Part I Introduction

Part I introduces the platform, the tutorial, and the examples.

This part contains the following chapters:

- Chapter 1, Overview
- Chapter 2, Using the Tutorial Examples

Overview

Developers today increasingly recognize the need for distributed, transactional, and portable applications that leverage the speed, security, and reliability of server-side technology.

Enterprise applications provide the business logic for an enterprise.

They are centrally managed and often interact with other enterprise software.

In the world of information technology, enterprise applications must be designed, built, and produced for less money, with greater speed, and with fewer resources.

With the Java Platform, Enterprise Edition (Java EE), development of Java enterprise applications has never been easier or faster.

The aim of the Java EE platform is to provide developers with a powerful set of APIs while shortening development time, reducing application complexity, and improving application performance.

The Java EE platform is developed through the Java Community Process (the JCP), which is responsible for all Java technologies.

Expert groups, composed of interested parties, have created Java Specification Requests (JSRs) to define the various Java EE technologies.

The work of the Java Community under the JCP program helps to ensure Java technology's standard of stability and cross-platform compatibility.

The Java EE platform uses a simplified programming model.

XML deployment descriptors are optional.

Instead, a developer can simply enter the information as an annotation directly into a Java source file, and the Java EE server will configure the component at deployment and runtime.

These annotations are generally used to embed in a program data that would otherwise be furnished in a deployment descriptor.

With annotations, you put the specification information in your code next to the program element affected.

In the Java EE platform, dependency injection can be applied to all resources that a component needs, effectively hiding the creation and lookup of resources from application code.

Dependency injection can be used in EJB containers, web containers, and application clients.

Dependency injection allows the Java EE container to automatically insert references to other required components or resources, using annotations.

This tutorial uses examples to describe the features available in the Java EE platform for developing enterprise applications.

Whether you are a new or experienced Enterprise developer, you should find the examples and accompanying text a valuable and accessible knowledge base for creating your own solutions.

If you are new to Java EE enterprise application development, this chapter is a good place to start.

Here you will review development basics, learn about the Java EE architecture and APIs, become acquainted with important terms and concepts, and find out how to approach Java EE application programming, assembly, and deployment.

The following topics are addressed here:

- Java EE 6 Platform Highlights
- Java EE Application Model
- Distributed Multitiered Applications
- Java EE Containers
- Web Services Support
- Java EE Application Assembly and Deployment
- Packaging Applications
- Development Roles
- Java EE 6 APIs

- Java EE 6 APIs in the Java Platform, Standard Edition 6.0
- GlassFish Server Tools

Java EE 6 Platform Highlights

The most important goal of the Java EE 6 platform is to simplify development by providing a common foundation for the various kinds of components in the Java EE platform.

Developers benefit from productivity improvements with more annotations and less XML configuration, more Plain Old Java Objects (POJOs), and simplified packaging.

The Java EE 6 platform includes the following new features:

- Profiles: configurations of the Java EE platform targeted at specific classes of applications.

Specifically, the Java EE 6 platform introduces a lightweight Web Profile targeted at next-generation web applications, as well as a Full Profile that contains all Java EE technologies and provides the full power of the Java EE 6 platform for enterprise applications.

- New technologies, including the following:
  - Java API for RESTful Web Services (JAX-RS)
  - Managed Beans
  - Contexts and Dependency Injection for the Java EE Platform (JSR 299), informally known as CDI
  - Dependency Injection for Java (JSR 330)
  - Bean Validation (JSR 303)
  - Java Authentication Service Provider Interface for Containers (JASPIC)

- New features for Enterprise JavaBeans (EJB) components (see Enterprise JavaBeans Technology for details)
- New features for ag
- s (see Java Servlet Technology for details)
- New features for JavaServer Faces components (see JavaServer Faces Technology for details)

Java EE Application Model

The Java EE application model begins with the Java programming language and the Java virtual machine.

The proven portability, security, and developer productivity they provide forms the basis of the application model.

Java EE is designed to support applications that implement enterprise services for customers, employees, suppliers, partners, and others who make demands on or contributions to the enterprise.

Such applications are inherently complex, potentially accessing data from a variety of sources and distributing applications to a variety of clients.

To better control and manage these applications, the business functions to support these various users are conducted in the middle tier.

The middle tier represents an environment that is closely controlled by an enterprise's information technology department.

The middle tier is typically run on dedicated server hardware and has access to the full services of the enterprise.

The Java EE application model defines an architecture for implementing services as multitier applications that deliver the scalability, accessibility, and manageability needed by enterprise-level applications.

This model partitions the work needed to implement a multitier service into the following parts:

- The business and presentation logic to be implemented by the developer
- The standard system services provided by the Java EE platform

The developer can rely on the platform to provide solutions for the hard systems-level problems of developing a multitier service.

Distributed Multitiered Applications

The Java EE platform uses a distributed multitiered application model for enterprise applications.

Application logic is divided into components according to function, and the application components that make up a Java EE application are installed on various machines, depending on the tier in the multitiered Java EE environment to which the application component belongs.

Figure 1-1 shows two multitiered Java EE applications divided into the tiers described in the following list.

The Java EE application parts shown in Figure 1-1 are presented in Java EE Components.

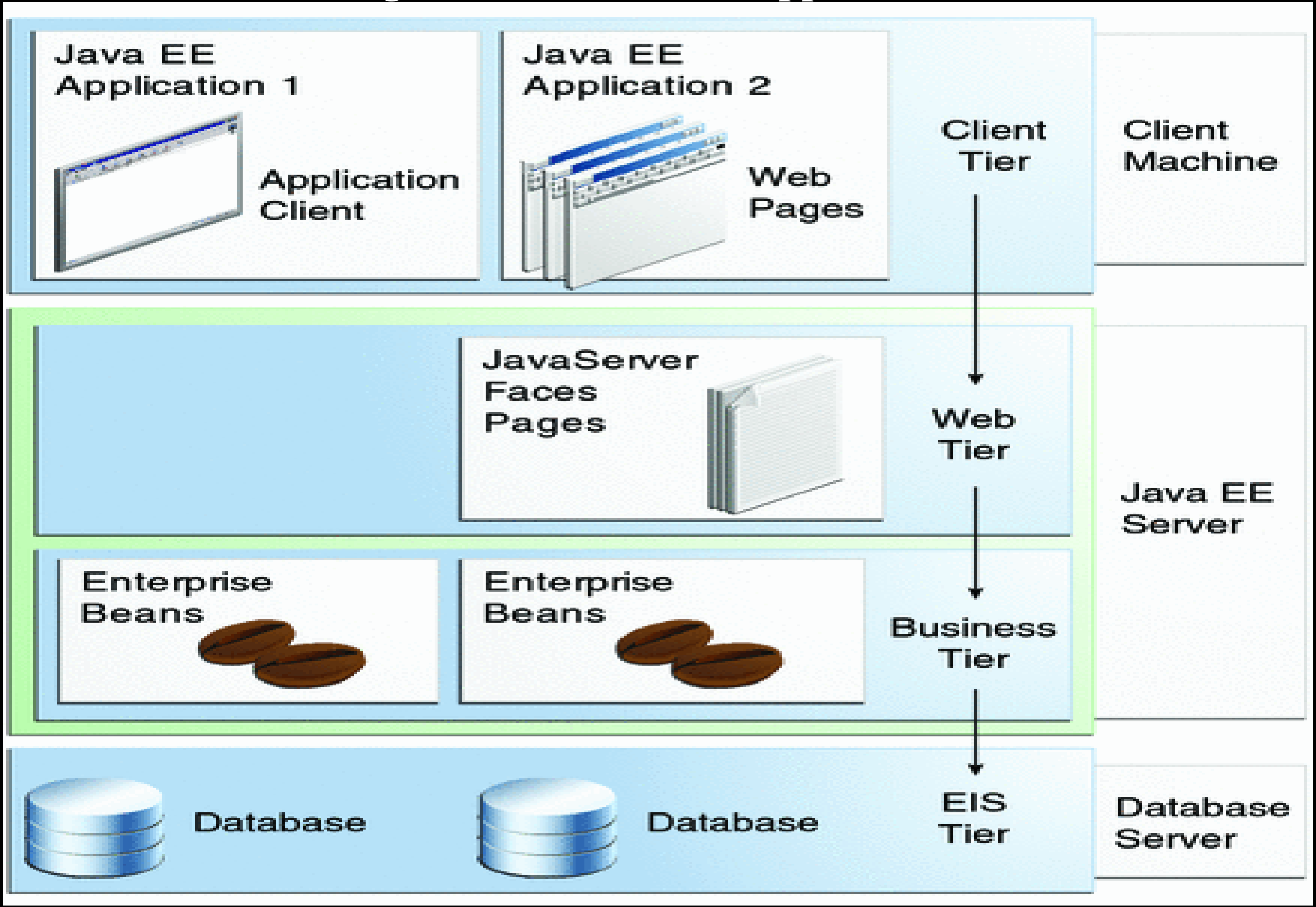
- Client-tier components run on the client machine.
- Web-tier components run on the Java EE server.
- Business-tier components run on the Java EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a Java EE application can consist of the three or four tiers shown in Figure 1-1, Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations:

client machines, the Java EE server machine, and the database or legacy machines at the back end.

