Beginning Java EE 5

From Novice to Professional

■ ■ ■

Kevin Mukhar and Chris Zelenak
with James L. Weaver and Jim Crume

Apress®

Beginning Java EE 5: From Novice to Professional

Copyright © 2006 by Kevin Mukhar and Chris Zelenak, with James L. Weaver and Jim Crume

ISBN (pbk): 1-59059-470-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Dilip Thomas
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editors: Marilyn Smith, Ami Knox, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert Stevens
Proofreader: Elizabeth Berry
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 1

Java EE Essentials

The word *enterprise* has magical powers in computer programming circles. It can increase the price of a product by an order of magnitude and double the potential salary of an experienced consultant. Your application may be free of bugs, and cleanly coded using all the latest techniques and tools, but is it enterprise-ready? What exactly is the magic ingredient that makes enterprise development qualitatively different from run-of-the-mill development?

Enterprise applications solve business problems. This usually involves the safe storage, retrieval, and manipulation of business data: customer invoices, mortgage applications, flight bookings, and so on. They might have multiple user interfaces: a web interface for consumers and a graphical user interface (GUI) application running on computers in the branch offices, for example. Enterprise applications must deal with communication between remote systems, coordinate data in multiple stores, and ensure the system always follows the rules laid down by the business. If any part of the system crashes, the business loses part of its ability to function and starts to lose money. If the business grows, the application needs to grow with it. All this adds up to what characterizes enterprise applications: robustness in the face of complexity.

When we set out to build a GUI application, we don't start by working out how to draw pixels on the screen and build our own code to track the user's mouse around the screen; we rely on a GUI library, like Swing, to do that for us. Similarly, when we set out to create the components of a full-scale enterprise solution, we would be crazy to start from scratch.

Enterprise programmers build their applications on top of systems called *application servers*. Just as GUI toolkits provide services of use to GUI applications, application servers provide services of use to enterprise applications—things like communication facilities to talk to other computers, management of database connections, the ability to serve web pages, and management of transactions.

Just as Java provides a uniform way to program GUI applications on any underlying operating system, Java also provides a uniform way to program enterprise applications on any underlying application server. The set of libraries developed by Sun Microsystems and the Java Community Process that represent this uniform application server application programming interface (API) is what we call the Java Platform, Enterprise Edition 5 (Java EE 5), and it is the subject of this book.

This chapter provides a high-level introduction to Java EE. In this chapter, you will learn:

- Why you would want to use Java EE
- What the benefits of a multitier application architecture are
- How Java EE provides vendor independence and scalability

- What the main Java EE features and concepts are
- How to use common Java EE architectures

So, without further ado, let's get started!

What Is Java EE?

Since you're reading this book, you obviously have some interest in Java EE, and you probably have some notion of what you're getting into. For many fledgling Java EE developers, Java EE equates to Enterprise JavaBeans (EJBs). However, Java EE is a great deal more than just EJBs.

While perhaps an oversimplification, Java EE is a suite of specifications for APIs, a distributed computing architecture, and definitions for packaging of distributable components for deployment. It's a collection of standardized components, containers, and services for creating and deploying distributed applications within a well-defined distributed computing architecture. Sun's Java web site says, " Java Platform, Enterprise Edition 5 (Java EE 5) defines the standard for developing component-based multitier enterprise applications."

As its name implies, Java EE is targeted at large-scale business systems. Software that functions at this level doesn't run on a single PC—it requires significantly more computing power and throughput than that. For this reason, the software needs to be partitioned into functional pieces and deployed on the appropriate hardware platforms. That is the essence of distributed computing. Java EE provides a collection of standardized components that facilitate software deployment, standard interfaces that define how the various software modules interconnect, and standard services that define how the different software modules communicate.

How Java EE Relates to J2SE

Java EE isn't a replacement for the Java 2 Standard Edition (J2SE). J2SE provides the essential language framework on which Java EE builds. It is the core on which Java EE is based. As you'll see, Java EE consists of several layers, and J2SE is right at the base of that pyramid for each component of Java EE.

As a Java developer, you've probably already learned how to build user interfaces with the Swing or Abstract Window Toolkit (AWT) components. You'll still be using those to build the user interfaces for your Java EE applications, as well as HTML-based user interfaces. Since J2SE is at the core of Java EE, everything that you've learned so far remains useful and relevant.

In addition, Java EE provides another API for creating user interfaces. This API is named JavaServer Faces (JSF) and is one of the newest Java EE technologies. You'll also see that the Java EE platform provides the most significant benefit in developing the middle-tier portion of your application—that's the business logic and the connections to back-end data sources. You'll use familiar J2SE components and APIs in conjunction with the Java EE components and APIs to build that part of your applications.

Why Java EE?

Java EE defines a number of services that, to someone developing enterprise-class applications, are as essential as electricity and running water. Life is simple when you simply turn the faucet and water starts running, or flip the switch and lights come on. If you have ever been involved with building a house, you know that there is a great deal of effort, time, and expense in building the infrastructure of plumbing and wiring, which is then so nicely hidden behind freshly painted walls. At the points where that infrastructure is exposed, there are standard interfaces for controlling (water faucets and light switches, for example) and connecting (power sockets, lamp sockets, and hose bibs, for example) to the infrastructure.

Suppose, though, that the wiring and plumbing in your home wasn't already there. You would need to put in your own plumbing and electricity. Without standard components and interfaces, you would need to fabricate your own pipes, wiring, and so on. It would be terrifically expensive and an awful lot of work.

Similarly, there is a great deal of infrastructure required to write enterprise-class applications. There are a bunch of different system-level capabilities that you need in order to write distributed applications that are scalable, robust, secure, and maintainable. Some vital pieces of that infrastructure include security, database access, and transaction control. Security ensures that users are who they claim to be and can access only the parts of the application that they're entitled to access. Database access is also a fundamental component so that your application can store and retrieve data. Transaction support is required to make sure that the right data is updated at the right time. If you're not familiar with some of these concepts, don't worry—you'll be introduced to them one at a time throughout this book.

Putting in a distributed computing infrastructure—the plumbing and wiring of an architecture that supports enterprise applications—is no simple feat. That's why Java EE-based architectures are so compelling; the hard system-level infrastructure is already in place.

But why not custom build (or pay someone to custom build) an infrastructure that is designed around your particular application? Well, for starters, it would take a fantastic amount of time, money, and effort. And even if you were to build up that infrastructure, it would be different from anyone else's infrastructure, so you wouldn't be able to share components or interoperate with anyone else's distributed computing model. That's a lot of work for something that sounds like a dead end. And if you were lucky enough to find a vendor that could sell you a software infrastructure, you would need to worry about being locked into that single vendor's implementation, and not being able to switch vendors at some point in the future.

The good news is, no surprise, that Java EE defines a set of containers, connectors, and components that fill that gap. Java EE not only fills the gap, but it's based on well-known, published specifications. That means that applications written for Java EE will run on any number of Java EE-compliant implementations. The reference implementation supplied with the Java EE Software Development Kit from Sun (Java EE SDK) provides a working model that we'll use throughout this book, since it's the implementation that Sun has built from the specification and is freely available. In the next chapter, you'll get an introduction to installing and testing the Java EE SDK.

Multitier Architecture

One of the recurring themes that you'll run into with Java EE is the notion of supporting applications that are partitioned into several levels, or *tiers*. That is an architectural cornerstone of Java EE and merits a little explanation. If you are already familiar with *n*-tier application architectures, feel free to skip ahead. Otherwise, the overview presented here will be a good introduction or review that will help lay the foundation for understanding the rationale behind much of Java EE's design and the services it provides.

If you think about a software application composition, you can break it down into three fundamental concerns, or logical layers:

- The first area of concern is displaying stuff to the user and collecting data from the user. That user interface layer is often called the *presentation layer*, since its job is to present stuff to the user and provide a means for the user to present stuff to the software system. The presentation layer includes the part of the software that creates and controls the user interface and validates the user's actions.
- Underlying the presentation layer is the logic that makes the application work and handles the important processing. The process in a payroll application to multiply the hours worked by the salary to determine how much to pay someone is one example of this kind of logic. This logical layer is called the *business rules layer*, or more informally the *middle tier*.
- All nontrivial business applications need to read and store data, and the part of the software that is responsible for reading and writing data—from whatever source that might be—forms the *data access layer*.

