

Building Scalable and High-Performance **Java**TM

Web Applications Using J2EETM Technology



Greg Barish



Building Scalable and High-Performance Java™ Web Applications Using J2EE™ Technology

Greg Barish

◆◆ Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid • Capetown
Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

Pearson Education Corporate Sales Division
201 W. 103rd Street
Indianapolis, IN 46290
(800) 428-5331
corpsales@pearsoned.com

Visit AW on the Web: www.aw.com/cseng/

Library of Congress Cataloging-in-Publication Data

Barish, Greg.

Building scalable and high-performance Java Web applications using J2EE technology / Greg Barish.

p. cm.

Includes bibliographical references and index.

ISBN 0-201-72956-3 (alk. paper)

1. Java (Computer program language) 2. Web sites—Design. I. Title.

QA76.73.J38 B36 2002

005.2'762--dc21

2001056071

Copyright © 2002 Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN 0-201-72956-3

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—CRW—0504030201

First printing, December 2001



Contents

Preface xv

CHAPTER 1 Scalable and High-Performance Web Applications 1

The Emergence of Web Applications	1
<i>Basic Definitions</i>	2
<i>The Nature of the Web and Its Challenges</i>	3
Performance and Scalability	5
<i>Performance</i>	6
<i>Scalability</i>	7
The Internet Medium	8
<i>Wide Audience</i>	9
<i>Interactive</i>	10
<i>Dynamic</i>	10
<i>Always On</i>	11
<i>Integrated</i>	12
<i>Lack of Complete Control</i>	12
Measuring Performance and Scalability	13
<i>Measuring Performance</i>	14
<i>Measuring Scalability</i>	16
<i>Throughput and Price/Performance</i>	20
Scalability and Performance Hints	21
<i>Think End-to-End</i>	21
<i>Scalability Doesn't Equal Performance</i>	21
<i>Measure Scalability by Comparison</i>	22
Summary	22

CHAPTER 2 Web Application Architecture 23

Web Application Terminology	23
Application Requirements	25
<i>Business Logic</i>	25

<i>Data Management</i>	26
<i>Interface</i>	26
Web Requirements	27
<i>Network Connectivity</i>	28
Abstract Web Application Architecture	29
<i>From Client to Server: Thin and Fat Clients</i>	29
<i>Persistent Data Management</i>	30
N-tier Application Architecture	32
<i>The Client</i>	32
<i>The Network</i>	33
<i>The Server</i>	36
<i>Tier-Based Designs</i>	38
<i>Multithreaded Application Servers</i>	41
<i>The Challenge of Efficient Middleware</i>	43
Scalability and Performance Hints	44
<i>Don't Always Starve Thin Clients</i>	44
<i>Use or Build Multithreaded Application Servers</i>	45
<i>Find the Right Granularity</i>	46
Summary	47

CHAPTER 3 **The J2EE Specification** 49

Overview of the Specification	49
Deployment Issues	53
<i>Packaging</i>	53
<i>Deployment Descriptor Files</i>	54
Platform Technologies and Services	57
<i>Component Communication via RMI-IIOP</i>	57
<i>Transaction Management Using the Java Transaction API</i>	59
<i>JNDI for Resource Location</i>	62
J2EE and Your Architecture	64
Summary	65

CHAPTER 4 **Scalability and Performance Techniques** 67

Caching and Replication	67
Parallelism	74
Redundancy	78
Asynchrony	80
Resource Pooling	82
Summary	88

CHAPTER 5 HTTP Client/Server Communication 91

The HTTP Protocol	91
Deployment Paradigms	93
<i>Applications with Browser Clients</i>	93
<i>Applications with Nonbrowser Clients</i>	94
HTTP Efficiency	95
HTTP Details	96
<i>Semantics</i>	97
<i>HTTP Requests</i>	99
<i>The GET Method</i>	99
<i>The POST Method</i>	105
<i>HTTP 1.1 Caching</i>	106
<i>Connection Management</i>	110
Scalability and Performance Hints	113
<i>Use GET and POST Judiciously</i>	113
<i>Consider HTTP for Nonbrowser Clients</i>	114
<i>Promote HTTP Response Caching</i>	114
<i>Support Persistent Connections</i>	116
Summary	116

CHAPTER 6 Request Processing 119

The General Problem	120
Specific Challenges	121
<i>Connection Management</i>	123
<i>Data Marshalling</i>	124
<i>Request Servicing</i>	124
<i>Data Locality in Caching Environments</i>	127
Request-Processing Modes	128
<i>Synchronous Communication</i>	128
<i>Asynchronous Communication</i>	129
<i>Scalability and Performance Tradeoffs</i>	133
Request Processing and J2EE	135
<i>Web Serving</i>	135
<i>Synchronous Processing with Java Servlets and JavaServer Pages</i>	135
<i>Asynchronous Processing with the Java Message Service</i>	136
Scalability and Performance Hints	137
<i>Build Asynchronous Solutions</i>	137

<i>Stream Data between Threads</i>	139
<i>Develop Efficient Remote Interfaces</i>	144
Summary	147

CHAPTER 7 Session Management with Java Servlets 149

Generating Dynamic Responses	149
<i>Common Gateway Interface</i>	150
<i>Extending the Web Server through Its API</i>	150
<i>Redirecting the Web Server Request</i>	151
Using Servlets	152
<i>Servlets and Servlet Containers</i>	152
<i>Interacting with a Servlet</i>	154
<i>Web Server and Servlet Container Integration</i>	154
Developing Servlets	157
<i>Designing the Servlet Interface</i>	157
<i>Coding the Servlet</i>	158
Servlet Execution	162
<i>Servlet Containers</i>	162
<i>Servlets and Multithreading</i>	163
Servlets and Session Management	167
<i>Session Identification</i>	167
Deploying Servlets	172
<i>Session Management with Multiple Containers</i>	172
Developing Servlets with JavaServer Pages	175
<i>Sample JSP Page</i>	175
<i>The Structure of a JSP Page</i>	177
<i>How JSP Works</i>	178
<i>JSP Directives</i>	182
<i>What Is JSP Really?</i>	183
Scalability and Performance Hints	184
<i>Use Fine-Grained Serialization</i>	184
<i>Use Hardware-Based Load Balancing</i>	184
<i>Use Servlets for Session Management, not Business Logic</i>	185
<i>Think Twice about JSP</i>	186
Summary	187