Three-tiered applications that run in this way extend the standard two-tiered client-and-server model by placing a multithreaded application server between the client application and back-end storage.

Figure 1-1 Multitiered Applications



Security

Although other enterprise application models require platform-specific security measures in each application, the Java EE security environment enables security constraints to be defined at deployment time.

The Java EE platform makes applications portable to a wide variety of security implementations by shielding application developers from the complexity of implementing security features.

The Java EE platform provides standard declarative access control rules that are defined by the developer and interpreted when the application is deployed on the server.

Java EE also provides standard login mechanisms so application developers do not have to implement these mechanisms in their applications.

The same application works in a variety of security environments without changing the source code.

Java EE Components

Java EE applications are made up of components.

A Java EE component is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.

The Java EE specification defines the following Java EE components.

- Application clients and applets are components that run on the client.
- Java Servlet, JavaServer Faces, and JavaServer Pages (JSP) technology components are web components that run on the server.
- Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

Java EE components are written in the Java programming language and are compiled in the same way as any program in the language.

The difference between Java EE components and "standard" Java classes is that Java EE components are assembled into a Java EE application, are verified to be well formed and in compliance with the Java EE specification, and are deployed to production, where they are run and managed by the Java EE server.

Java EE Clients

A Java EE client is usually either a web client or an application client.

Web Clients

A web client consists of two parts:

- Dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier
- A web browser, which renders the pages received from the server

A web client is sometimes called a thin client.

Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications.

When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the Java EE server, where they can leverage the security, speed, services, and reliability of Java EE server-side technologies.

Application Clients

An application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language.

An application client typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible.

Application clients directly access enterprise beans running in the business tier.

However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier.

Application clients written in languages other than Java can interact with Java EE servers, enabling the Java EE platform to interoperate with legacy systems, clients, and non-Java languages.

Applets

A web page received from the web tier can include an embedded applet.

Written in the Java programming language, an applet is a small client application that executes in the Java virtual machine installed in the web browser.

However, client systems will likely need the Java Plug-in and possibly a security policy file for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program, because no plug-ins or security policy files are needed on the client systems.

Also, web components enable cleaner and more modular application design because they provide a way to separate applications programming from web page design.

Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

The JavaBeans Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between the following:

- An application client or applet and components running on the Java EE server
- Server components and a database

JavaBeans components are not considered Java EE components by the Java EE specification.

JavaBeans components have properties and have `get` and `set` methods for accessing the properties.

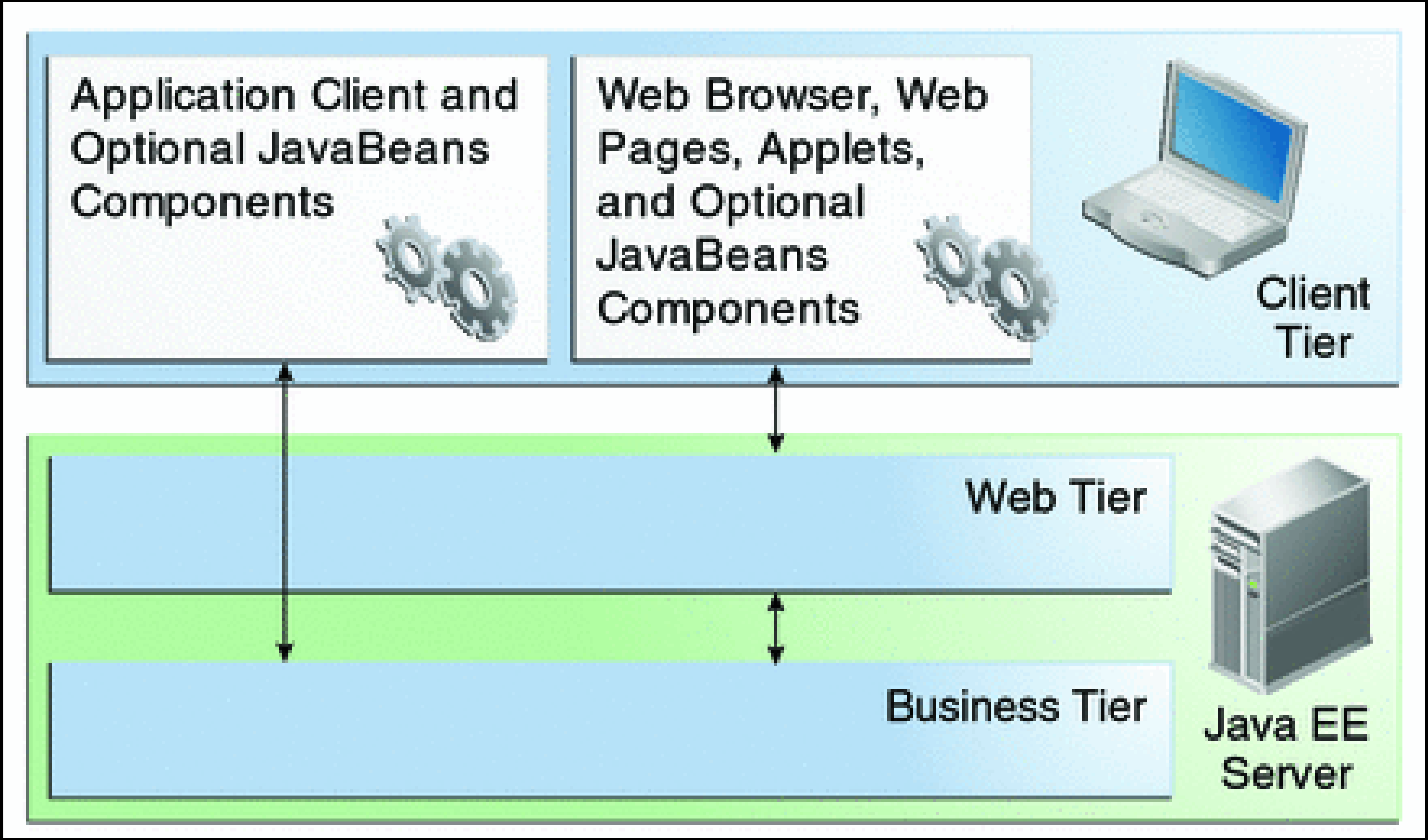
JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

Java EE Server Communications

Figure 1-2 shows the various elements that can make up the client tier.

The client communicates with the business tier running on the Java EE server either directly or, as in the case of a client running in a browser, by going through web pages or servlets running in the web tier.

Figure 1-2 Server Communication


Application Client and Optional JavaBeans Components
Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
Web Tier
Business Tier
Java EE Server

Web Components

Java EE web components are either servlets or web pages created using JavaServer Faces technology and/or JSP technology (JSP pages).

Servlets are Java programming language classes that dynamically process requests and construct responses.

JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content.