Single-Tier Systems

Simple software applications are written to run on a single computer, as illustrated in Figure 1-1. All of the services provided by the application—the user interface, the persistent data access, and the logic that processes the data input by the user and reads from storage—all exist on the same physical machine and are often lumped together into the application. That monolithic architecture is called *single tier*, because all of the logical application services—the presentation, the business rules, and the data access layers—exist in a single computing layer.

Single-tier systems are relatively easy to manage, and data consistency is simple because data is stored in only one single location. However, they also have some disadvantages. Single-tier systems do not scale to handle multiple users, and they do not provide an easy means of sharing data across an enterprise. Think of the word processor on your personal computer: It does an excellent job of helping you to create documents, but the application can be used by only a single person. Also, while you can share documents with other people, only one person can work on the document at a time.

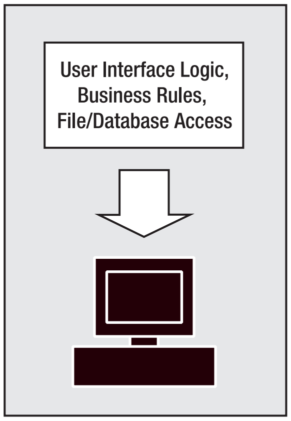

Figure 1-1. *In the traditional computer application, all of the functionality of the application exists on the user's computer.*

Client/Server (Two-Tier) Architecture

More significant applications may take advantage of a database server and access persistent data by sending SQL commands to a database server to save and retrieve data. In this case, the database runs as a separate process from the application, or even on a different machine than the machine that runs the rest of the program. As illustrated in Figure 1-2, the components for data access are segregated from the rest of the application logic. The rationale for this approach is to centralize data to allow multiple users to simultaneously work with a common database, and to provide the ability for a central database server to share some of the load associated with running the application. This architecture is usually referred to as *client/server* and includes any architecture where a client communicates with a server, whether that server provides data access or some other service.

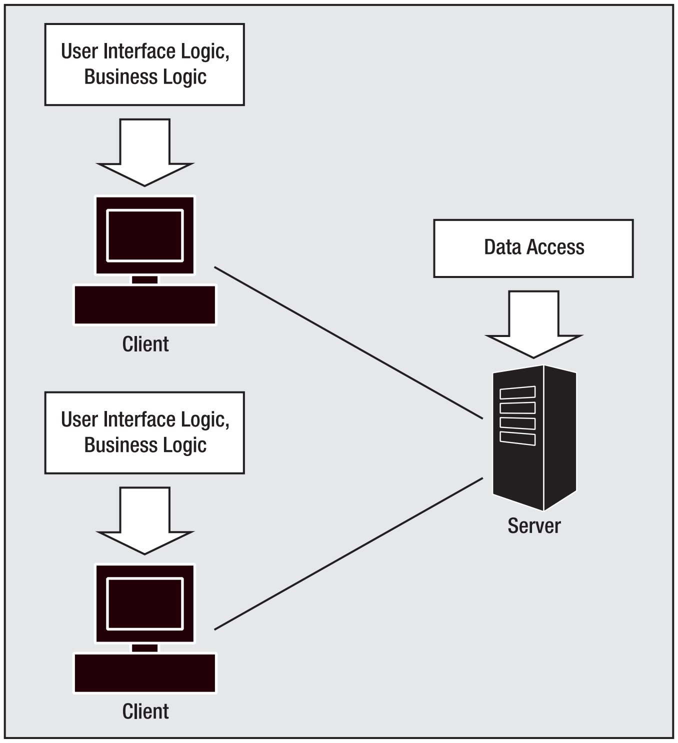
User Interface Logic, Business Logic
Client
User Interface Logic, Business Logic
Client
Data Access
Server

Figure 1-2. *In a client/server architecture, an application client accesses services from another process to do its job.*

It's convenient and more meaningful to conceptualize the division of the responsibility into layers, or tiers. Figure 1-3 shows the client/server software architecture in two tiers.

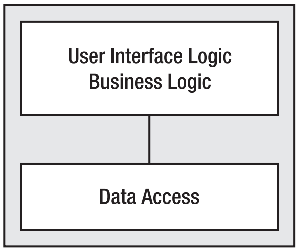

Figure 1-3. *The client/server architecture shown in a layer, or tier, diagram*

One of the disadvantages of two-tier architecture is that the logic that manipulates the data and applies specific application rules concerning the data is lumped into the application itself. This poses a problem when multiple applications use a shared database. Consider, for example, a database that contains customer information that is used for order fulfillment, invoicing, promotions, and general customer resource management. Each one of those applications would need to be built with all of the logic and rules to manipulate and access customer data. For example, there might be a standard policy within a company that any customer whose account is more than 90 days overdue will be subject to a credit hold. It seems simple enough to build that rule into every application that's accessing customer data, but when the policy changes to reflect a credit hold at 60 days, updating each application becomes a real mess.

You might be tempted to try to solve this problem by building a reusable library that encapsulates the business rules. When the rules change, you can just replace that library, rebuild the application, and redistribute it to the computers running the application. There are some fundamental problems with that strategy, however. First, that strategy assumes that all of the applications have been created using the same programming language, run on the same platform, or at least have some strategy for gluing the library to the application. Next, the applications may need to be recompiled or reassembled with the new library. Moreover, even if the library is a drop-in replacement without requiring recompiling, it's still going to be a royal pain to make sure that each installation of the application has the right library installed simultaneously (it wouldn't do to have conflicting business rules being enforced by different applications at the same time).

In order to get out of that mess, the logical thing to do is to physically separate those business rules out from the computers running the applications onto a separate server so that the software that runs the business rules needs to be updated only once, not for each computer that runs the application.

N-Tier Architecture

Figure 1-4 shows a third tier added to the two-tier client/server model. In this model, all of the business logic is extracted out of the application running at the desktop. The application at the desktop is responsible for presenting the user interface to the end user and for communicating to the business logic tier. It is no longer responsible for enforcing business rules or accessing databases. Its job is solely as the presentation layer.

Note Bear in mind that at this point we're talking somewhat abstractly and theoretically. In a perfect world, without performance and other implications, the division of responsibility in an application would be very clear-cut. You'll see throughout this book that you must make practical, balanced implementation decisions about how responsibilities are partitioned in order to create an application that is flexible and performs well.

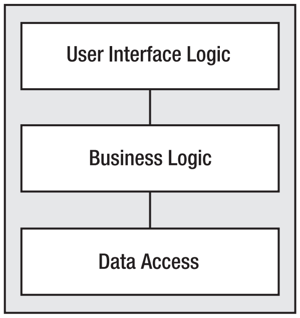

Figure 1-4. *A common enterprise architecture consists of three tiers: presentation, business, and data.*

Typically, in a deployed application, the business logic tier executes on a server apart from the workstation (you'll see shortly that this isn't absolutely required, though). The business logic tier provides the logical glue to bind the presentation to the database. Since it's running on a server, it's accessible to any number of users on the network running applications that take advantage of its business rules. As the number of users demanding those services increases, and the business logic becomes increasingly complex and processor-intensive, the server can be scaled up or more servers can be added. Scaling a single server is a lot easier and cheaper than upgrading everyone's workstations.

One of the really great things that this architecture makes possible is the ability to start to build application models where the classes defined in the business logic tier are taken directly from the application domain. The code in the business logic layer can work with classes that model things in the real world (like a `Customers` class) rather than working with complex SQL statements. By pushing implementation details into the appropriate layer, and designing applications that work with classes modeled from the real world, applications become much easier to understand and extend.

It's possible to continue the process of partitioning the application functionality into increasingly thin functional layers, as illustrated in Figure 1-5. There are some very effective application architectures based on *n*-tier architecture. The application architect is free to partition the application into as many layers as appropriate, based on the capabilities of the computing and network hardware on which the system is deployed. However, you do need to be careful about reaching a point of diminishing returns, since the performance penalty for the network communication between the layers can start to outweigh any gains in performance.