CHAPTER 8 Building Application Servers with Enterprise Java Beans 189

The Need for Application Servers	189
<i>Application Logic and Where to Deploy It</i>	190

Enterprise Java Beans: The J2EE Solution	191
How EJBs Work	192
Types of EJB	194
Sample Application	195
EJB Design	196
<i>Session Beans</i>	196
<i>Entity Beans</i>	197
<i>Message-Driven Beans</i>	203
EJB Implementation	203
<i>Session Beans</i>	203
<i>Entity Beans</i>	211
<i>Entity Beans and EJB 2.0</i>	212
<i>Message-Driven Beans</i>	224
Client/EJB Integration	228
<i>Client/EJB Communication behind the Scenes</i>	230
Scalability and Performance Hints	232
<i>Prefer Message-Driven Beans over Session Beans</i>	232
<i>If You Use Session Beans, Use Stateless Session Beans</i>	234
<i>Strive for Coarse-Grained EJB Methods</i>	236
<i>Use BMP Well or Don't Use It at All</i>	238
<i>Know Your Vendor</i>	239
Summary	240

CHAPTER 9 Messaging for Efficient Enterprise Application Integration 241

A B2B-style Working Example	241
The Java Message Service	243
JMS Concepts	243
<i>Providers</i>	244
<i>Clients</i>	244
<i>Messages</i>	244
<i>Administrative Objects</i>	246
JMS Programming Models	246
<i>Model-Specific Administrative Object Interfaces</i>	247
<i>The Synchrony of Message Consumption</i>	247
JMS Reliability versus Performance	249
<i>Client Acknowledgment</i>	249
<i>Message Persistence</i>	250
<i>Timing Dependencies and JMS Publishing Models</i>	251

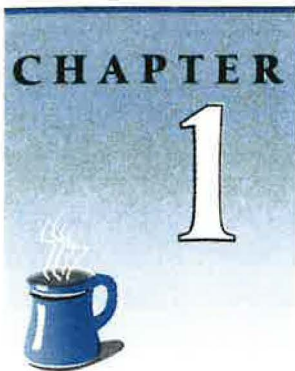
A Sample JMS Pub/Sub Application	251
<i>Developing the Message Publisher</i>	252
<i>Developing the Message Subscriber</i>	254
<i>Toward Deployment</i>	258
Scalability and Performance Hints	258
<i>Use Messaging</i>	258
<i>Understand the JMS Efficiency–Reliability Tradeoff</i>	261
Summary	262

CHAPTER 10 **Effective Database Design** 265

Database Technology and the Relational Model	267
<i>Relational Databases</i>	267
Logical Database Design	268
<i>Entities, Attributes, and Relationships</i>	269
Physical Database Design	270
<i>Tables and Rows</i>	271
<i>Constraints</i>	272
Querying a Database	273
<i>Querying Data</i>	273
<i>Nested Queries</i>	275
<i>Join Queries</i>	276
Other Important Database Objects	278
<i>Views</i>	278
<i>Stored Procedures</i>	280
<i>Triggers</i>	282
<i>Indexes</i>	283
<i>Sequences</i>	284
<i>Other Objects</i>	285
Query Processing	286
Scalability and Performance Hints	290
<i>Understand How to Use the Database</i>	290
<i>Understand When to Use the Database</i>	291
<i>Understand How Your Data Will Be Accessed</i>	293
<i>Normalize Your Data Model</i>	294
<i>Selectively Denormalize Your Model</i>	299
<i>Use Stored Procedures</i>	303
<i>Avoid Triggers and Other Implicit Execution</i>	306
<i>Know Your Vendor</i>	307
Summary	307

CHAPTER 11	Using JDBC and SQL to Query Databases Efficiently	309
	How JDBC Fits In	309
	JDBC Concepts and Objects	310
	<i>Relevant JDBC Objects and Their Relationships</i>	310
	<i>Connecting to a Database</i>	311
	Writing JDBC Queries	312
	<i>Processing a Statement</i>	312
	<i>Iterating through Results</i>	314
	<i>Executing Single Updates</i>	316
	<i>Other Kinds of Updates: Creating Tables and Stored Procedures</i>	317
	Beyond the Basics	318
	<i>Prepared Statements</i>	318
	<i>Dynamic SQL</i>	320
	<i>Transaction Management</i>	321
	<i>Bidirectional Results Iteration</i>	323
	<i>Updateable Results</i>	324
	<i>Executing Batch Updates</i>	324
	Scalability and Performance Hints	325
	<i>Use PreparedStatement When Possible</i>	325
	<i>Use Batch Updates with a Remote Database</i>	328
	<i>Don't Overcommit</i>	329
	<i>Use Multithreading to Query in Parallel</i>	330
	Summary	331
CHAPTER 12	Web Services: The Future of Web Applications	333
	Practical Use of Web Services	334
	What Exactly Is a Web Service?	335
	Web Services Technologies	338
	<i>A Quick Tour</i>	338
	<i>Putting It All Together</i>	339
	XML: Self-Describing Data	340
	<i>DTDs and Schema Languages</i>	341
	<i>Parsing XML</i>	343
	<i>XML-Related Technologies</i>	348
	Developing Web Services	349
	Describing Web Services with WSDL	349
	<i>Definitions</i>	350