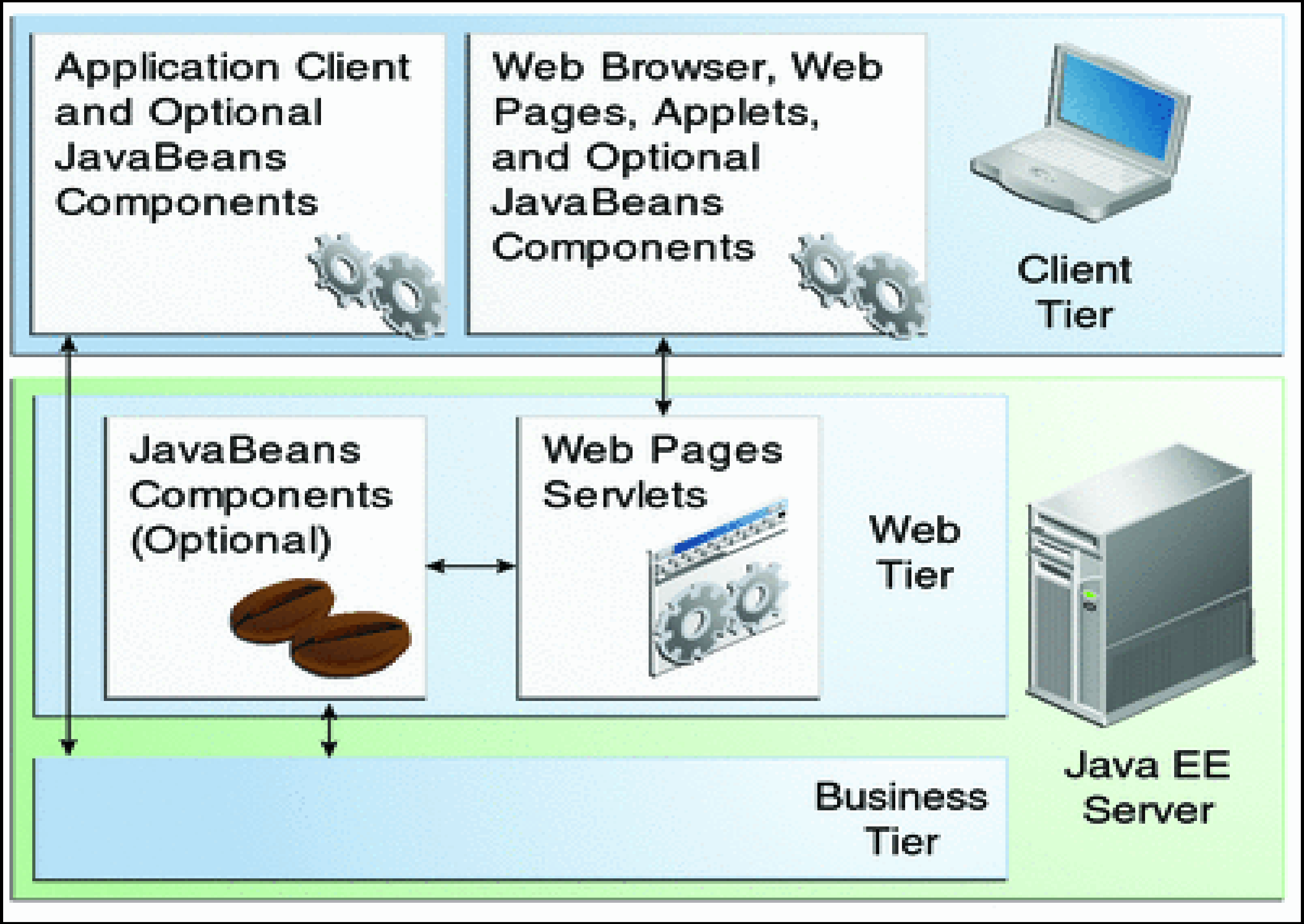
JavaServer Faces technology builds on servlets and JSP technology and provides a user interface component framework for web applications.

Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the Java EE specification.

Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in Figure 1-3, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

Figure 1-3 Web Tier and Java EE Applications


Application Client and Optional JavaBeans Components
Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
JavaBeans Components (Optional)
Web Pages Servlets
Web Tier
Business Tier
Java EE Server

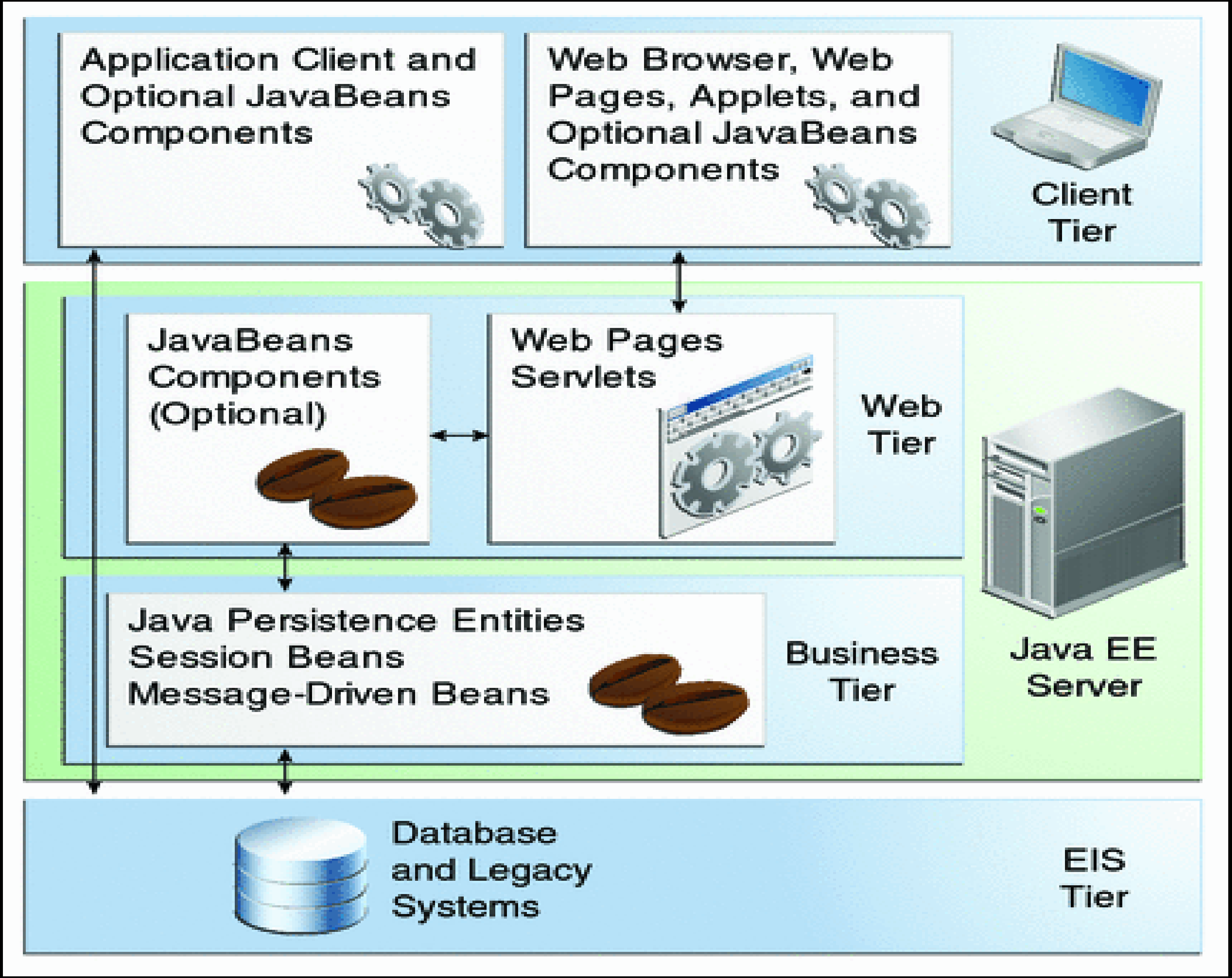
Business Components

Business code, which is logic that solves or meets the needs of a particular business domain, such as banking, retail, or finance, is handled by enterprise beans running in either the business tier or the web tier.

Figure 1-4 shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage.

An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

Figure 1-4 Business and EIS Tiers


Application Client and Optional JavaBeans Components
Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
JavaBeans Components (Optional)
Web Pages Servlets
Web Tier
Java Persistence Entities Session Beans Message-Driven Beans
Business Tier
Java EE Server
Database and Legacy Systems
EIS Tier

Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems, such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems.

For example, Java EE application components might need access to enterprise information systems for database connectivity.

Java EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details.

The component-based and platform-independent Java EE architecture makes Java EE applications easy to write because business logic is organized into reusable components.

In addition, the Java EE server provides underlying services in the form of a container for every component type.

Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component.

Before it can be executed, a web, enterprise bean, or application client component must be assembled into a Java EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the Java EE application and for the Java EE application itself.

Container settings customize the underlying support provided by the Java EE server, including such services as security, transaction management, Java Naming and Directory Interface (JNDI) API lookups, and remote connectivity.

Here are some of the highlights.

- The Java EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
- The Java EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.

- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access these services.
- The Java EE remote connectivity model manages low-level communications between clients and enterprise beans.

After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the Java EE architecture provides configurable services, application components within the same Java EE application can behave differently based on where they are deployed.