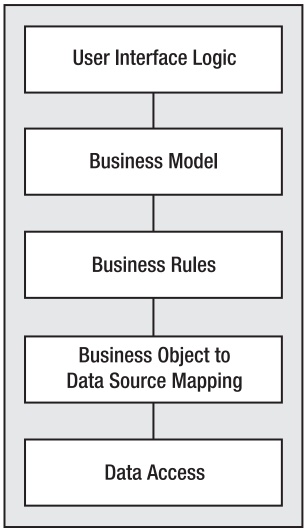

Figure 1-5. *An enterprise application is not limited to two or three tiers. The software architect can design the system to consist of any number of layers, depending on the system requirements and deployment configuration.*

In summary, *n*-tier application architecture is intended to address a number of problems, including the following:

- The high cost of maintenance when business rules change. *N*-tier applications have improved maintainability.
- Inconsistent business rule implementation between applications. *N*-tier applications provide consistency.
- Inability to share data or business rules between applications. *N*-tier applications offer interoperability.
- Inability to provide web-based front ends to line-of-business applications. *N*-tier applications are flexible.
- Poor performance and inability to scale applications to meet increased user load. *N*-tier applications are scalable.
- Inadequate or inconsistent security across applications. *N*-tier applications can be designed to be secure.

The Java EE architecture is based on the notion of *n*-tier applications. Java EE makes it very easy to build industrial-strength applications based on two, three, or more application layers, and provides all of the plumbing and wiring to make that possible.

Note that *n*-tier architecture does not demand that each of the application layers run on a separate machine. It's certainly possible to write *n*-tier applications that execute on a standalone machine, as you'll see. The merit of the application design is that the layers can be split apart and deployed on separate machines, as the application requires.

Note Labeling a particular architecture as *three-tier*, *five-tier*, and so on is almost guaranteed to spur some academic debate. Some insist that tiers are defined by the physical partitioning, so if the application components reside on client workstations, an application server, and a database server machine, it's definitively a three-tier application. Others will classify applications by the logical partitioning where the potential exists for physical partitioning. For the discussions in this chapter, we'll take the latter approach, with apologies in advance for those who subscribe to the former.

Vendor Independence

Sun Microsystems—the company that created the Java platform and plays a central role in Java technologies, including the Java EE specification—has promoted the Java platform as a solid strategy for building applications that aren't locked into a single platform. In the same way, the architects of Java EE have created it as an open specification that can be implemented by anyone. To date, there are scores of Java EE-based application servers that provide a platform for building and deploying scalable *n*-tier applications. Any application server that bills itself as Java EE-compliant must provide the same suite of services using the interfaces and specifications that Sun has made part of Java EE.

This provides the application developer with a number of choices when implementing a project, and similar choices down the road as more applications are added to an organization's suite of solutions. Building an application atop the Java EE architecture provides substantial decoupling between the application logic that you write and the other stuff—security, database access, transaction support, and so on—provided by the Java EE server.

Remember that all Java EE servers must support the same interfaces defined in the Java EE specification. That means you can design your application on one server implementation and deploy it on a different one. You can decide later that you want to change which Java EE server you use in your production environment. Moving your application over to the new production environment can be almost trivial.

Platform independence is something that you can take advantage of in your development. For example, you may be away from the office quite a bit, and use your notebook computer running Windows to do development. It's pretty easy to use that configuration to build, test, and debug (Java EE has great support for pool-side computing). When you're back in the office and happy with a particular component, you can deploy it to, say, Linux-based servers with little effort, despite the fact that those servers are running a different operating system and different Java EE implementation (after testing, of course!).

Bear in mind that each Java EE vendor provides some added value to its particular Java EE implementation. After all, if there weren't market differentiators, there would be no competition. The Java EE specification covers a lot, but there is also a lot that is not specified in Java EE. Performance, reliability, and scalability are just a few of the areas that aren't part of the Java EE specification but are areas where vendors have focused a great deal of time and attention. That added value may be ease of use in its deployment tools, highly optimized performance, support for server clustering (which makes a group of servers able to serve application clients as if it were a single super-fast, super-big server), and so on. The key point here is to keep two issues in mind:

- Your production applications can potentially benefit from capabilities not supported in the Sun Java EE reference implementation. Just because your application's performance stinks on the reference implementation running on your laptop doesn't mean that Java EE is inherently slow.
- Any vendor-specific capabilities that you take advantage of in your production applications may impact the vendor independence of your application.

Scalability

Defining throughput and performance requirements is a vital step in requirements definition. Even the best of us get caught off-guard sometimes, though. Things can happen down the road—an unanticipated number of users using a system at the same time, increased loading on hardware, unsatisfactory availability in the event of server failure, and so on—that can throw a monkey wrench into the works.

The Java EE architecture provides a lot of flexibility to accommodate changes as the requirements for throughput, performance, and capacity change. The *n*-tier application architecture allows software developers to apply additional computing power where it's needed. Partitioning applications into tiers also enables refactoring of specific pain points without impacting adjacent application components.

Clustering, connection pooling, and failover will become familiar terms to you as you build Java EE applications. Several providers of Java EE application servers have worked diligently to come up with innovative ways to improve application performance, throughput, and availability—each with its own special approach within the Java EE framework.

Features and Concepts in Java EE

Getting your arms around the whole of Java EE will take some time, study, and patience. You'll need to understand a lot of concepts to get started, and these concepts will be the foundation of more concepts to follow. The journey through Java EE will be a bit of an alphabet soup of acronyms, but hang tough—you'll catch on, and we'll do our best on our end to help you make sense of it. Here, we'll provide an overview of some important Java EE features and concepts.

Java EE Clients and Servers

Up to this point, we've been using terms like *client* and *server* somewhat loosely. These terms represent fairly specific concepts in the world of distributed computing and Java EE.

A Java EE client can be a console (text) application written in Java, or a GUI application written using the Java Foundation Classes (JFC) and Swing or AWT. These types of clients are often called *fat clients* because they tend to have a fair amount of supporting code for the user interface.

Java EE clients may also be web-based clients; that is, clients that live inside a browser. Because these clients offload much of their processing to supporting servers, they have very little in the way of supporting code. This type of client is often called a *thin client.* A thin client may be a purely HTML-based interface, a JavaScript-enriched page, or one that contains a fairly simple applet where a slightly richer user interface is needed.

It would be an oversimplification to describe the application logic called by the Java EE clients as the "server," although it is true that, from the perspective of the developer of the client-side code, that illusion is in no small way the magic of what the Java EE platform provides. In fact, the Java EE application server is the actual server that connects the client application to the business logic.

The server-side components created by the application developer can be in the form of web components and business components. Web components come in the form of JSPs or Servlets. Business components, in the world of Java EE, are EJBs.

These server-side components rely on the Java EE framework. Java EE provides support for the server-side components in the form of *containers*.

Containers

Containers are a central theme in the Java EE architecture. Earlier in this chapter, we talked about application infrastructure in terms of the plumbing and electricity that a house provides for its inhabitants. Containers are like the rooms in the house. People and things exist in the rooms, and interface with the infrastructure through well-defined interfaces. In an application server, web and business components exist inside containers and interface with the Java EE infrastructure through well-defined interfaces.

In the same way that application developers can partition application logic into tiers of specific functionality, the designers of Java EE have partitioned the infrastructure logic into logical tiers. They have done the work of writing the application support infrastructure—things that you would otherwise need to build yourself. These include security, data access, transaction handling, naming, resource location, and the guts of network communications that connect the client to the server. Java EE provides a set of interfaces that allow you to plug your application logic into that infrastructure and access those services.

Think of containers as playing a role much like a video gaming console into which you plug game cartridges. As shown in Figure 1-6, the gaming console provides a point of interface for the game—a suite of services that lets the game be accessed by the user and allows the game to interact with the user. The game cartridge needs to be concerned only with itself; it doesn't need to concern itself with how the game is displayed to the user, what sort of controller is being used, or even if the household electricity is 120VAC or 220VAC. The console provides a container that abstracts all of that stuff out for the game, allowing the game programmer to focus solely on the game and not worry about the infrastructure.