<i>Example</i>	351
Invoking Web Services with SOAP	353
<i>How SOAP Works</i>	354
<i>Using SOAP over HTTP</i>	356
Registering Web Services with UDDI	357
<i>Standards</i>	359
<i>The UDDI APIs</i>	359
The Big Picture	362
<i>The Provider's Perspective</i>	362
<i>The Consumer's Perspective</i>	363
Salability and Performance Challenges	363
<i>Replicating and Load-Balancing Remote Functions</i>	363
<i>XML Parsing Performance</i>	364
<i>Parsing versus Querying XML</i>	365
Summary	368
<i>Suggested Reading</i>	369
<i>Index</i>	371



Scalable and High-Performance Web Applications

The Emergence of Web Applications

In the span of just a few years, the Internet has transformed the way information is both provided and consumed throughout the world. Its hardware and software technologies have made it possible for anyone not only to be an information consumer, but also for nearly anyone to be an information provider. Although the Internet—specifically the World Wide Web (the Web)—has been treated seriously as a platform for information sharing among the mass public for only a short time, many organizations have managed to create useful Web applications that provide significant value to consumers.

These Web applications allow consumers to buy books and compact discs online. They enable businesses to use the Internet for secure data transactions. Workers use Web applications to find jobs; employers use them to find employees; stocks are bought and sold using online applications provided by brokerages; and travelers book flight and hotel reservations using Web applications. The list goes on and on. Obviously, many useful Web applications are available on the public Internet as well as within countless corporate intranets today.

This book describes general techniques for building *high-performance and scalable enterprise Web applications*. Generally speaking, this means building applications that are reasonably and consistently fast and have a strong, gradual tolerance for rising user and request demands. Although we will spend a lot of time considering this topic in general, the core of our discussion will be phrased in terms of a solution built around the Java 2 Enterprise Edition (J2EE) specification. Now, before we dive into the details of building these kinds of applications, it is important to identify and understand the overall problem. More specifically, it is important to define *Web applications* and *scalability*.

Basic Definitions

In this book, *Web application* has a very general definition—client/server software that is connected by Internet technologies to route the data it processes. By “Internet technologies,” I mean the collection of hardware and software that comprises the network infrastructure between consumers and providers of information. Web applications can be made accessible *by specific client software or by one or more related Web pages that are logically grouped for a specific productive purpose*. That purpose can be one of any number of things, for example, to buy books, to process stock orders, or to simply exist as content to be read by the consumer.

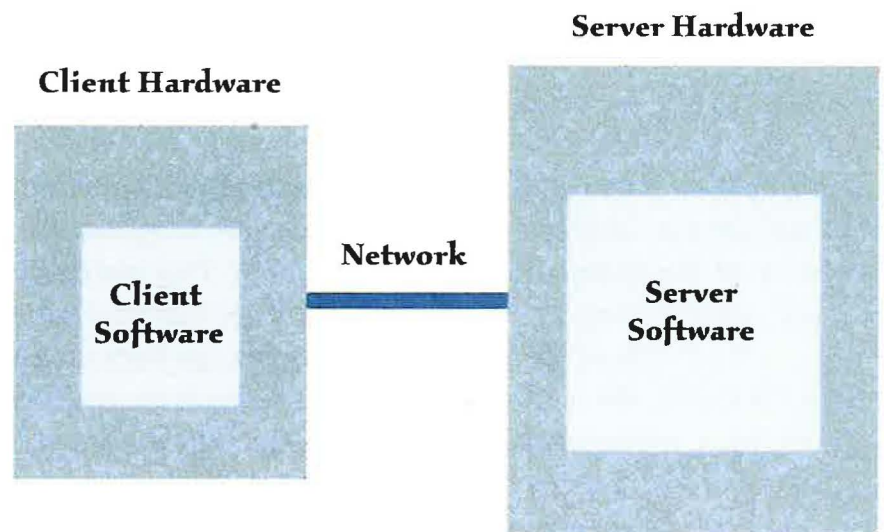
Notice that our discussion is about Web applications, not just “Web sites.” In truth, the difference between the two is essential to understanding one of the key themes of this book. Most nonengineers do not make a distinction between a Web site and a Web application. Regardless of the term, it’s the thing that allows them to buy their books online, to make plane reservations, to purchase tickets, and so forth.

If you’re an engineer, however, there is a difference. For you, it’s likely that when someone talks about, say, the performance of a Web site, you start thinking of back-end details. And so do I. Your mind begins to consider if it’s running an Apache or IIS and whether it works using Java servlets, PHP, or CGI-bin Perl scripts. This difference in thinking between engineers and nonengineers could be confusing. Engineers, by habit, tend to associate “Web site” with the server side. However, as we all know, there is more to a Web application than just the server side; there’s the network and the client. So, based on just that, a Web site (server) is not the same thing as a Web application (the client, network, and server).

While this book emphasizes server-side solutions, it is also concerned with client-side and networking topics because they have a fundamental impact on how end users perceive Web applications. That is, we will be concerned with the *end-to-end* interaction with a Web site, which simply means from *client to server and back to client*. This is a reasonable focus. After all, most people who use the Web are concerned with its end-to-end behavior. If it takes them a while to buy concert tickets online, it doesn’t matter if the problem is caused by a slow modem, an overtaxed server, or network congestion. Whatever the reason(s), the effect is the same—a slow application that’s eating up time. As engineers, we are concerned not only that such applications might be slow for one user, but also that the system becomes slower as more users access it.