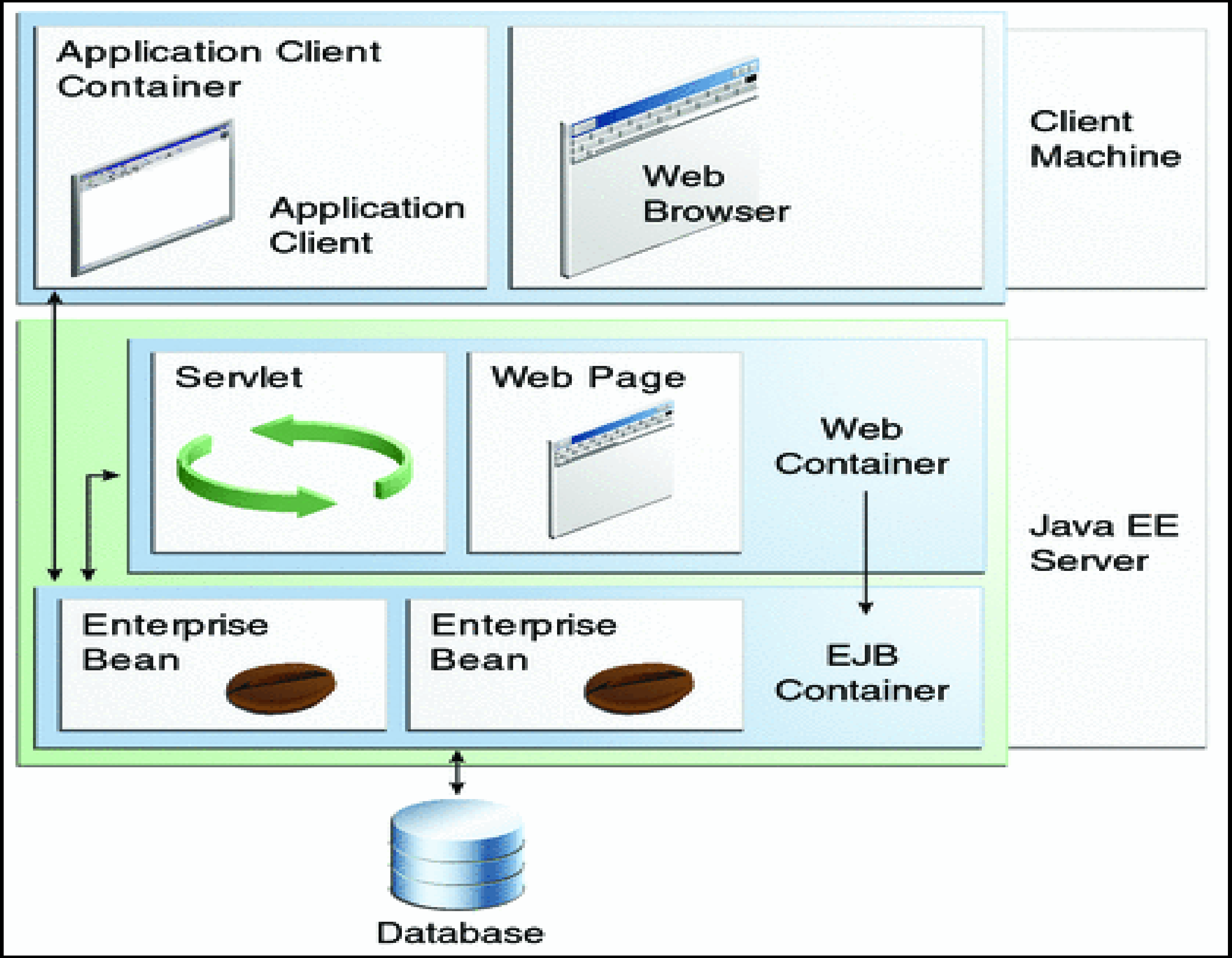
For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services, such as enterprise bean and servlet lifecycles, database connection resource pooling, data persistence, and access to the Java EE platform APIs (see Java EE 6 APIs).

Container Types

The deployment process installs Java EE application components in the Java EE containers as illustrated in Figure 1-5.

Figure 1-5 Java EE Server and Containers



- **Java EE server: The runtime portion of a Java EE product.**

  A Java EE server provides EJB and web containers.

- **Enterprise JavaBeans (EJB) container: Manages the execution of enterprise beans for Java EE applications.**

Enterprise beans and their container run on the Java EE server.

- **Web container**: Manages the execution of web pages, servlets, and some EJB components for Java EE applications.

  Web components and their container run on the Java EE server.

- **Application client container: Manages the execution of application client components.**

  Application clients and their container run on the client.

- **Applet container: Manages the execution of applets.**

Consists of a web browser and Java Plug-in running on the client together.

Web Services Support

Web services are web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients.

The Java EE platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy web services and clients that fully interoperate with other web services and clients running on Java-based or non-Java-based platforms.

To write web services and clients with the Java EE XML APIs, all you do is pass parameter data to the method calls and process the data returned; for document-oriented web services, you send documents containing the service data back and forth.

No low-level programming is needed, because the XML API implementations do the work of translating the application data to and from an XML-based data stream that is sent over the standardized XML-based transport protocols.

These XML-based standards and protocols are introduced in the following sections.

The translation of data to a standardized XML-based data stream is what makes web services and clients written with the Java EE XML APIs fully interoperable.

This does not necessarily mean that the data being transported includes XML tags, because the transported data can itself be plain text, XML data, or any kind of binary data, such as audio, video, maps, program files, computer-aided design (CAD) documents, and the like.

The next section introduces XML and explains how parties doing business can use XML tags and schemas to exchange data in a meaningful way.

XML

Extensible Markup Language (XML) is a cross-platform, extensible, text-based standard for representing data.

Parties that exchange XML data can create their own tags to describe the data, set up schemas to specify which tags can be used in a particular kind of XML document, and use XML style sheets to manage the display and handling of the data.

For example, a web service can use XML and a schema to produce price lists, and companies that receive the price lists and schema can have their own style sheets to handle the data in a way that best suits their needs.

Here are examples.

- One company might put XML pricing information through a program to translate the XML to HTML so that it can post the price lists to its intranet.
- A partner company might put the XML pricing information through a tool to create a marketing presentation.
- Another company might read the XML pricing information into an application for processing.

SOAP Transport Protocol

Client requests and web service responses are transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and web services, all running on different platforms and at various locations on the Internet.

HTTP is a familiar request-and-response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request-and-response model.

The SOAP portion of a transported message does the following:

- Defines an XML-based envelope to describe what is in the message and explain how to process the message
- Includes XML-based encoding rules to express instances of application-defined data types within the message
- Defines an XML-based convention for representing the request to the remote service and the resulting response

WSDL Standard Format

The Web Services Description Language (WSDL) is a standardized XML format for describing network services.

The description includes the name of the service, the location of the service, and ways to communicate with the service.

WSDL service descriptions can be published on the Web.

GlassFish Server provides a tool for generating the WSDL specification of a web service that uses remote procedure calls to communicate with clients.

Java EE Application Assembly and Deployment

A Java EE application is packaged into one or more standard units for deployment to any Java EE platform-compliant system.

Each unit contains

- A functional component or components, such as an enterprise bean, web page, servlet, or applet
- An optional deployment descriptor that describes its content

Once a Java EE unit has been produced, it is ready to be deployed.

Deployment typically involves using a platform's deployment tool to specify location-specific information, such as a list of local users who can access it and the name of the local database.

Once deployed on a local platform, the application is ready to run.

Packaging Applications

A Java EE application is delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file.

A WAR or EAR file is a standard JAR (`.jar`) file with a `.war` or `.ear` extension.

Using JAR, WAR, and EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components.

No extra coding is needed; it is only a matter of assembling (or packaging) various Java EE modules into Java EE JAR, WAR, or EAR files.