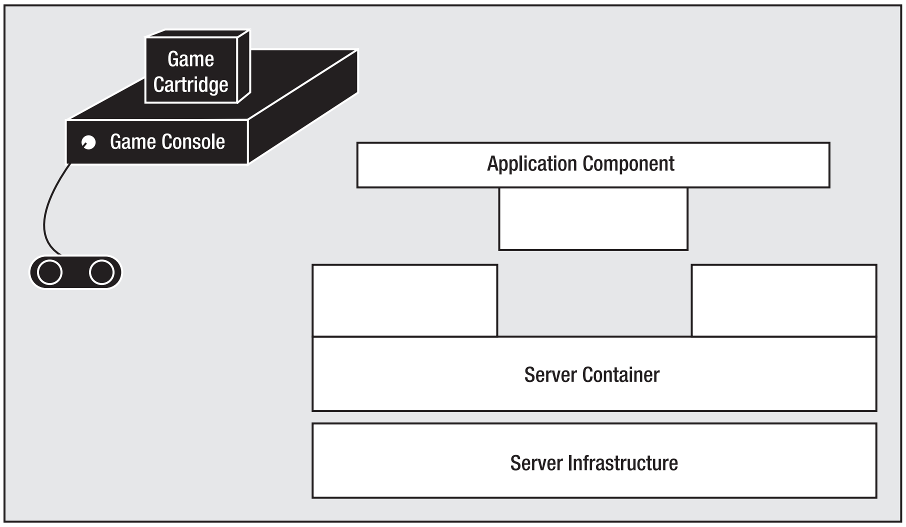

Figure 1-6. *The container provides an environment for components and an interface between the components and the services of the server.*

If you've ever created an applet, you're already familiar with the concept of containers. Most web browsers provide a container for applet components, as illustrated in Figure 1-7. The browser's container for applets provides an environment for the applet. The browser and the container know how to interact with any applet because all applets implement the `java.applet.Applet` class interface. When you develop applets, you are relieved of the burden of interfacing with a web browser, and are free to spend your time and effort on the applet logic. You do not need to be concerned with the issues associated with making your application appear to be an integral part of the web browsers.

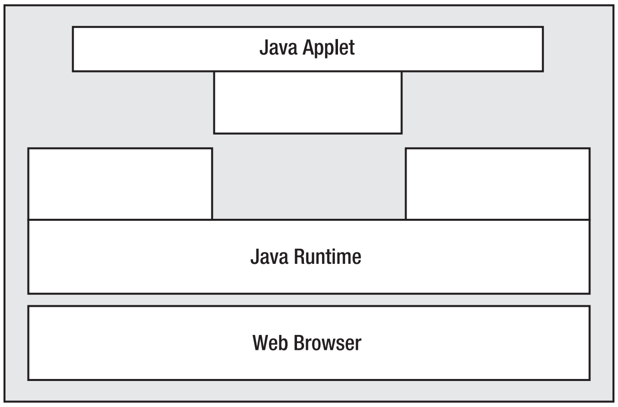

Figure 1-7. *Browsers don't directly access applets. Instead, the applet runs in a container inside the browser. The container provides an environment for the applet and acts as an interface between the browser and the applet.*

Java EE provides server-side containers for the same reason: To provide a well-defined interface, along with a host of services that allow application developers to focus on the business problems they're trying to solve, without worrying about the plumbing and electricity. Containers handle all of the mundane details involved with starting up services on the server side, activating the application logic, and cleaning up the component.

Java EE and the Java platform provide containers for web components and business components. These containers—like the gaming console analogy presented earlier in the chapter—provide an environment and interface for components that conform to the container's established interfaces. The containers defined in Java EE include a container for Servlets, JSPs, and EJBs.

Java Servlets

You are no doubt familiar with accessing simple, static HTML pages using a browser that sends a request to a web server, which, in turn, sends back a web page that's stored at the server, as illustrated in Figure 1-8. In that role, the web server is simply being used as a virtual librarian that returns a document based on a request.

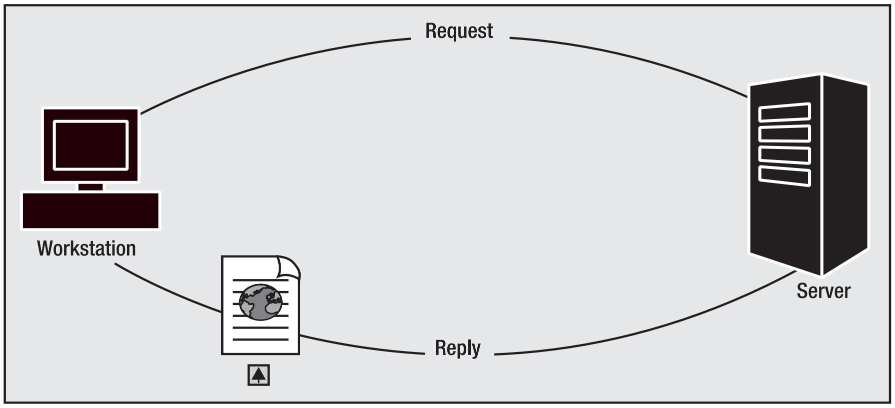

Figure 1-8. *A web browser running on a workstation sends a request to a web server. The server identifies the web page specified in the request and returns that web page to the browser.*

That model of serving up static web pages doesn't provide for dynamically generated content, though. For example, suppose that the web client wants the server to return a list of HTML documents based on some query criteria. In that case, some means of generating HTML on the fly and returning it to the client is needed, as illustrated in Figure 1-9.

Servlets are one of the technologies developed to enhance servers. A Servlet is a Java component implementing the `javax.servlet.Servlet` interface. It is invoked as a result of a client request for that particular Servlet. The Servlet model is fairly generic and not necessarily bound to the Web and HTTP, but all of the Servlets that you'll encounter will fall into that category. The web server receives a request for a given Servlet in the form of an HTTP query. The web server, in turn, invokes the Servlet and passes back the results to the requesting client. The Servlet can be passed parameters from the requesting web client. The Servlet is free to perform whatever computations it cares to, and returns results to the client in the form of HTML.

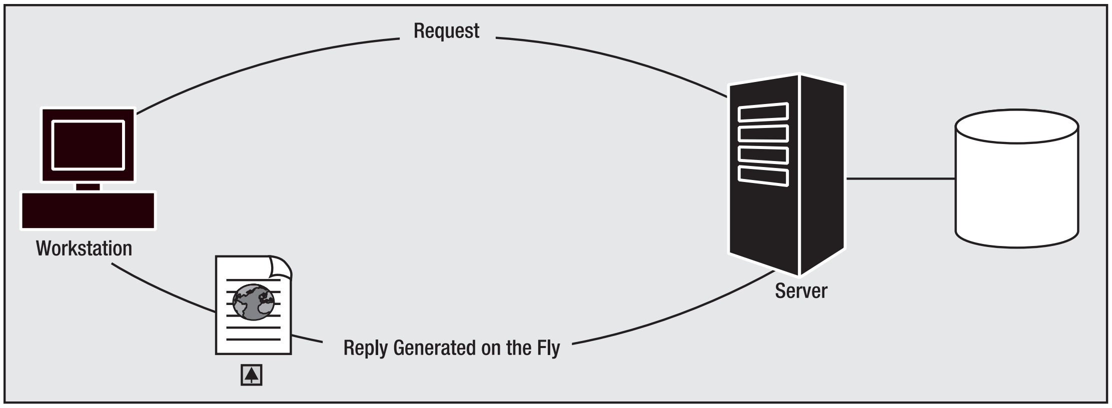

Figure 1-9. *Web servers can be supplemented by other processes that perform data access or some other processing. This other processing is then converted into an HTML web page and sent back to the client. Thus, web servers that were designed to serve static content can be enhanced to provide dynamic content.*

The Servlet itself is managed and invoked by the Java EE Servlet container. When the web server receives the request for the Servlet, it notifies the Servlet container, which will load the Servlet as necessary, and invoke the appropriate `javax.servlet.Servlet` interface service method to satisfy the request.

■**Note** Servlets were not the first technology designed to enhance web servers. One of the earlier solutions is known as the Common Gateway Interface (CGI). CGI provided a means for a server to call an external process that performed additional work for the server. If you've done any web application programming using CGI, you'll be familiar with the limitations of that mechanism, including lack of portability (CGI programs were often written in C) and no intrinsic support for session management (a much-overused example is the ability to maintain a list of items in a virtual shopping cart). If you have not done any development using CGI, consider yourself lucky and take our word for it—life with Java EE is a whole lot better!

Java Servlets are portable, and as you will see in later chapters, the Servlet containers provide support for session management that allows you to write complex web-based applications. Servlets can also incorporate JavaBean components (which share little more than a name with Enterprise JavaBeans) that provide an additional degree of application compartmentalization. Servlets are covered in detail in Chapter 6.

JavaServer Pages (JSPs)

JSPs, like Servlets, are concerned with dynamically generated web content. These two web components—Servlets and JSPs—comprise a huge percentage of the content of real-world Java EE applications.

Building Servlets involves building Java components that emit HTML. In a lot of cases, that works out well. However, that approach isn't very accessible for people who spend their time on the visual side of building web applications and don't necessarily care to know much about software development. Enter JSP. JSP pages are HTML-based text documents with chunks of Java code called *scriptlets* embedded into the HTML document.