Now that we have a better fix on the scope of a Web application, let us review its core components. These are the major pieces of any online application and each represents an opportunity—a problem or a challenge, depending on how you look at it. Although you’re probably familiar with the components, it doesn’t hurt to make sure everyone is on the same page, especially since these terms appear throughout the book. Let’s start with the client side.

Figure 1–1
Client, network, and server



We will say that Web applications are used by consumers via client software (i.e., Web browsers or applications that use the Web to retrieve or process data) running on client hardware (i.e., PCs, PDAs). Application data is provided and processing is handled by producers via server software (i.e., Web server, server-side component software, database) running on server hardware (i.e., high-end multiprocessor systems, clusters, etc.). Connecting the client to the server (from the modem or network port on the client device to the networking equipment on the server side) is the *networking infrastructure*. Figure 1–1 shows the client/server relationship graphically. Notice that the server side is bigger; in general, we assume that the server side has more resources at its disposal.

At this point, it is important to distinguish one piece of server software, the Web server, because it nearly always plays a central role in brokering communication (HTTP traffic) between client and server. In this book, when I refer to the “server side,” I am nearly always including the Web server. When it is necessary to distinguish it from the rest of the software on the server side, I will do so explicitly.

The Nature of the Web and Its Challenges

Although Web applications have rapidly made the Internet a productive medium, the nature of the Internet poses many engineering puzzles. Even the most basic of challenges—engineering how a provider can quickly and reliably deliver information to all who want it—is neither simple nor well understood. Like other challenges, this problem’s complexity has to do with the nature of the medium. The Internet is

different from the information-sharing paradigms of radio, television, and newspapers for several reasons. Perhaps two of the most important reasons are its incredibly wide audience (unpredictable number of customers) and the potential at any time for that wide audience to request information from any given provider (unpredictable work demands).

Unlike in other media, Internet information providers simply do not have the ability to know their audience in advance. Newspapers, for example, know their circulation before they print each edition. They also have the advantage of being able to control their growth, making sure they have enough employees to deliver the paper daily, and have the resources and time to go from deadline on the previous night to delivery on the next morning. Furthermore, newspapers do not have to deal with sudden jumps in circulation. Compared to the Internet, the growth of big newspapers in metropolitan areas seems far more gradual. For example, when the *Washington Post* was founded in 1877, it had a circulation of 10,000. By 1998, that circulation had reached nearly 800,000 for its daily edition and more than that for its Sunday edition.* That's an average growth rate of just over 6,500 subscribers per year, or 17 per day.

Deployers of Web applications have a love/hate relationship with their growth rates. In one sense, they would love the gradual growth of 17 new users per day. How nice life would be if you had to worry about scaling at that rate! You could finally go home at 5 P.M., not 9:30 P.M. At the same time, such growth rates are the reason that people are so excited about Web applications—because you can potentially reach the whole world in a matter of seconds. Your growth rate out of the gate could be hundreds of thousands of users. Although this bodes well for the business side of the things, it creates a tremendous challenge in terms of dealing with such demands.

On the Internet, circulation is akin to *page hits*, that is, the number of requests for a given document. Page hits can jump wildly overnight. A favorite example in the Web-caching community is the popularity of the online distribution of the Starr report. As most Americans know, this report was put together by the Office of the Independent Counsel during the Clinton administration. Let us just say that, while it was not flattering by any means, it was eagerly awaited by both the American public and the international press corps.

When the Starr report was released online in the summer of 1998 at government Web sites, tens of thousands of people tried to download it. A representative for Sprint, Inc., one of the Internet's backbone providers, reported a surge in bandwidth demand that ranged between 10 and 20 percent above normal; a representative of AOL reported an "immediate 30 percent spike"; and NetRatings, a Nielsen-like Internet content popularity company, estimated that at one point, more than

*Source: <http://www.thewashingtonpost.com>.

one in five Web users was requesting the report or news about it. CNET.COM ran a number of stories about the event and its ramifications for Internet scalability in the Fall of 1998.*

The conclusion among network administrators and engineers was universal. There were simply too many requests to be handled at once, and the distribution mechanisms were unable to scale to demand. It was a real test of the scalability of the Internet itself. Not only were the Web servers that provided this information overloaded, but the networking infrastructure connecting consumers to providers became heavily congested and severely inefficient. The effect was much like that of a traffic jam on a freeway.

This phenomenon was unique because it demonstrated the effects of sudden popularity as well as the short-lived nature of that popularity. For example, it is unlikely that you or anyone else remembers the URL(s) where the report was first available. And it is unlikely that you have it bookmarked. Thus, even had those sites been able to accommodate the demands of the time by buying bigger and faster machines, it would likely have been money wasted because the need for those resources dropped dramatically after the public lost interest in the report.

Other media, such as radio and television, are broadcast and do not need to worry about the size of their audience affecting their ability to deliver information. Consider television or radio programs, such as the national and local news. Their programmers know in advance when they are scheduled to broadcast. They have the luxury of being able to prepare ahead of time. Even when live radio or television beckons, the fact that both media are broadcast means that there is only one audience to address. Cable companies and good old TV antennae are already in place to facilitate the transport of that information. If we all watch or listen to the same channel, we all see or hear the same program. This is not the case with Internet audiences, where it is usually impossible to prepare for every request, where every consumer of information requires a unique response, and where there is a continual need for new virtual links (HTTP connections) between consumer and provider to be both created and then destroyed.

Performance and Scalability

Have you ever gone to a Web site, clicked on a link, and really *waited* for a response? Of course you have; we all have. It's annoying and frustrating. Worst are those content-laden sites that are meant to be read like newspapers. You want to jump from link to link, but every time you click, you have to wait *seconds* (not milliseconds) for

*Source: <http://news.cnet.com/news/0-1005-204-332427.html>.

the page and the ads and the embedded applets to download. You almost begin to hate clicking on a link because you know you will have to wait. You've learned to classify this kind of site as *slow*.