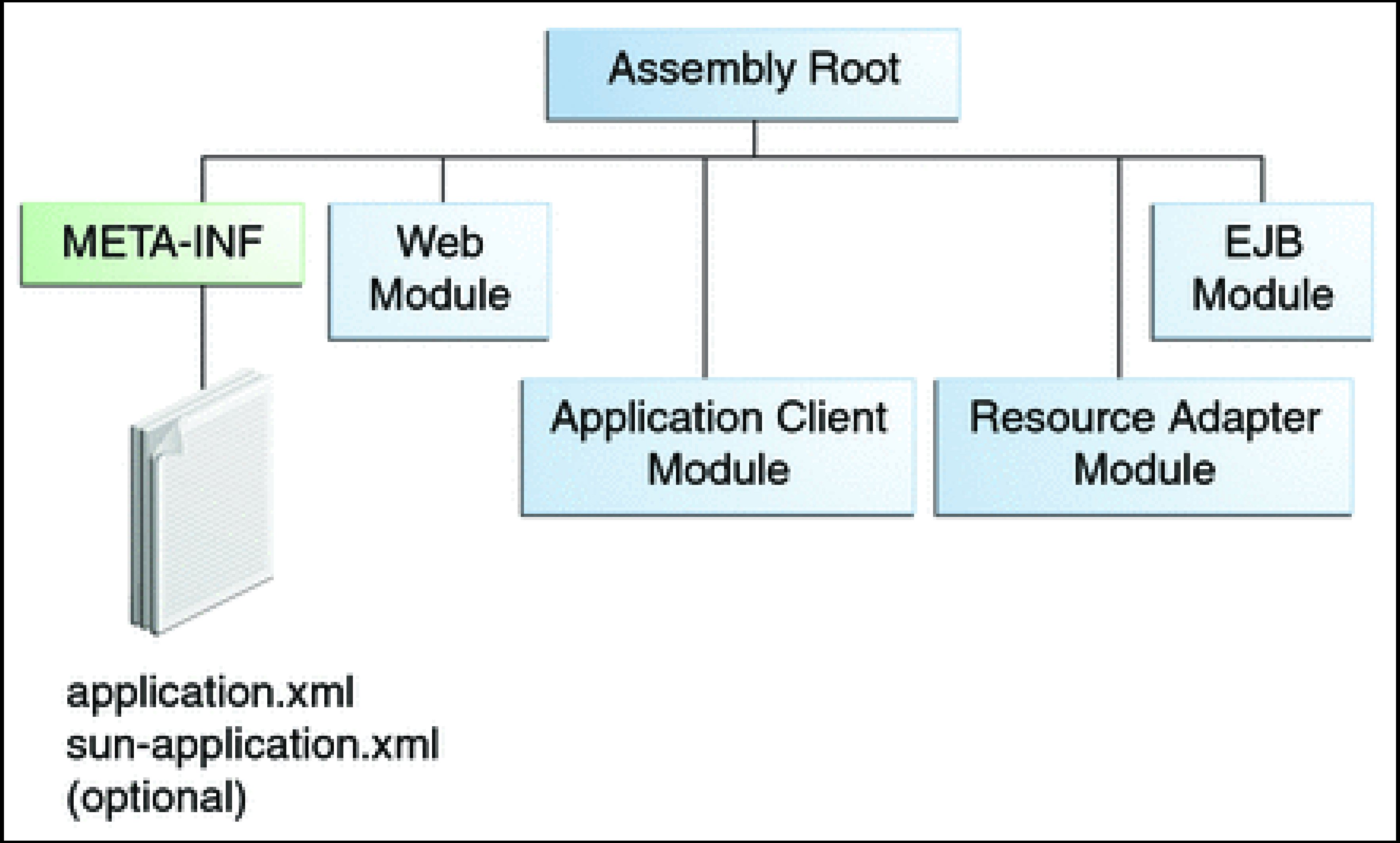
An EAR file (see Figure 1-6) contains Java EE modules and, optionally, deployment descriptors.

A deployment descriptor, an XML document with an `.xml` extension, describes the deployment settings of an application, a module, or a component.

Because deployment descriptor information is declarative, it can be changed without the need to modify the source code.

At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

Figure 1-6 EAR File Structure



The two types of deployment descriptors are Java EE and runtime.

A Java EE deployment descriptor is defined by a Java EE specification and can be used to configure deployment settings on any Java EE-compliant implementation.

A runtime deployment descriptor is used to configure Java EE implementation-specific parameters.

For example, the GlassFish Server runtime deployment descriptor contains such information as the context root of a web application, as well as GlassFish Server implementation-specific parameters, such as caching directives.

The GlassFish Server runtime deployment descriptors are named `sun-`*moduleType*`.xml` and are located in the same `META-INF` directory as the Java EE deployment descriptor.

A Java EE module consists of one or more Java EE components for the same container type and, optionally, one component deployment descriptor of that type.

An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean.

A Java EE module can be deployed as a stand-alone module.

Java EE modules are of the following types:

- EJB modules, which contain class files for enterprise beans and an EJB deployment descriptor.

  EJB modules are packaged as JAR files with a `.jar` extension.

- Web modules, which contain servlet class files, web files, supporting class files, GIF and HTML files, and a web application deployment descriptor.

  Web modules are packaged as JAR files with a `.war` (web archive) extension.

- Application client modules, which contain class files and an application client deployment descriptor.

  Application client modules are packaged as JAR files with a `.jar` extension.

- Resource adapter modules, which contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor.

Together, these implement the Connector architecture (see Java EE Connector Architecture) for a particular EIS.

Resource adapter modules are packaged as JAR files with an `.rar` (resource adapter archive) extension.

Development Roles

Reusable modules make it possible to divide the application development and deployment process into distinct roles so that different people or companies can perform different parts of the process.

The first two roles, Java EE product provider and tool provider, involve purchasing and installing the Java EE product and tools.

After software is purchased and installed, Java EE components can be developed by application component providers, assembled by application assemblers, and deployed by application deployers.

In a large organization, each of these roles might be executed by different individuals or teams.

This division of labor works because each of the earlier roles outputs a portable file that is the input for a subsequent role.

For example, in the application component development phase, an enterprise bean software developer delivers EJB JAR files.

In the application assembly role, another developer may combine these EJB JAR files into a Java EE application and save it in an EAR file.

In the application deployment role, a system administrator at the customer site uses the EAR file to install the Java EE application into a Java EE server.

The different roles are not always executed by different people.

If you work for a small company, for example, or if you are prototyping a sample application, you might perform the tasks in every phase.

Java EE Product Provider

The Java EE product provider is the company that designs and makes available for purchase the Java EE platform APIs and other features defined in the Java EE specification.

Product providers are typically application server vendors that implement the Java EE platform according to the Java EE 6 Platform specification.

Tool Provider

The tool provider is the company or person who creates development, assembly, and packaging tools used by component providers, assemblers, and deployers.

Application Component Provider

The application component provider is the company or person who creates web components, enterprise beans, applets, or application clients for use in Java EE applications.

Enterprise Bean Developer

An enterprise bean developer performs the following tasks to deliver an EJB JAR file that contains one or more enterprise beans:

- Writes and compiles the source code
- Specifies the deployment descriptor (optional)
- Packages the `.class` files and deployment descriptor into the EJB JAR file

Web Component Developer

A web component developer performs the following tasks to deliver a WAR file containing one or more web components:

- Writes and compiles servlet source code
- Writes JavaServer Faces, JSP, and HTML files
- Specifies the deployment descriptor (optional)
- Packages the `.class`, `.jsp`, and `.html` files and deployment descriptor into the WAR file

Application Client Developer

An application client developer performs the following tasks to deliver a JAR file containing the application client:

- **Writes and compiles the source code**
- **Specifies the deployment descriptor for the client (optional)**
- **Packages the `.class` files and deployment descriptor into the JAR file**

Application Assembler

The application assembler is the company or person who receives application modules from component providers and may assemble them into a Java EE application EAR file.

The assembler or deployer can edit the deployment descriptor directly or can use tools that correctly add XML tags according to interactive selections.

A software developer performs the following tasks to deliver an EAR file containing the Java EE application:

- **Assembles EJB JAR and WAR files created in the previous phases into a Java EE application (EAR) file**

- Specifies the deployment descriptor for the Java EE application (optional)
- Verifies that the contents of the EAR file are well formed and comply with the Java EE specification

Application Deployer and Administrator

The application deployer and administrator is the company or person who configures and deploys the Java EE application, administers the computing and networking infrastructure where Java EE applications run, and oversees the runtime environment.

Duties include setting transaction controls and security attributes and specifying connections to databases.

During configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes.

During installation, the deployer moves the application components to the server and generates the container-specific classes and interfaces.

A deployer or system administrator performs the following tasks to install and configure a Java EE application:

- Configures the Java EE application for the operational environment
- Verifies that the contents of the EAR file are well formed and comply with the Java EE specification
- Deploys (installs) the Java EE application EAR file into the Java EE server

Java EE 6 APIs

Figure 1-7 shows the relationships among the Java EE containers.