When JSPs are deployed, something remarkable happens: The contents of the JSP are rolled inside out, like a sock, and a Servlet is created based on the embedded tags and Java code scriptlets, as shown in Figure 1-10. This happens pretty much invisibly. If you care to, you can dig under the covers and see how it works (which makes learning about Servlets all the more worthwhile).

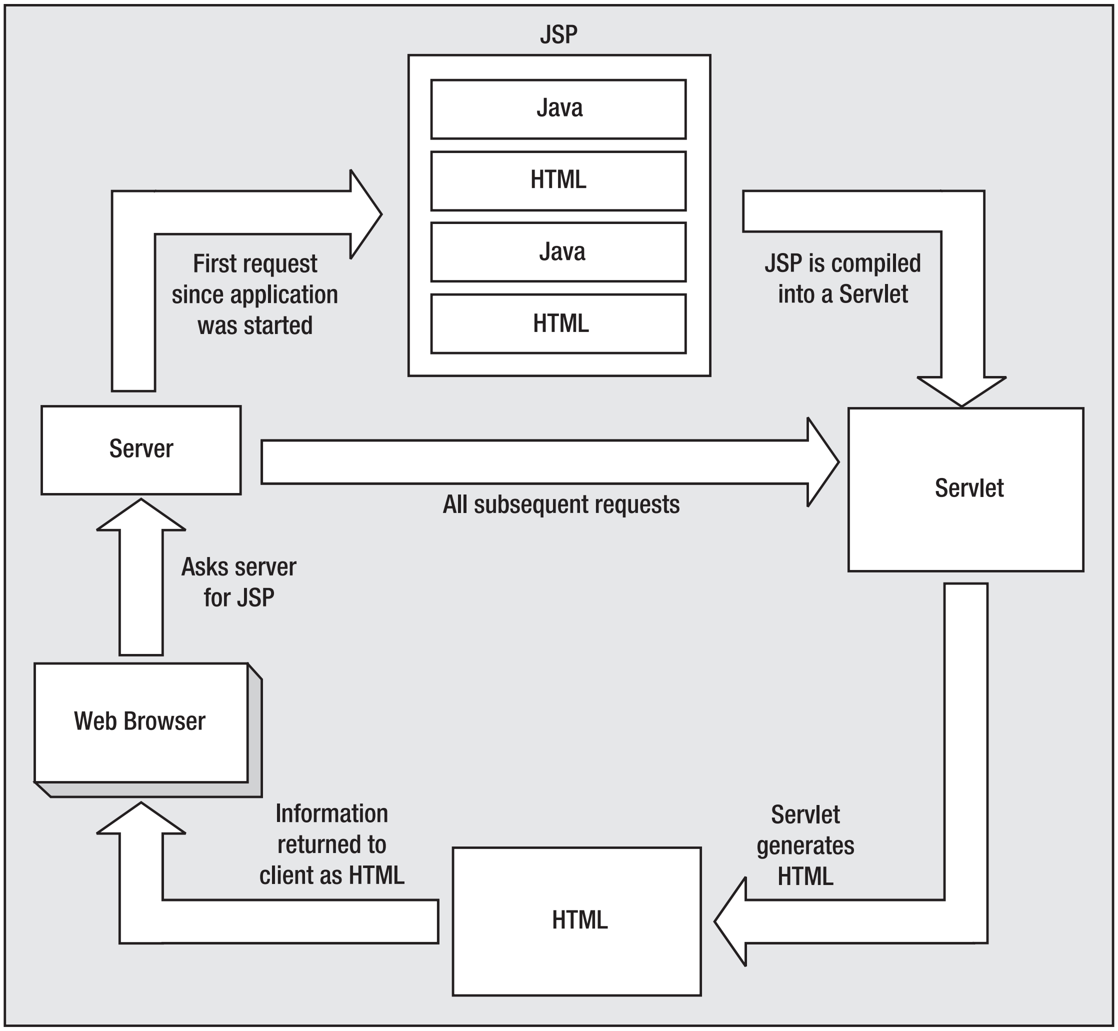
JSP
Java
HTML
Java
HTML
First request since application was started
JSP is compiled into a Servlet
Server
All subsequent requests
Servlet
Asks server for JSP
Web Browser
Information returned to client as HTML
Servlet generates HTML
HTML

Figure 1-10. *When a web server receives a request for a JSP, it passes the request to the JSP container (not shown). If the JSP page has not been translated, the container translates the JSP into a Java Servlet source file, and then compiles the source file into a class. The Servlet class is loaded and the request is passed to the class. The Servlet processes the request and returns the result to the client. All subsequent requests are routed directly to the Servlet class, without the need to translate or compile again.*

You may have had some exposure to JavaScript, which is a Java-like scripting language that can be included within a web page, and is executed by the web browser when a page containing JavaScript code is sent to the browser. JSP is a little like that, but the code is compiled and executed at the *server*, and the resulting HTML is fed back to the requesting client. JSP pages are lightweight and fast (after the initial compilation to the Servlet), and they provide a lot of scalability for web-based applications.

Developers can create both static and dynamic content in a JSP page. Because content based on HTML, XML, and so on forms the basis of a JSP page, a nontechnical person can create and update that portion of a page. A more technical Java developer can create the snippets of Java code that will interface with data sources, perform calculations, and so on—the dynamic stuff.

Since an executing JSP is a Servlet, JSP provides the same support for session management as Servlets. JSPs can also load and call methods of JavaBean components, access server-based data sources, or perform complex calculations at the server.

JSPs are introduced in detail in Chapter 3. Chapter 4 continues with more advanced JSP concepts.

JavaServer Faces (JSF)

JSF is a relatively new technology that attempts to provide a robust, rich user interface for web applications. JSF is used in conjunction with Servlets and JSPs.

When using just JSPs or Servlets to generate the presentation, your user interface is limited to what can be implemented in HTML. HTML does provide a good set of user interface components, such as lists, check boxes, radio buttons, fields, labels, and buttons. Alternatively, the client might be implemented as an applet. Applets can provide a rich user interface, but they do require the client to download and execute code in the browser.

The main drawback with both Servlet-generated HTML and applets is that the user interface components still must be connected to the business logic. When using this solution, much of your time as a developer will be spent retrieving and validating request parameters, and passing those parameters to business logic components.

JSF provides a component-based API for building user interfaces. The components in JSF are user interface components that can be easily put together to create a server-side user interface. The JSF technology also makes it easy to connect the user interface components to application data sources, and to connect client-generated events to event handlers on the server.

The JSF components handle all the complexity of managing the user interface, leaving the developer free to concentrate on business logic. The flexibility comes from the fact the user interface components do not directly generate any specific presentation code. Creating the client presentation code is the job of custom renderers. With the correct renderer, the same user interface components could be used to generate presentation code for any arbitrary device. Thus, if the client's device changed, you would simply configure your system to use a renderer for the new client, without needing to change any of the JSF code. At the moment, the most common presentation format is HTML, and JSF comes with a custom renderer to create HTML user interfaces. JSF technology is covered in Chapter 5.

JDBC

If you've done anything at all on the Web other than simple surfing, you've probably used a database. Of course, that database has been hidden behind a fancy user interface, but you've used one nonetheless.

Have you searched for books or other products at `www.amazon.com` or `www.costco.com` or any other online store? The information about the products for sale is kept in some kind of database.

Have you searched for web sites on `www.google.com` or `www.yahoo.com` or any other search engine? Information about web pages and the data in them is kept is some kind of database.

Have you looked for information about public laws (`thomas.loc.gov`), driving directions (`www.mapquest.com`), or satellite imagery (`www.terraserver.com`)? This information is kept in some kind of database.

The examples can go on and on. The point should be clear though: Almost any type of nontrivial application will use a database of some kind. In the previous sentence, the term *database* is used in its loosest most general meaning as a collection of some data. That database could be anything from a text file of information for very simple applications to full-blown, enterprise-level relational or object databases for very complex systems. It could also include other data-storage systems, such as directories.

Most Java EE applications will include some kind of data-storage solution. Most often, that data-storage solution will be a relational database server of some kind. The database server may be an integral part of the application server, or it may be an application separate from the application server.

In any case, your application components need some means to communicate with the data-storage system. That is the job of JDBC. JDBC is a set of common APIs and system-specific libraries for communicating with a data-storage system. By communicating with the data-storage system through the common APIs, you can concentrate on the data, without needing to learn custom syntax for the particular data-storage system; that job is left to the system-specific library.