Then there are sites that are suspiciously slow. In these cases, you have reason to believe that bazillions of people are trying to connect, and this mass, not the technology, is responsible for the slowness. Say you're ordering a book at a site that has just announced a 50%-off sale. Or suppose tickets for a really hot concert have just gone on sale. When you're able to connect, the site seems unresponsive. When it does respond, it crawls. You guess that the site is buckling under the demand caused by the event. You've learned to classify this kind of site as *not scalable*.

As users, we have learned what poor performance and scalability are because we have experienced them. As engineers, we would like to understand these faults better so that our own users don't experience them. Because that is the focus of this book, let's start our discussion of performance and scalability by defining our terms.

Performance

Performance can be described simply as the raw speed of your application in terms of a single user. How long does a single application-level operation take? How long does it take to search for a book? How long does it take to confirm an online registration once we click Confirm? How long does it take to check out and pay at an online store? Notice that some of these examples describe atomic operations and some don't. When describing performance, we have to be clear if we are talking about one application operation or an entire session.

Consider the user interaction required to buy an airline ticket in Figure 1-2: In this session, there are three application operations, each consisting of a roundtrip between client and server. The operations are listed in Table 1-1 with their code names.

When we are talking about the performance of an operation, such as selection, we are interested in the end-to-end time required to complete that operation. In other words, the clock starts ticking when the user clicks the button and stops ticking when the user sees the information delivered. Why all this focus on end-to-end performance?

Table 1-1: Application Operations

<i>Code Name</i>	<i>User Action</i>	<i>Server Action</i>
Search	Criteria specified	Search based on criteria
Selection	Flight chosen	Confirmation for that flight generated
Confirmation	Flight confirmed	Confirmation processed



The J2EE Specification

Sun Microsystem's Java 2 Platform, Enterprise Edition (J2EE) specification is the proposed Java-based solution to the problem of building *n*-tier applications. J2EE focuses on defining client- and server-side technologies that make these applications not only easier to build but also easier to integrate. The J2EE spec encompasses all kinds of client/server types and interactions: It deals with Web clients, Web-based information servers, pure application servers, applets, and both synchronous and asynchronous solutions.

Although J2EE specifies many complex and powerful technologies, it is still just a specification. It requires vendors to actually develop what it proposes. Several have: BEA offers its WebLogic suite of products and IBM offers WebSphere, just to name the two most well-known. Sun has also released a reference implementation, which is a very useful and cost-effective way to get started with J2EE. Vendors differ on some important details, as we'll see later, but they must all implement the specification correctly or they won't be certified.

In this chapter, I provide a brief overview of J2EE and highlight the key component- and platform-level technologies it defines.

Overview of the Specification

The J2EE specification (version 1.3) describes a set of technologies designed to address the presentation, business logic, and persistent storage needs of *n*-tier applications. Generally speaking, a J2EE environment consists of several types of *components* that communicate with each other and a persistent storage device (a database). There are four categories of components:

- *Java applets*, which typically execute in a Web browser on a client machine
- *Java applications*, which execute on a local or remote client machine

- *Java servlets, JavaServer Pages (JSPs)*, filters, and Web event listeners, which execute on a server machine
- *Enterprise JavaBeans (EJBs)*, which are application objects that execute on a server machine

Each of these component categories is associated with a specific type of J2EE *container*:

- *Applet containers*, which host applet components
- *Application containers*, which host application components
- *Web containers*, which host servlets and JSPs
- *EJB containers*, which host EJBs

Containers manage their components. Component developers can assume that the containers exist and manage the components per a specified *contract* (agreement). In plain terms, this means that components must obey certain interface guidelines and that containers rely on interface-guaranteed functionality in the course of their management duties.

From the component perspective, a container is used to access the rest of the J2EE services. Thus, the container acts as the layer (or API) that makes the rest of the system visible to the component being contained. Through this layer, components can access other components or resources, such as an external database.

Each container type is targeted toward a particular style of deployment:

- Applet containers are the most limited. They simply act as proxies to the other types of container. Applets run in a Web browser and can contact Web containers in order to access the underlying business logic.
- Application containers run as standalone programs and can access Web containers, EJB containers, or the database.
- Web containers receive requests and format replies for Web clients. They enable a J2EE system to be accessed via HTTP.
- EJB containers represent the heart of the J2EE architecture: They manage the underlying business logic of the application. This is where operations like order processing occur or new accounts are created. EJBs are constructed based on the language of your business. They consist of a number of built-in features to enable reliability, availability, security, scalability, and robustness.

Figure 3–1 shows the general relationships of the different types of containers. The purpose of each of four J2EE containers is to permit access to underlying J2EE services (local or remote) through a set of common APIs. These APIs allow the containers to do things such as conduct transactions, manage security, and pool resources. The APIs and their basic purposes are listed in Table 3–1. Figure 3–2, a more detailed version of Figure 3–1, indicates where the various APIs and services fit into the different J2EE containers.

There is considerable opportunity for J2EE containers to interoperate with technologies that live outside a J2EE environment. For example, a Web container can use the HTTP, HTTPS, or IIOP protocols for communicating with other objects and systems. Table 3–2 summarizes the basic interoperability capabilities of the four types of containers.

There are many details of the J2EE spec that we could cover at this point. Rather than discuss all of them, we will focus on those that will play a role later in our discussions about scalability and performance.

