which was serialized in a file in EJB 1.0). This version addressed the overhead of passing arguments by value with remote interfaces, by introducing local interfaces and passing arguments by reference. J2EE Connector Architecture (JCA) was introduced to connect Java EE to EIS.

■ **Note** CORBA originated about 1988 precisely because enterprise systems were beginning to be distributed (e.g., Tuxedo and CICS). EJBs and then J2EE followed on with the same assumptions, but ten years later. By the time J2EE was begun, CORBA was fully backed and at industrial strength, but companies found simpler, more decoupled ways to connect distributed systems, like SOAP or REST web services. So CORBA became redundant for most enterprise systems.

J2EE 1.4 (JSR 151) included 20 specifications in 2003 and added support for web services. EJB 2.1 allowed session beans to be invoked over SOAP/HTTP. A timer service was created to allow EJBs to be invoked at designated times or intervals. This version provided better support for application assembly and deployment. Although its supporters predicted a great future for it, not all of J2EE's promise materialized. The systems created with it were too complicated, and development time was frequently out of all proportion to the complexity of the user's requirements. J2EE was seen as a heavyweight component model: difficult to test, difficult to deploy, difficult to run. That's when frameworks such as Struts, Spring, or Hibernate emerged and showed a new way of developing an enterprise application.

Fortunately, in the second quarter of 2006, Java EE 5 (JSR 244) was released and turned out to be a remarkable improvement. It took some inspiration from open source frameworks by bringing back a POJO programming model. Metadata could be defined with annotations, and XML descriptors became optional. From a developer's point of view, EJB 3 and the new JPA were more of a quantum leap than an evolution of the platform .JavaServer Faces (JSF) was introduced as the standard presentation tier framework, and JAX-WS 2.0 replaced JAX-RPC as the SOAP web services API.

In 2009, Java EE 6 (JSR 316) followed the path of ease of development by embracing the concepts of annotations, POJO programming, and the configuration-by-exception mechanism throughout the platform, including the web tier. It came with a rich set of innovations such as the brand-new JAX-RS 1.1, Bean Validation 1.0, and CDI 1.0; it simplified mature APIs like EJB 3.1, and enriched others such as JPA 2.0 or the EJB timer service. But the major themes for Java EE 6 were portability (through standardizing global JNDI naming, for example), deprecation of some specifications (via pruning), and creating subsets of the platform through profiles.

Today Java EE 7 brings many new specifications (batch processing, websockets, JSON processing) as well as improving the others. Java EE 7 also improves integration between technologies by adopting CDI in most of the specifications. In this book, I want to show you these improvements and how much easier and richer Java Enterprise Edition has become.

Pruning

Java EE was first released in 1999, and ever since, new specifications have been added at each release (as shown previously in Figure 1-4). This became a problem in terms of size, implementation, and adoption. Some features were not well supported or not widely deployed because they were technologically outdated or other alternatives were made available in the meantime. So the expert group decided to propose the removal of some features through pruning. The pruning process (also known as marked for deletion) consists of proposing a list of features for possible removal in the following Java EE release. Note that none of the proposed removal items are actually removed from the current version but could be in the following one. Java EE 6 proposed the following specification and features to be pruned, and they indeed disappeared from Java EE 7:

- *EJB 2.x Entity Beans CMP (was part of JSR 318)*: The complex and heavyweight persistent component model of EJB 2.*x* entity beans has been replaced by JPA.
- *JAX-RPC (JSR 101)*: This was the first attempt to model SOAP web services as RPC calls. It has now been replaced by the much easier to use and robust JAX-WS.
- *JAXR (JSR 93)*: JAXR is the API dedicated to communicating with UDDI registries. Because UDDI is not widely used, JAXR has left Java EE and evolves as a separate JSR.
- *Java EE Application Deployment (JSR 88):* JSR 88 is a specification that tool developers can use for deployment across application servers. This API hasn't gained much vendor support, so it leaves Java EE 7 to evolve as a separate JSR.

Java EE 7 Specifications

The Java EE 7 specification is defined by the JSR 342 and contains 31 other specifications. An application server that aims to be Java EE 7 compliant has to implement all these specifications. Tables 1-2 to 1-6 list them all, grouped by technological domain, with their version and JSR numbers.

Table 1-2. *Java Enterprise Edition Specification*

Specification	Version	JSR	URL
Java EE	7.0	342	http://jcp.org/en/jsr/detail?id=342
Web Profile	7.0	342	http://jcp.org/en/jsr/detail?id=342
Managed Beans	1.0	316	http://jcp.org/en/jsr/detail?id=316

In the web service domain (Table 1-3) no improvement has been made to SOAP web service as no specification has been updated (see Chapter 14). REST web services have been heavily utilized lately in major web applications. JAX-RS 2.0 has followed a major update with the introduction of the client API for example (see Chapter 15). The new JSON-P (JSON Processing) specification is the equivalent of JAXP (Java API for XML Processing) but for JSON instead of XML (Chapter 12).

Table 1-3. *Web Services Specifications*

Specification	Version	JSR	URL
JAX-WS	2.2a	224	http://jcp.org/en/jsr/detail?id=224
JAXB	2.2	222	http://jcp.org/en/jsr/detail?id=222
Web Services	1.3	109	http://jcp.org/en/jsr/detail?id=109
Web Services Metadata	2.1	181	http://jcp.org/en/jsr/detail?id=181
JAX-RS	2.0	339	http://jcp.org/en/jsr/detail?id=339
JSON-P	1.0	353	http://jcp.org/en/jsr/detail?id=353

In the Web specifications (Table 1-4) no change has been made to JSPs or JSTL as these specifications have not been updated. Expression Language has been extracted from JSP and now evolves in its own JSR (341). Servlet and JSF (Chapters 10 and 11) have both been updated and the brand new WebSocket 1.0 has been introduced in Java EE 7.

Table 1-4. *Web Specifications*

Specification	Version	JSR	URL
JSF	2.2	344	http://jcp.org/en/jsr/detail?id=344
JSP	2.3	245	http://jcp.org/en/jsr/detail?id=245
Debugging Support for Other Languages	1.0	45	http://jcp.org/en/jsr/detail?id=45
JSTL (JavaServer Pages Standard Tag Library)	1.2	52	http://jcp.org/en/jsr/detail?id=52
Servlet	3.1	340	http://jcp.org/en/jsr/detail?id=340
WebSocket	1.0	356	http://jcp.org/en/jsr/detail?id=356
Expression Language	3.0	341	http://jcp.org/en/jsr/detail?id=341

In the enterprise domain (Table 1-5) there are two major updates: JMS 2.0 (Chapter 13) and JTA 1.2 (Chapter 9), which hadn't been updated for more than a decade. On the other hand EJBs (Chapters 7 and 8), JPA (Chapters 4, 5 and 6), and Interceptors (Chapter 2) specifications have evolved with minor updates.