Most JDBC applications are used to communicate with a relational database. In a relational database, data is stored, conceptually, in tables. Each row in a table represents a set of data—a customer record, product information, a web site listing, and so on. And each column in the table represents a piece of data in that set. Tables can be linked by creating a *relation* between tables, thus it's called a *relational* database. For example, a database might have a table of customer information and a table of information about orders. It makes no sense to repeat customer information for each order, so the orders table would include a customer ID that corresponds to a similar piece of data in the customers table, thus relating every order to a customer.

While JDBC is used most often with relational databases, it can be used with any data-storage system, as long as someone has created a system-specific library for that data-storage system. Using JDBC in Java EE applications is covered in Chapters 7 and 8.

EJBs

EJBs are to Java EE what Mickey Mouse is to Disney—they represent the flagship technology of the platform. When Java EE is mentioned, EJBs are what immediately comes to mind. We mentioned earlier that Java EE is a whole lot more than EJB, but we don't mean to trivialize EJBs; the attention that the technology gets is certainly merited.

In order to better understand what EJBs are and do, it helps to start out with Java's Remote Method Invocation (RMI). If you're not already familiar with RMI, or if you need a quick overview or a refresher, you may want to refer to `http://java.sun.com/rmi`.

RMI is Java's native means of allowing a Java object to run on one computer and have its methods called by another object running on a separate computer across a network. In order to create a remote object with RMI, you first design an interface that extends the `java.rmi.Remote` interface. This interface defines the operations that you want to expose on your remote object. The next step is to design the remote object as a Java class that implements the interface you've defined. This class extends the `java.rmi.server.UnicastRemoteObject` class, which provides the necessary network communications between this object and the objects that call it. Finally, you write an application that creates an instance of this class and registers that instance with the RMI registry.

The RMI registry is a simple lookup service that provides a means to associate a name with an object, analogous to the way a phone directory associates a name to a phone number. The same registry service is used by the client application, which requests a named object from the registry. Once it receives a local reference to the remote object, it can call the methods of the object; however, rather than executing the method on the client's computer, the method call is passed across the network and executed on the machine where the remote object resides.

What RMI provides is a bare-bones client/server implementation. It provides the basic stuff: a registry for lookup, the guts of network communication for invoking operations and passing parameters to and from remote objects, and a basic mechanism for managing access to system resources as a safeguard against malicious code running on a remote computer.

However, RMI is lightweight. It's not designed to satisfy the requirements of enterprise-class distributed applications. It lacks the essential infrastructure that enterprise-class applications rely on, such as security, data access, transaction management, and scalability. While it supplies base classes that provide networking, it doesn't provide a framework for an application server that hosts your server-side business components and scales along with your application. You must write the client and the server applications. This is where EJBs come into the picture.

EJBs are Java components that implement business logic. This allows the business logic of an application (or suite of applications) to be compartmentalized into EJBs and kept separate from the front-end applications that use that business logic.

The Java EE architecture includes a server that is a container for EJBs. The EJB container loads the bean as needed, invokes the exposed operations, applies security rules, and provides the transaction support for the bean. If it sounds to you like the EJB container does a lot of work, you're right—the container provides all of the necessary plumbing and wiring needed for enterprise applications.

As you'll see in Chapter 9, building EJBs follows the same basic steps as creating an RMI object. You create an interface that exposes the operations or services provided by the EJB. You then create a class that implements the interface. When you deploy an EJB to an application server, the EJB is associated with a name in a registry. Clients can look up the EJB in the registry, and then remotely call the methods of the EJB. Since the EJB container provides all of the enterprise plumbing, you get to spend more time building your application and less time messing around with trying to shoehorn in services like security and transaction support.

EJBs come in a few different flavors: session beans, entity beans, and message beans. *Session beans*, as the name implies, live only as long as the conversation, or session, between the client application and the bean lasts. The session bean's primary reason for being is to provide application services, defined and designed by the application developer, to client applications. Depending on the design, a session bean may maintain state during the session or may be stateless. With a *stateful* EJB, when a subsequent request comes from a client, the values of the internal member variables have the same values they had when the previous request ended, so that the EJB can maintain a conversation with the client. A *stateless* EJB provides business rules through its exposed operations but doesn't provide any sense of state; that responsibility is delegated to the client.

Entity beans represent business objects—such as customers, invoices, and products—in the application domain. These business objects are persisted so they can be stored and retrieved at will. The Java EE architecture provides a lot of flexibility for the persistence model. You can defer all of the work of storing and retrieving the bean's state information to the container, as shown in Figure 1-11. This is known as *container-managed persistence*.

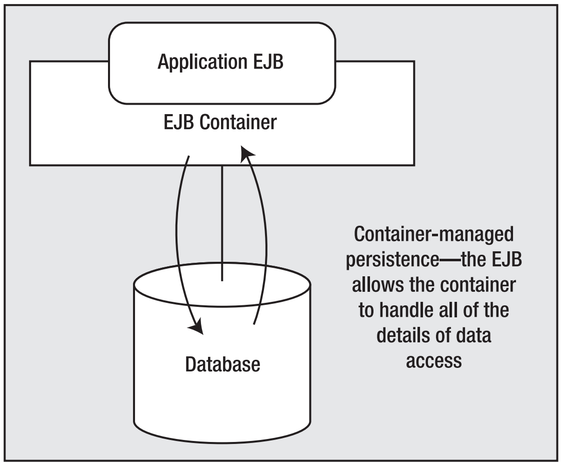
Application EJB
EJB Container
Database
Container-managed persistence—the EJB allows the container to handle all of the details of data access

Figure 1-11. *In container-managed persistence, the EJB container is responsible for all actions required to save the state of the EJB to some persistent store, usually a database.*

Alternatively, the Java EE architecture allows you to have complete control over how the EJB is persisted (which is very useful when you're dealing with interfacing your Java EE system to a legacy application!). This is known as *bean-managed persistence* and is illustrated in Figure 1-12.

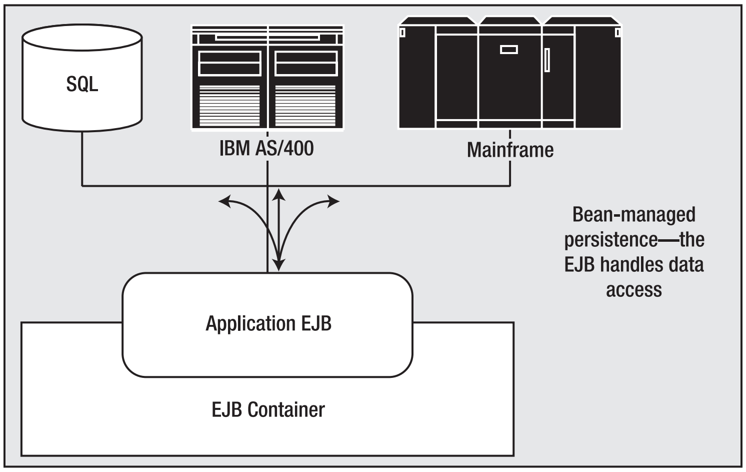

Figure 1-12. *With bean-managed persistence, the developer must manage all aspects of persisting the state of the EJB.*

The third type of EJB, the *message bean*, provides a component model for services that listen to Message Service messages, as illustrated in Figure 1-13. The Java EE platform includes a message queue that allows applications to post messages to a queue, as well as to *subscribe* to queues that get messages. The advantage of this particular way of doing things is that the sender and the receiver of the message don't need to know anything about each other. They need to know only about the message queue itself. This differs from a client/server model, where a client must know the server so that it can make a connection and a specific request, and the server sends the response directly to the client. One example of using a message queue is an automated stock trading system. Stock prices are sent as messages to a message queue, and components that are interested in stock prices consume those messages. With message-driven EJBs, it is possible to create an EJB that responds to messages concerning stock prices and makes automatic trading decisions based on those messages.