Figure 1-7 Java EE Containers

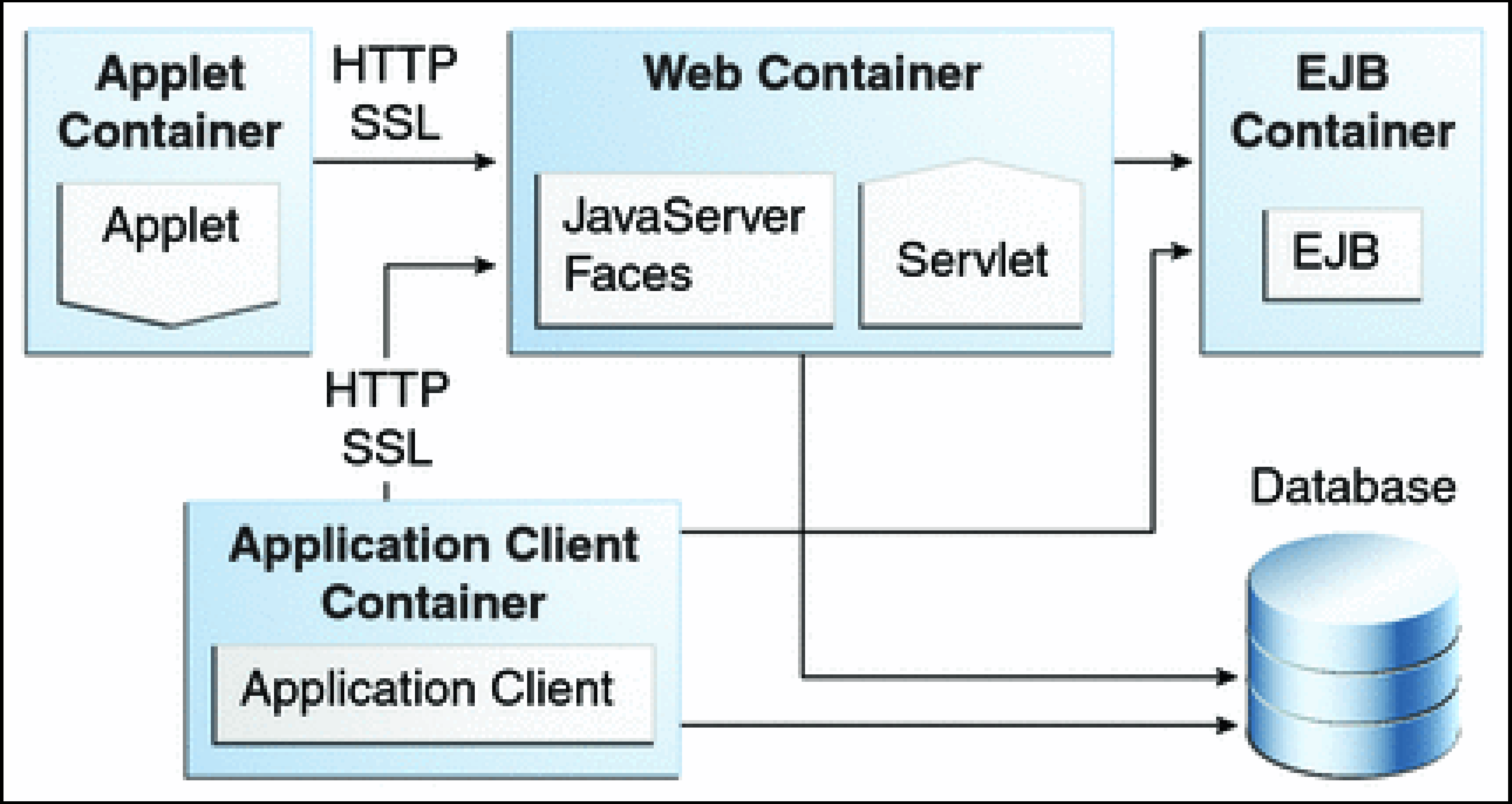


Figure 1-8 shows the availability of the Java EE 6 APIs in the web container.

Figure 1-8 Java EE APIs in the Web Container

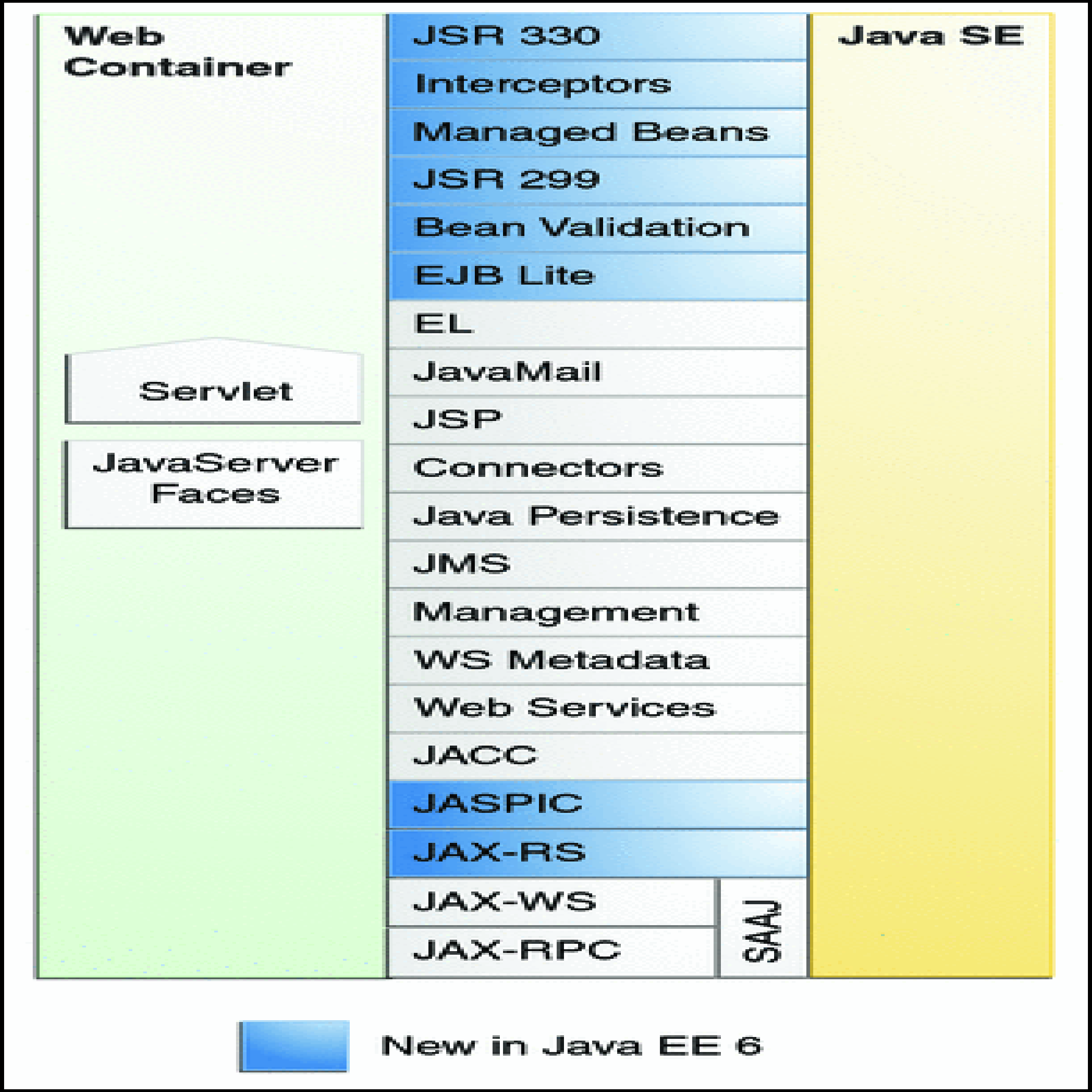


Figure 1-9 shows the availability of the Java EE 6 APIs in the EJB container.