Figure 3–1
Relationships
among J2EE
containers

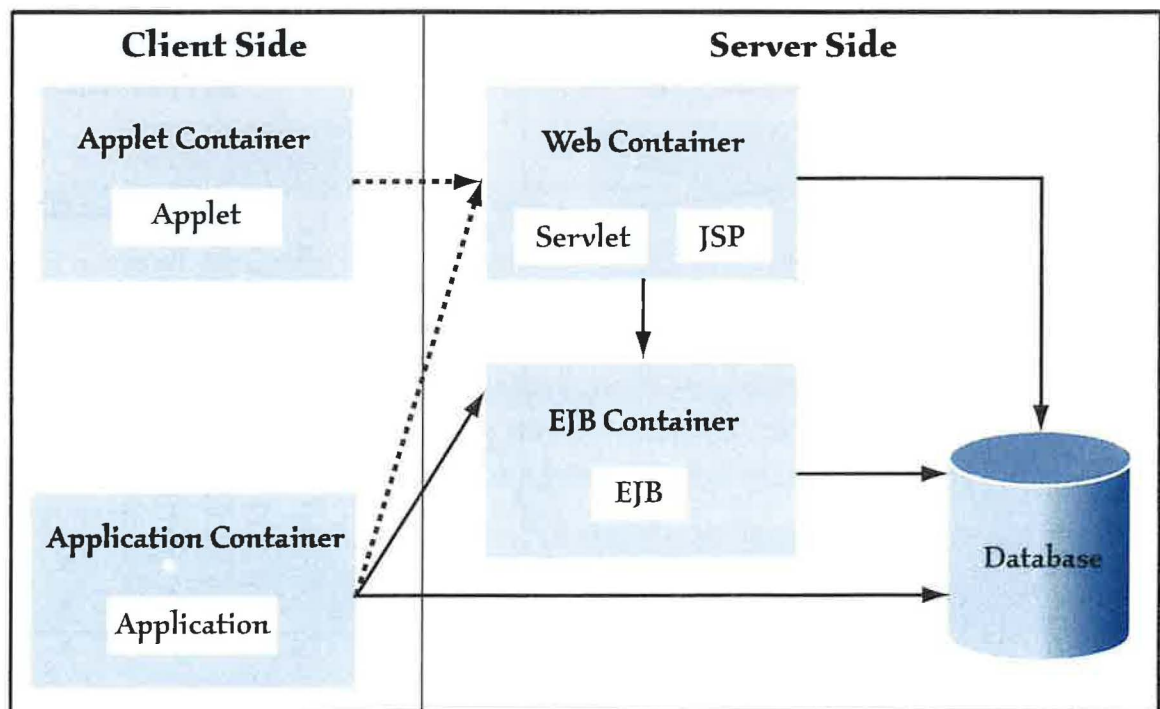


Table 3-1: J2EE Services and APIs

Service	Purpose
HTTP	Message-based Web communication
HTTPS	Message-based secure Web communication protocol
RMI-IIOP	RMI accessibility for CORBA objects
Java Database Connectivity	Database management
Java Naming and Directory Interface	Resource identification and state management
Java Message Service	Asynchronous messaging between containers
Java Interface Definition Language	Access to CORBA objects outside of a J2EE deployment
JavaMail	Notification via e-mail
JavaBeans Application Framework	(Required by Java Mail)
Java Transaction API	Transaction management
Java API for XML Parsing	Parsing XML
Java Connectors	Container access to Enterprise Information Systems
Java Authentication and Authorization	Security within J2EE containers

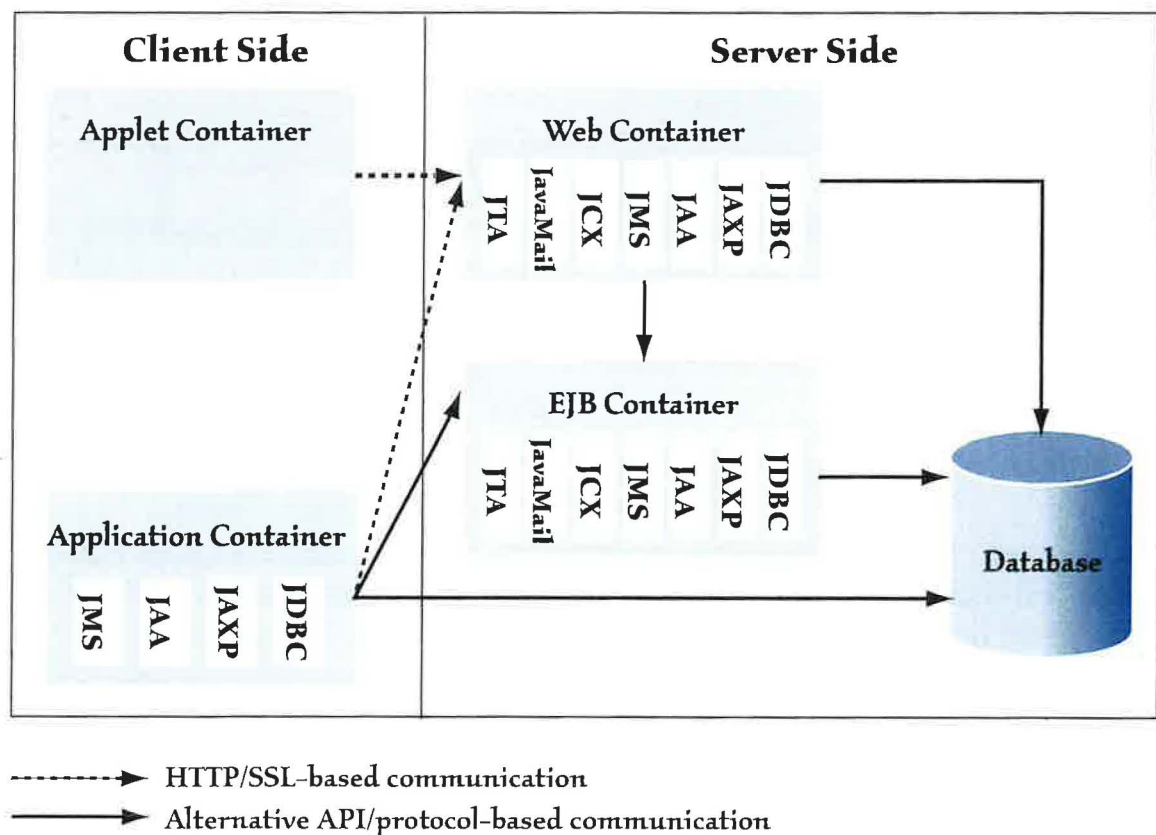
Figure 3-2
APIs
supported
by J2EE
containers