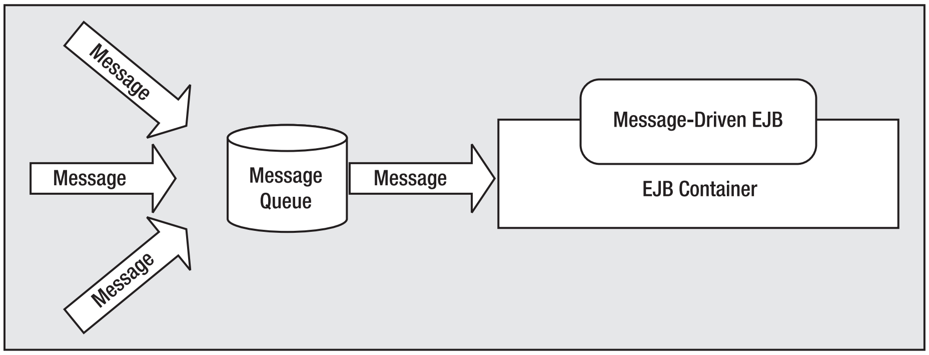

Figure 1-13. *A message queue allows senders and receivers of messages to remain unaware of each other. Senders of messages can send the message to a queue, knowing that something will get the message, but not knowing exactly what receives the message or when it will be received. Receivers can subscribe to queues and get the messages they are interested in, without needing to know who sent the message.*

You will learn a lot about the ins and outs of using session and entity beans in Chapters 9 through 12. Your Java EE applications will typically be comprised of both session and entity beans. Message beans are covered in Chapter 14. They're not used as frequently as the other flavors in most applications, but they're still pretty darn cool!

XML Support

Extensible Markup Language (XML) is a significant cornerstone for building enterprise systems that provide interoperability and are resilient in the face of changes. There are several key technologies in Java EE that rely on XML for configuration and integration with other services.

Java EE provides a number of APIs for developers working with XML. Java API for XML Processing (JAXP) provides support for generating and parsing XML with both the Document Object Model (DOM), which is a tree-oriented model, and the Simple API for XML (SAX), which is a stream-based, event-driven processing model.

The Java API for XML Binding (JAXB) provides support for mapping XML to and from Java classes. It provides a compiler and a framework for performing the mapping, so you don't need to write custom code to perform those transformations.

The Java API for XML Registries (JAXR), Java API for XML Messaging (JAXM), and Java API for XML-based Remote Procedure Calls (JAX-RPC) round out the XML API provisions. These sets of APIs provide support for SOAP and web services (discussed in the following section).

This book assumes that you are familiar with XML basics. If you need a refresher on XML, you might want to review the Sun Java XML tutorial at `http://java.sun.com/xml/tutorial_intro.html`.

Web Services

The World Wide Web is becoming an increasingly prevalent backbone of business applications. The endpoints that provide web applications with server-side business rules are considered *web services*. The World Wide Web Consortium (W3C), in an effort to unify how web services are published, discovered, and accessed, has sought to provide more concrete definitions for web services. Here's a definition from the Web Services Architecture, Working Group Note 11 (`www.w3.org/TR/ws-arch`):

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

This definition contains some specific requirements:

- A web service allows one computer to request some service from another machine.
- Service descriptions are machine-processible.
- Systems access a service using XML messages sent over HTTP.

The W3C has established the Web Service Description Language (WSDL) as the XML format that is used by web services to describe their services and how clients access those services. In order to call those services, clients need to be able to get their hands on those definitions. XML registries provide the ability to publish service descriptions, search for services, and obtain the WSDL information describing the specifics of a given service.

There are a number of overlapping XML registry service specifications, including ebXML and Universal Description, Discovery, and Integration (UDDI). The JAXR API provides an implementation-independent API for accessing those XML registries.

Simple Object Access Protocol (SOAP) is the *lingua franca* used by web services and their clients for invocation, parameter passing, and obtaining results. SOAP defines the XML message standards and data mapping required for a client application to call a web service and pass it parameters. The JAX-RPC API provides an easy-to-use developer interface that masks the complex underlying plumbing.

Not surprisingly, the Java EE architecture provides a container that hosts web services, and a component model for easily deploying web services. Chapters 15 and 16 in this book cover SOAP and web services.

Transaction Support

One of the basic requirements of enterprise applications is the ability to allow multiple users of multiple applications to simultaneously access shared databases and to absolutely ensure the integrity of that data across those systems. Maintaining data consistency is no simple thing.

Suppose that your application was responsible for processing bank deposits, transfers, and withdrawals. Your application is processing a transfer request from one account to another. That process seems pretty straightforward: deduct the requested amount from one account and add that same amount to the other account. Suppose, however, that immediately after deducting the sum from the source account, something went horribly wrong—perhaps a server failed or a network link was severed—and it became impossible to add the transfer to the target account. At that point, the data's integrity has been compromised (and worse yet, someone's money is now missing).

Transactions can help to address this sort of problem. A transaction represents a set of activities that collectively will either succeed and be made permanent, or fail and be discarded. In the situation of a bank account transfer, you could define the transaction boundaries to start as the transfer amount is withdrawn from the source account, and end after the target account is updated successfully. When the transaction had been made successfully, the changes are committed. Any failure inside the transaction boundary would result in the changes being rolled back and the account balances restored back to the original values that existed before the start of the transaction.

Java EE—and the EJB in particular—provides substantial transaction support. The EJB container provides built-in support for managing transactions, and allows the developer to specify and modify transaction boundaries without changing code. Where more complex transaction control is required, the EJB can take over the transaction control from the container and perform fine-grained or highly customized transaction handling.

You'll find an introduction to transactions, in the context of database transactions with JDBC, in Chapter 8.

Security

Security is a vital component in enterprise applications, and Java EE provides built-in security mechanisms that are far more secure than homegrown security solutions that are typically added as an afterthought.

Java EE allows application resources to be configured for anonymous access where security isn't a concern. Where there are system resources that need to be secured, however, it provides authentication (making sure your users really are who they say they are) and authorization (matching up users with the privileges they are granted).

Authorization in Java EE is based on roles of users of applications. You can classify the roles of users who will be using your application, and authorize access to application components based on those roles. Java EE provides support for declarative security that is specified when the application is deployed, as well as programmatic security that allows you to build fine-grained security into the Java code.

■ **Note** If you're interested in learning more about Java EE-specific security, refer to a book devoted to Java security. One such book is *Hacking Exposed J2EE & Java*, by Art Taylor, Brian Buege, and Randy Layman (Osborne/McGraw Hill, 2002; ISBN 0-07-222565-3).

Sample Java EE Architectures

There is no such thing as a single software architecture that fits all applications, but there are some common architectural patterns that reappear frequently enough to merit attention.

As we explained earlier in the chapter, Java EE provides a platform that enables developers to easily create *n*-tier (multitier) applications in a number of different configurations. The basic n-tier architecture can have any number of components in each tier, and any combination between tiers.

Here, we will briefly review some architectures that you're likely to run into as you examine and develop Java EE-based systems. Each one of these has its own merits and strong points. We present them here to illustrate that there are a number of ways to put together applications and as a short "field guide" for identifying these architectures as you spot them in the wild.

Application Client with EJB

Figure 1-14 shows an architecture where an application client composes the presentation tier and communicates with an EJB in the business tier.

The client application is built as a stand-alone (JFC/Swing or console) application. The application relies on business rules implemented as EJBs running on a separate machine.

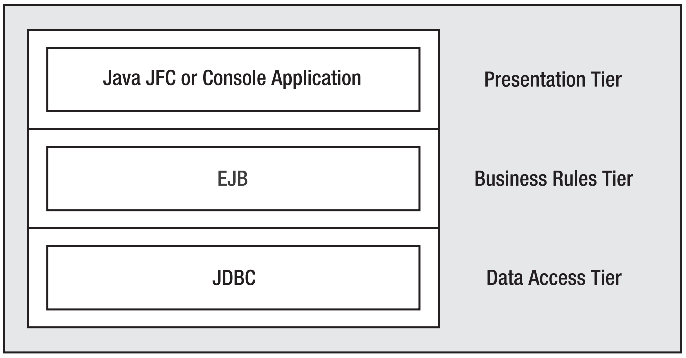

Figure 1-14. *An application client can be implemented as a normal Java application based on Swing or AWT, or even as a console application. The client communicates with EJBs in the business tier.*

JSP Client with EJB

Figure 1-15 shows an architecture based on JSPs. JSPs on the server interface with the business layer in response to requests. The response is generated as a web page, which is sent to the client's web browser.