Figure 1-9 Java EE APIs in the EJB Container

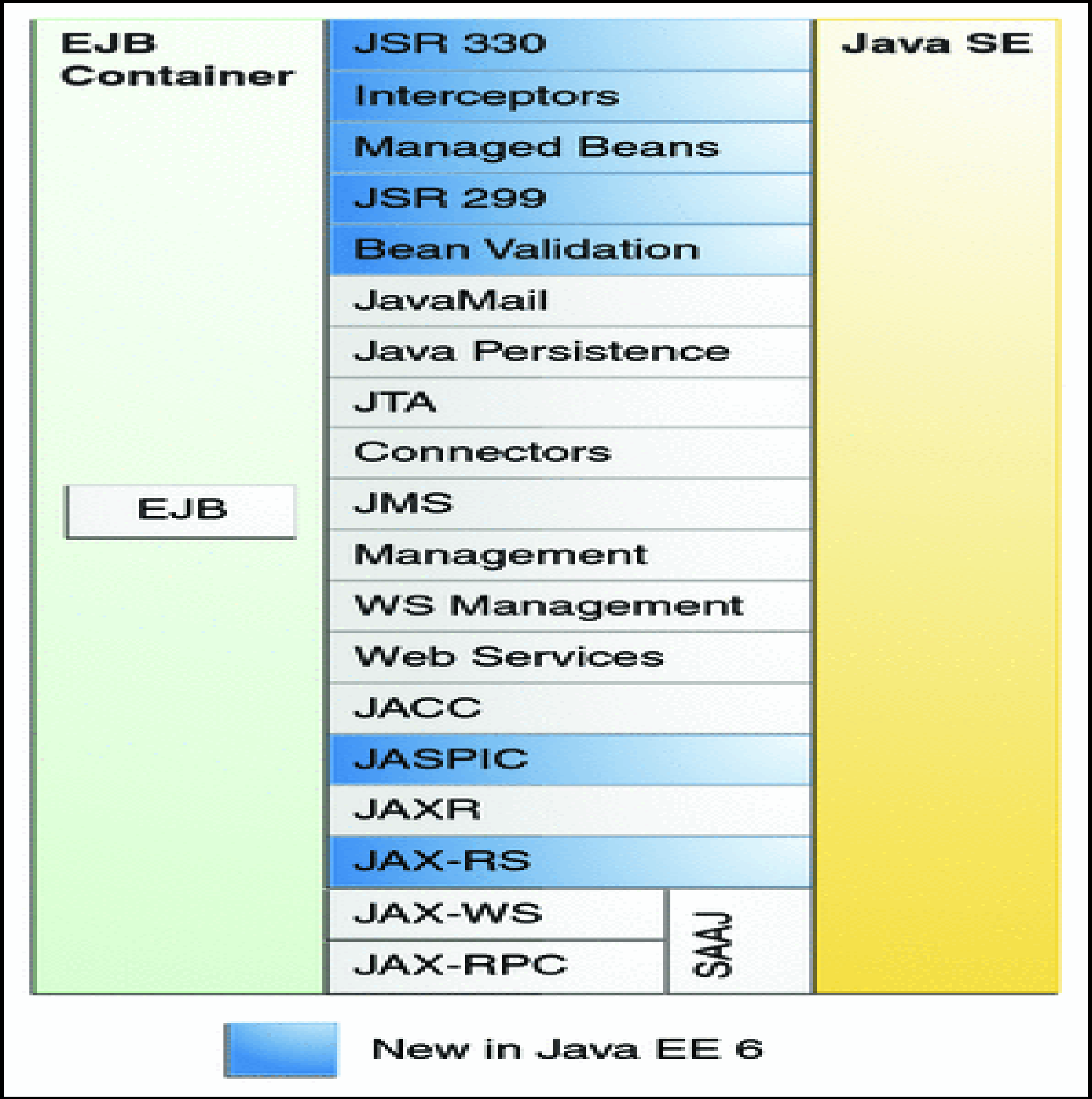
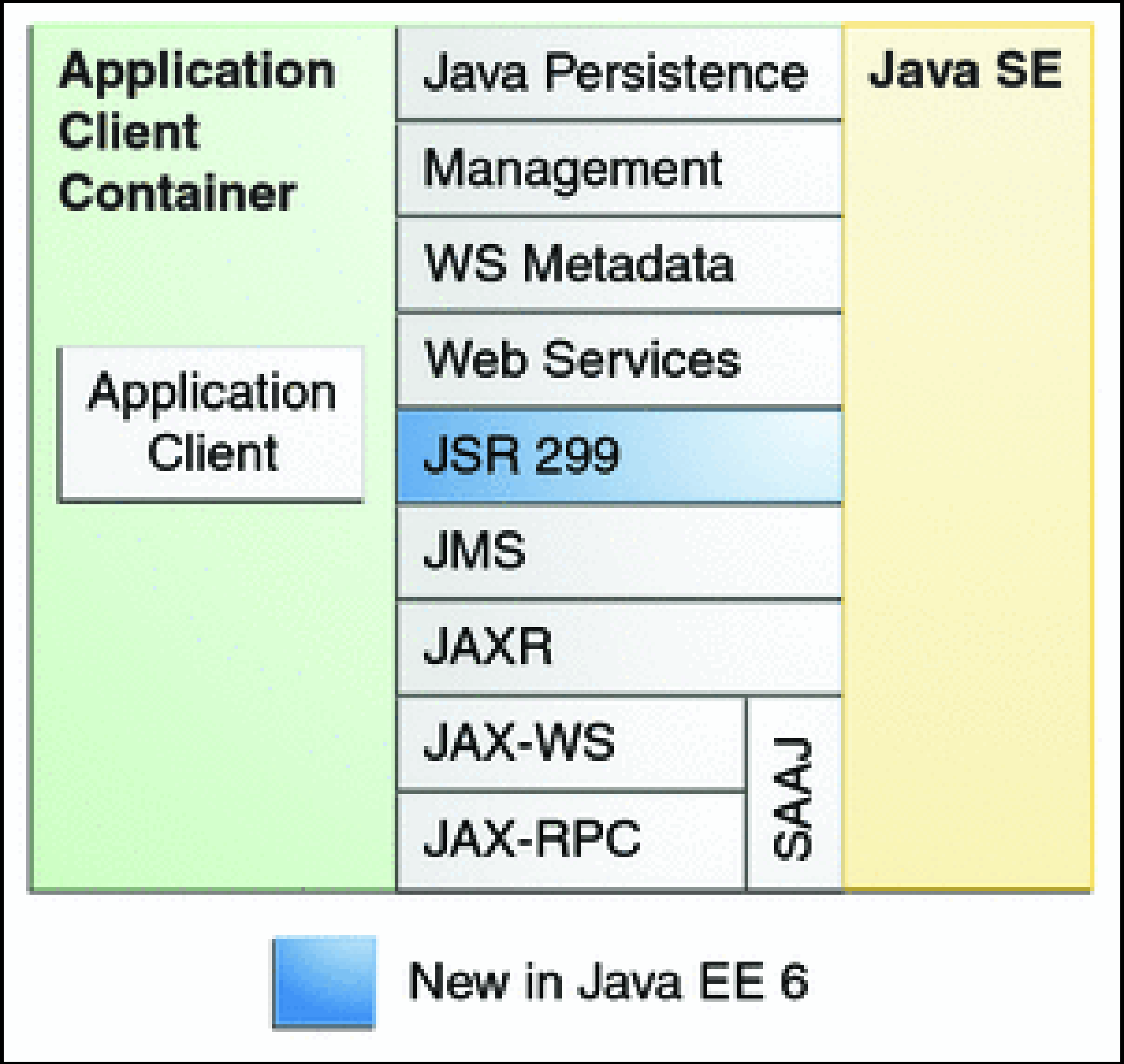


Figure 1-10 shows the availability of the Java EE 6 APIs in the application client container.

Figure 1-10 Java EE APIs in the Application Client Container



The following sections give a brief summary of the technologies required by the Java EE platform and the APIs used in Java EE applications.

Enterprise JavaBeans Technology

An Enterprise JavaBeans (EJB) component, or enterprise bean, is a body of code having fields and methods to implement modules of business logic.

You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server.

Enterprise beans are either session beans or message-driven beans.

- A session bean represents a transient conversation with a client.

When the client finishes executing, the session bean and its data are gone.

- A message-driven bean combines features of a session bean and a message listener, allowing a business component to receive messages asynchronously.

  Commonly, these are Java Message Service (JMS) messages.

In the Java EE 6 platform, new enterprise bean features include the following:

- The ability to package local enterprise beans in a WAR file
- Singleton session beans, which provide easy access to shared state
- A lightweight subset of Enterprise JavaBeans functionality (EJB Lite) that can be provided within Java EE Profiles, such as the Java EE Web Profile.

The Interceptors specification, which is part of the EJB 3.1 specification, makes more generally available the interceptor facility originally defined as part of the EJB 3.0 specification.

Java Servlet Technology

Java Servlet technology lets you define HTTP-specific servlet classes.

A servlet class extends the capabilities of servers that host applications accessed by way of a request-response programming model.

Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

In the Java EE 6 platform, new Java Servlet technology features include the following:

- Annotation support
- Asynchronous support
- Ease of configuration
- Enhancements to existing APIs
- Pluggability

JavaServer Faces Technology

JavaServer Faces technology is a user interface framework for building web applications.

The main components of JavaServer Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies.

- A `Renderer` object generates the markup to render the component and converts the data stored in a model object to types that can be represented in a view.

- A standard `RenderKit` for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
- Data conversion between model objects and components
- Managed model object creation
- Page navigation configuration
- Expression Language (EL)

All this functionality is available using standard Java APIs and XML-based configuration files.

In the Java EE 6 platform, new features of JavaServer Faces include the following:

- The ability to use annotations instead of a configuration file to specify managed beans
- Facelets, a display technology that replaces JavaServer Pages (JSP) technology using XHTML files
- Ajax support
- Composite components
- Implicit navigation

JavaServer Pages Technology

JavaServer Pages (JSP) technology lets you put snippets of servlet code directly into a text-based document.

A JSP page is a text-based document that contains two types of text:

- Static data, which can be expressed in any text-based format such as HTML or XML
- JSP elements, which determine how the page constructs dynamic content

JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications.

Instead of mixing tags from numerous vendors in your JSP applications, you use a single, standard set of tags.

This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

Java Persistence API

The Java Persistence API is a Java standards-based solution for persistence.

Persistence uses an object/relational mapping approach to bridge the gap between an object-oriented model and a relational database.

The Java Persistence API can also be used in Java SE applications, outside of the Java EE environment.

Java Persistence consists of the following areas:

- The Java Persistence API
- The query language
- Object/relational mapping metadata
-

Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions.

The Java EE architecture provides a default auto commit to handle transaction commits and rollbacks.

An auto commit means that any other applications that are viewing data will see the updated data after each database read or write operation.

However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

Java API for RESTful Web Services

The Java API for RESTful Web Services (JAX-RS) defines APIs for the development of web services built according to the Representational State Transfer (REST) architectural style.

A JAX-RS application is a web application that consists of classes that are packaged as a servlet in a WAR file along with required libraries.