Table 3–2: J2EE Container Interoperability

<i>Container Type</i>	<i>Inbound Protocols</i>	<i>Outbound Protocols</i>
Application		HTTP, SSL, IIOP, JRMP
Applet		HTTP, SSL, IIOP, JRMP
Web	HTTP, SSL	HTTP, SSL, IIOP, JRMP
EJB	EJB, IIOP, SSL	HTTP, SSL, IIOP, JRMP

Deployment Issues

In this book, we will not be covering things like how to run a certain implementation of the J2EE or the exact details of variations of J2EE across vendors. In fact, we'll generally ignore detailed deployment issues, because they tend to be highly vendor specific and relate more to J2EE operation than architecture and design. Nevertheless, it is useful at this point to say a few things about how a J2EE application is packaged and deployed. Later, when I refer to something as being a deployment issue, it is often related to one or more of the aspects that I'm going to cover here.

Packaging

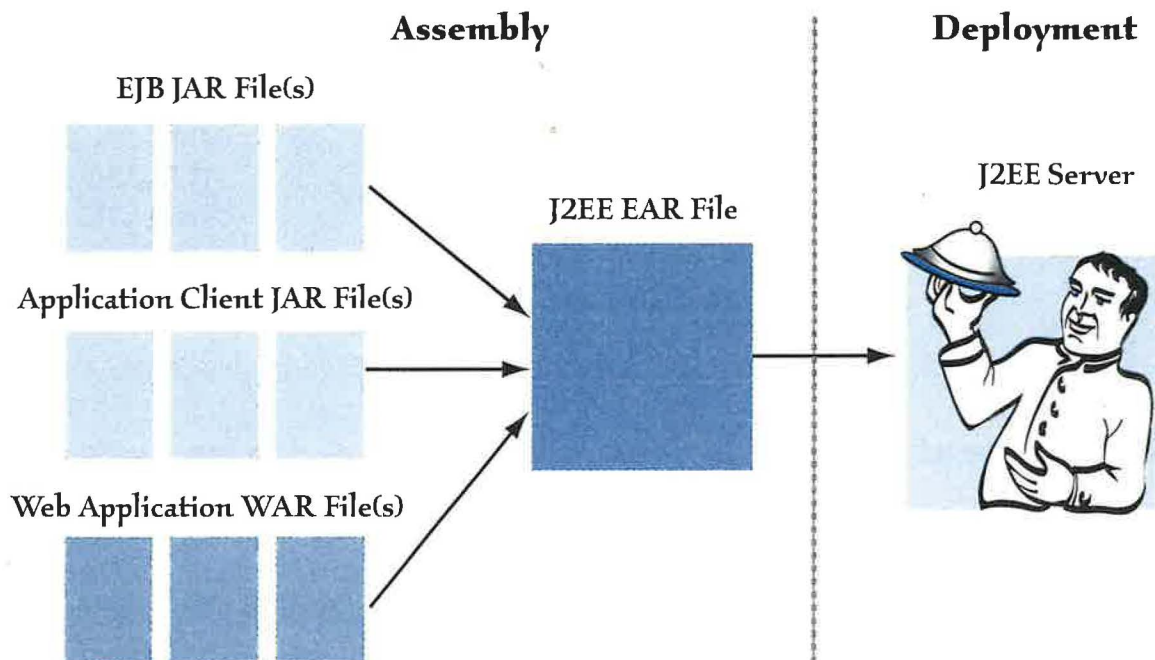
In terms of packaging, a J2EE application is stored in a file with an `.ear` extension (hereafter called an EAR file). An EAR file is the same as a `.jar` (JAR) file, except that it refers to an enterprise Java application archive, not just to a Java application archive. Another important type of file is the Web application archive, or `.war` (WAR) file, which archives files related to servlets and their JSP pages.

An EAR file can contain one or more of the following:

- JAR files for each EJB component
- JAR files for each application client (if any)
- WAR files for each servlet and set of related JSPs

The EJB and application client JARs contain the relevant `.class` files, any related files, and something called a deployment descriptor (more about this in a moment). The WAR files contain servlet `.class` files, static Web resources such as GIF files, a set of related files (i.e., utility Java classes), and a deployment descriptor file.

Figure 3-3
J2EE service
packaging and
deployment



To get a feel for what an EJB JAR file looks like, consider a JAR file for a sample EJB called **BenefitSession** (this EJB will appear in a later example). Here are its contents:

```
BenefitSession/
    EjbBenefitSessionbean.class
    EjbBenefitSessionRemote.class
    EjbBenefitSessionHome.class

META-INF/
    ejb-jar.xml
```

The relationship between what is assembled and what is deployed is shown in Figure 3-3. As you can see, WAR and JAR files are the guts of a J2EE application EAR file, which is deployed as a J2EE server.