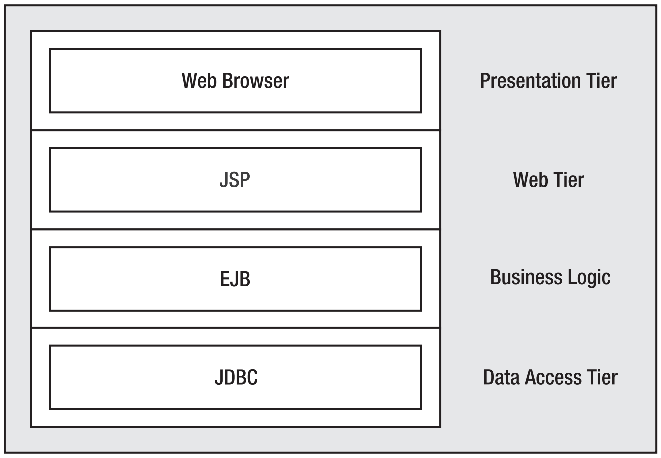

Figure 1-15. *In this architecture, the application client is a web page in a browser. The web page is generated by a JSP that communicates with the business layer.*

The client in this architecture is a web browser. JSPs access business rules and generate content for the browser.

Applet Client with JSP and Database

Figure 1-16 shows an architecture similar to the one shown in Figure 1-15. In this case, the client is a Java applet that resides entirely in the presentation tier and communicates with the business layer. Although the business layer could be an EJB, as in the previous example, in this example, the business layer is constructed from JSPs.

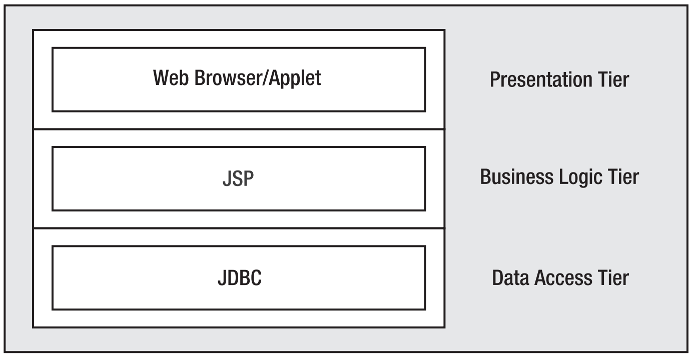

Figure 1-16. *An applet in the presentation layer can communicate over the network with JSPs (or Servlets) in the business layer.*

The Java applet is used within a web page to provide a more interactive, dynamic user interface for the user. That applet accesses additional content from JSPs. Even though JSPs are normally used to generate HTML web pages, a JSP could consist of only business logic. The JSPs access data from a database using the JDBC API.

Web Services for Application Integration

Even though Java EE is Java-based, a web application architecture is not limited solely to Java components. An obvious example is the data tier, where many enterprise-level databases are implemented in a high-level language such as C or C++. Similarly, components in a tier can be implemented in languages other than Java, as long as they provide a well-defined interface that allows for interprocess communication. In the example shown in Figure 1-17, a client application implemented in C# accesses data from a web service implemented in Java.

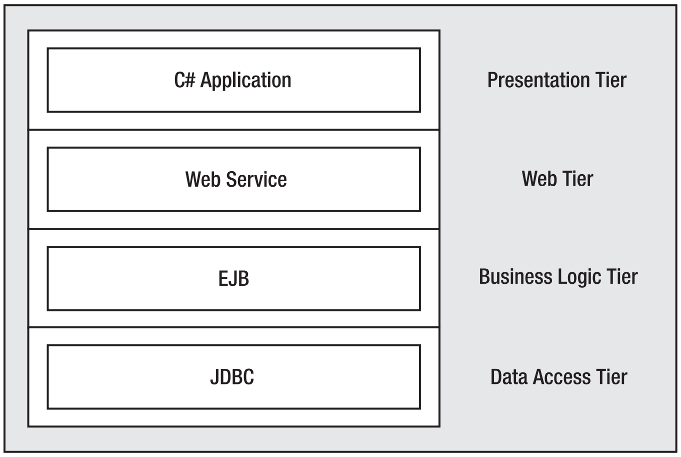

Figure 1-17. *The web service interface provides a well-defined interface between clients and web services. It allows clients to be implemented in any language that supports making HTTP requests to a server. For example, clients written in C# can format a web service request, which can be serviced by an EJB written in Java and running in a Java EE application server.*

Summary

In this opening chapter, we provided an overview of Java EE and how all the various bits fit together to enable you to create powerful business components. We first looked at what Java EE is and tackled the obvious issue of moving from creating desktop applications with J2SE to building enterprise-level applications and dynamic, data-driven web sites using Java EE. We covered how the two relate to each other and how they differ from each other, as well as looking at how applications are built using Java EE.

Java EE provides a platform for developing and deploying multitiered, distributed applications that are designed to be maintainable, scalable, and portable. Just as an office building requires a lot of hidden infrastructure of plumbing, electricity, and telecommunications, large-scale applications require a great deal of support infrastructure. This infrastructure includes database access, transaction support, and security. Java EE provides that infrastructure and allows you to focus on your applications.

Building distributed applications (software with components that run as separate processes, or on separate computers) allows you to partition the software into layers of responsibility, or *tiers*. Distributed applications are commonly partitioned into three primary tiers: presentation, business rules, and data access. Partitioning applications into distinct tiers makes the software more maintainable and provides opportunities for scaling up applications as the demand on those applications increases.

Java EE architecture is based on the idea of building applications around multiple tiers of responsibility. The application developer creates components, which are hosted by the Java EE containers. Containers play a central theme in the Java EE architecture.

Servlets are one type of Java EE web component. They are Java classes that are hosted within, and invoked by the Java EE server by requests made to, a web server. These Servlets respond to those requests by dynamically generating HTML, which is then returned to the requesting client.

JSPs are very similar in concept to Servlets, but differ in that the Java code is embedded within an HTML document. The Java EE server then compiles that HTML document into a Servlet, and that Servlet generates HTML in response to client requests.

JSF is a Java EE technology designed to create full and rich user interfaces. Standard user interface components are created on the server and connected to business logic components. Custom renderers take the components and create the actual user interface.

JDBC is a technology that enables an application to communicate with a data-storage system. Most often that is a relational database that stores data in tables that are linked through logical relations between tables. JDBC provides a common interface that allows you to communicate with the database through a standard interface without needing to learn the syntax of a particular database.

EJBs are the centerpiece of Java EE and are the component model for building the business rules logic in a Java EE application. EJBs can be designed to maintain state during a conversation with a client, or can be stateless. They can also be designed to be short-lived and ephemeral, or can be persisted for later recall. EJBs can also be designed to listen to message queues and respond to specific messages. Java EE is about a lot more than EJBs, although EJBs do play a prominent role.

The Java EE platform provides a number of services beyond the component hosting of Servlets, JSPs, and EJBs. Fundamental services include support for XML, web services, transactions, and security.

Extensive support for XML is a core component of Java EE. Support for both document-based and stream-based parsing of XML documents forms the foundation of XML support. Additional APIs provide XML registry service, remote procedure call invocation via XML, and XML-based messaging support.

Web services, which rely heavily on XML, provide support for describing, registering, finding, and invoking object services over the Web. Java EE provides support for publishing and accessing Java EE components as web services.

Transaction support is required in order to ensure data integrity for distributed database systems. This allows complex, multiple-step updates to databases to be treated as a single step with provisions to make the entire process committed upon success, or completely undone by rolling back on a failure. Java EE provides intrinsic support for distributed database transactions.

Java EE provides configurable security to ensure that sensitive systems are afforded appropriate protection. Security is provided in the form of authentication and authorization.

After reading this chapter, you should know:

- Containers provide an environment and infrastructure for executing Java EE components.
- Servlets and JSPs provide server-side processing and are used to create the presentation layer of a Java EE system.
- JSF provides user interface components that make it easy to create flexible user interfaces and connect user interface widgets to business objects.
- JDBC is an interface to database systems that allows developers to easily read and persist business data.
- EJBs represent business objects in a Java EE application. EJBs come in various categories, including stateful session beans, stateless session beans, entity beans, and message-driven beans.

- Java EE systems can be used to develop service-oriented architectures or web services systems. A web service architecture is one that provides machine-to-machine services over a network using a well-defined protocol.
- Some of the essential architectural patterns used in Java EE applications include an application client with EJBs, a JSP client with EJBs, an applet client with JSPs and a database, and web services used for application integration.

That's it for your first taste of how Java EE works and why it is so popular. In the next chapter, you'll see the steps required to set up your environment and make it ready for developing powerful Java EE applications.