The JAX-RS API is new to the Java EE 6 platform.

Managed Beans

Managed Beans, lightweight container-managed objects (POJOs) with minimal requirements, support a small set of basic services, such as resource injection, lifecycle callbacks, and interceptors.

Managed Beans represent a generalization of the managed beans specified by JavaServer Faces technology and can be used anywhere in a Java EE application, not just in web modules.

The Managed Beans specification is part of the Java EE 6 platform specification (JSR 316).

Managed Beans are new to the Java EE 6 platform.

Contexts and Dependency Injection for the Java EE Platform (JSR 299)

Contexts and Dependency Injection (CDI) for the Java EE platform defines a set of contextual services, provided by Java EE containers, that make it easy for developers to use enterprise beans along with JavaServer Faces technology in web applications.

Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate different kinds of components in a loosely coupled but type-safe way.

CDI is new to the Java EE 6 platform.

Dependency Injection for Java (JSR 330)

Dependency Injection for Java defines a standard set of annotations (and one interface) for use on injectable classes.

In the Java EE platform, CDI provides support for Dependency Injection.

Specifically, you can use DI injection points only in a CDI-enabled application.

Dependency Injection for Java is new to the Java EE 6 platform.

Bean Validation

The Bean Validation specification defines a metadata model and API for validating data in JavaBeans components.

Instead of distributing validation of data over several layers, such as the browser and the server side, you can define the validation constraints in one place and share them across the different layers.

Bean Validation is new to the Java EE 6 platform.

Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows Java EE application components to create, send, receive, and read messages.

It enables distributed communication that is loosely coupled, reliable, and asynchronous.

Java EE Connector Architecture

The Java EE Connector architecture is used by tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any Java EE product.

A resource adapter is a software component that allows Java EE application components to access and interact with the underlying resource manager of the EIS.

Because a resource adapter is specific to its resource manager, a different resource adapter typically exists for each type of database or enterprise information system.

The Java EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of Java EE based web services with existing EISs that can be either synchronous or asynchronous.

Existing applications and EISs integrated through the Java EE Connector architecture into the Java EE platform can be exposed as

XML-based web services by using JAX-WS and Java EE component models.

Thus JAX-WS and the Java EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

JavaMail API

Java EE applications use the JavaMail API to send email notifications.

The JavaMail API has two parts:

- An application-level interface used by the application components to send mail
- A service provider interface

The Java EE platform includes the JavaMail API with a service provider that allows application components to send Internet mail.

Java Authorization Contract for Containers

The Java Authorization Contract for Containers (JACC) specification defines a contract between a Java EE application server and an authorization policy provider.

All Java EE containers support this contract.

The JACC specification defines `java.security.Permission` classes that satisfy the Java EE authorization model.

The specification defines the binding of container access decisions to operations on instances of these permission classes.

It defines the semantics of policy providers that use the new permission classes to address the authorization requirements of the Java EE platform, including the definition and use of roles.

Java Authentication Service Provider Interface for Containers

The Java Authentication Service Provider Interface for Containers (JASPIC) specification defines a service provider interface (SPI) by which authentication providers that implement message authentication mechanisms may be integrated in client or server message-processing containers or runtimes.

Authentication providers integrated through this interface operate on network messages provided to them by their calling container.

The authentication providers transform outgoing messages so that the source of the message can be authenticated by the receiving container, and the recipient of the message can be authenticated by the message sender.

Authentication providers authenticate incoming messages and return to their calling container the identity established as a result of the message authentication.

JASPIC is new to the Java EE 6 platform.

Java EE 6 APIs in the Java Platform, Standard Edition 6.0

Several APIs that are required by the Java EE 6 platform are included in the Java Platform, Standard Edition 6.0 (Java SE 6) platform and are thus available to Java EE applications.

Java Database Connectivity API

The Java Database Connectivity (JDBC) API lets you invoke SQL commands from Java programming language methods.

You use the JDBC API in an enterprise bean when you have a session bean access the database.

You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts:

- An application-level interface used by the application components to access a database
- A service provider interface to attach a JDBC driver to the Java EE platform

Java Naming and Directory Interface API

The Java Naming and Directory Interface (JNDI) API provides naming and directory functionality, enabling applications to access multiple naming and directory services, including existing naming and directory services, such as LDAP, NDS, DNS, and NIS.

The JNDI API provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes.

Using JNDI, a Java EE application can store and retrieve any type of named Java object, allowing Java EE applications to coexist with many legacy applications and systems.

Java EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment.

A naming environment allows a component to be customized without the need to access or change the component's source code.

A container implements the component's environment and provides it to the component as a JNDI naming context.

A Java EE component can locate its environment naming context by using JNDI interfaces.

A component can create a `javax.naming.InitialContext` object and look up the environment naming context in `InitialContext` under the name `java:comp/env`.

A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A Java EE component can access named system-provided and user-defined objects.

The names of system-provided objects, such as JTA `UserTransaction` objects, are stored in the environment naming context `java:comp/env`.

The Java EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC `DataSource` objects, and message connections.

An object should be named within a subcontext of the naming environment according to the type of the object.

For example, enterprise beans are named within the subcontext `java:comp/env/ejb`, and JDBC `DataSource` references are named within the subcontext `java:comp/env/jdbc`.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is used by the JavaMail API.

JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP), part of the Java SE platform, supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT).

JAXP enables applications to parse and transform XML documents independently of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts.

Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the Worldwide Web Consortium (W3C) schema.

You can find information on the W3C schema at this URL: http://www.w3.org/XML/Schema.

Java Architecture for XML Binding

The Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in Java language programs.

JAXB can be used independently or in combination with JAX-WS, where it provides a standard data binding for web service messages.

All Java EE application client containers, web containers, and EJB containers support the JAXB API.

SOAP with Attachments API for Java

The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-WS depends.

SAAJ enables the production and consumption of messages that conform to the SOAP 1.1 and 1.2 specifications and SOAP with Attachments note.

Most developers do not use the SAAJ API, instead using the higher-level JAX-WS API.

Java API for XML Web Services

The Java API for XML Web Services (JAX-WS) specification provides support for web services that use the JAXB API for binding XML data to Java objects.

The JAX-WS specification defines client APIs for accessing web services as well as techniques for implementing web service endpoints.

The Implementing Enterprise Web Services specification describes the deployment of JAX-WS-based services and clients.

The EJB and Java Servlet specifications also describe aspects of such deployment.

It must be possible to deploy JAX-WS-based applications using any of these deployment models.

The JAX-WS specification describes the support for message handlers that can process message requests and responses.

In general, these message handlers execute in the same container and with the same privileges and execution context as the JAX-WS client or endpoint component with which they are associated.

These message handlers have access to the same JNDI `java:comp/env` namespace as their associated component.

Custom serializers and deserializers, if supported, are treated in the same way as message handlers.

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) provides a way for a Java EE application to authenticate and authorize a specific user or group of users to run it.

JAAS is a Java programming language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java Platform security architecture to support user-based authorization.

GlassFish Server Tools

The GlassFish Server is a compliant implementation of the Java EE 6 platform.

In addition to supporting all the APIs described in the previous sections, the GlassFish Server includes a number of Java EE tools that are not part of the Java EE 6 platform but are provided as a convenience to the developer.

This section briefly summarizes the tools that make up the GlassFish Server.

Instructions for starting and stopping the GlassFish Server, starting the Administration Console, and starting and stopping the Java DB server are in Chapter 2, Using the Tutorial Examples.

The GlassFish Server contains the tools listed in Table 1-1.

Basic usage information for many of the tools appears throughout the tutorial.

For detailed information, see the online help in the GUI tools.

Table 1-1 GlassFish Server Tools

Tool	Description
Administration Console	A web-based GUI GlassFish Server administration utility. Used to stop the GlassFish Server and manage users, resources, and applications.
`asadmin`	A command-line GlassFish Server administration utility. Used to start and stop the GlassFish Server and manage users, resources, and applications.

`appclient`	A command-line tool that launches the application client container and invokes the client application packaged in the application client JAR file.
`capture-schema`	A command-line tool to extract schema information from a database, producing a schema file that the GlassFish Server can use for container-managed persistence.
`package-appclient`	A command-line tool to package the application client container libraries and JAR files.
Java DB database	A copy of the Java DB server.
`xjc`	A command-line tool to transform, or bind, a source XML schema to a set of JAXB content classes in the Java programming language.

`schemagen`	A command-line tool to create a schema file for each namespace referenced in your Java classes.
`wsimport`	A command-line tool to generate JAX-WS portable artifacts for a given WSDL file. After generation, these artifacts can be packaged in a WAR file with the WSDL and schema documents, along with the endpoint implementation, and then deployed.
`wsgen`	A command-line tool to read a web service endpoint class and generate all the required JAX-WS portable artifacts for web service deployment and invocation.