Deployment Descriptor Files

A deployment descriptor file is simply an XML file that typically contains structural, assembly, and runtime information about J2EE components. A J2EE application consists of a deployment descriptor for the application as a whole and specific deployment descriptor files for each component as necessary. The information in a descriptor file has to do with the component assembly (e.g., the names of EJB interface

classes), security information (e.g., definitions and applications of roles), and other runtime dependencies and information.

Part of a sample EJB deployment descriptor file is shown here. This snippet indicates something about the structure of the object (i.e., its attributes and relationships) and its resource dependencies (i.e., database information):

```
<persistence-type>Container</persistence-type>

<cmp-field><field-name>first_name</field-name></cmp-field>
<cmp-field><field-name>last_name</field-name></cmp-field>
<cmp-field><field-name>hire_date</field-name></cmp-field>

<resource-ref>
  <res-ref-name>employee</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Without going into the details of what a deployment descriptor is made of, we can observe that it is simply an XML file that contains metadata about the EJB or the application component being deployed. As an XML file, the deployment descriptor helps to ensure the portability of deployment—independent of the deployment platform, the same set of properties can be used. The XML file is, after all, just a text file that is independent of the operating system and CPU. If you are new to XML, you may find it helpful to skip ahead to the first part of Chapter 12.

To illustrate how deployment descriptors can contain information about security roles and privileges, consider the following example.

Listing 3–1: Sample deployment descriptor

```
<ejb-jar>
  ...
  <enterprise-beans>
    ...
    <entity>
      <ejb-name>MyBean</ejb-name>
      <ejb-class>MyEntityBean.class</ejb-class>
      ...
      <security-role-ref>
        <role-name>entity-admin</role-name>
        <role-link>administrator</role-link>
      </security-role-ref>
      ...
    </entity>
    ...
  </enterprise-beans>
</ejb-jar>
```

```

</enterprise-beans>
...
<assembly-descriptor>
  <security-role>
    <description>The Admin Role</description>
    <role-name>administrator</role-name>
  </security-role>
</assembly-descriptor>
...
</ejb-jar>

```

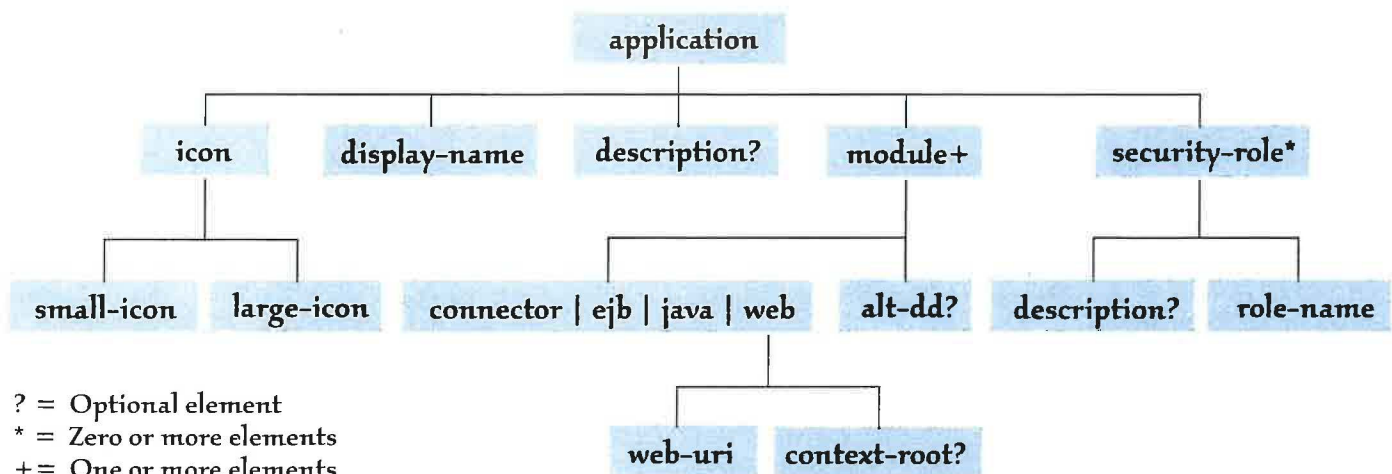
Notice that new security roles can be both defined and linked, as well as applied to various application components such as EJBs.

The entire XML DTD for J2EE deployment descriptors can be found at http://java.sun.com/dtd/application_1_3.dtd. Taken almost directly from the J2EE specification, Figure 3–4 is a graphical summary of this DTD, which may help you visualize how a J2EE application breaks down and organizes its assembly and deployment information.

As the figure shows, a deployment descriptor contains information about all types of application information, including its modules. Note that this DTD is relevant for the entire application (i.e., the EAR file). Individual elements of the application, such as the EJB JAR file and the WAR file, have their own deployment descriptor DTDs.

Again, we won't consider the detailed modifications of the deployment descriptor file for each component. Books devoted to specific J2EE technologies like JMS

Figure 3–4 J2EE deployment descriptor XML DTD



and EJBs already offer such information. They are often the best resource because there are so many possible deployment descriptor settings, many of which are independent. The J2EE vendor documentation is also a good source; it will provide the most comprehensive information about deployments. Finally, continual comments about deployment descriptor settings would be tedious and a distraction. We are occasionally interested in what *kind* of information is stored in a file for a given component—not so much in how it is actually specified.

Platform Technologies and Services

One of the benefits of building your application on top of the J2EE infrastructure is that doing so allows you to inherit key technologies that facilitate scalable deployment. In this section, we visit some of the key platform technologies and services J2EE infrastructures provide. These technologies are related to simplifying and empowering distributed application components to coordinate and communicate. Providing this type of transparency in a distributed system allows application designers to focus on the business logic of their applications, rather than waste their time tuning the replication and distribution of components that implement that logic.

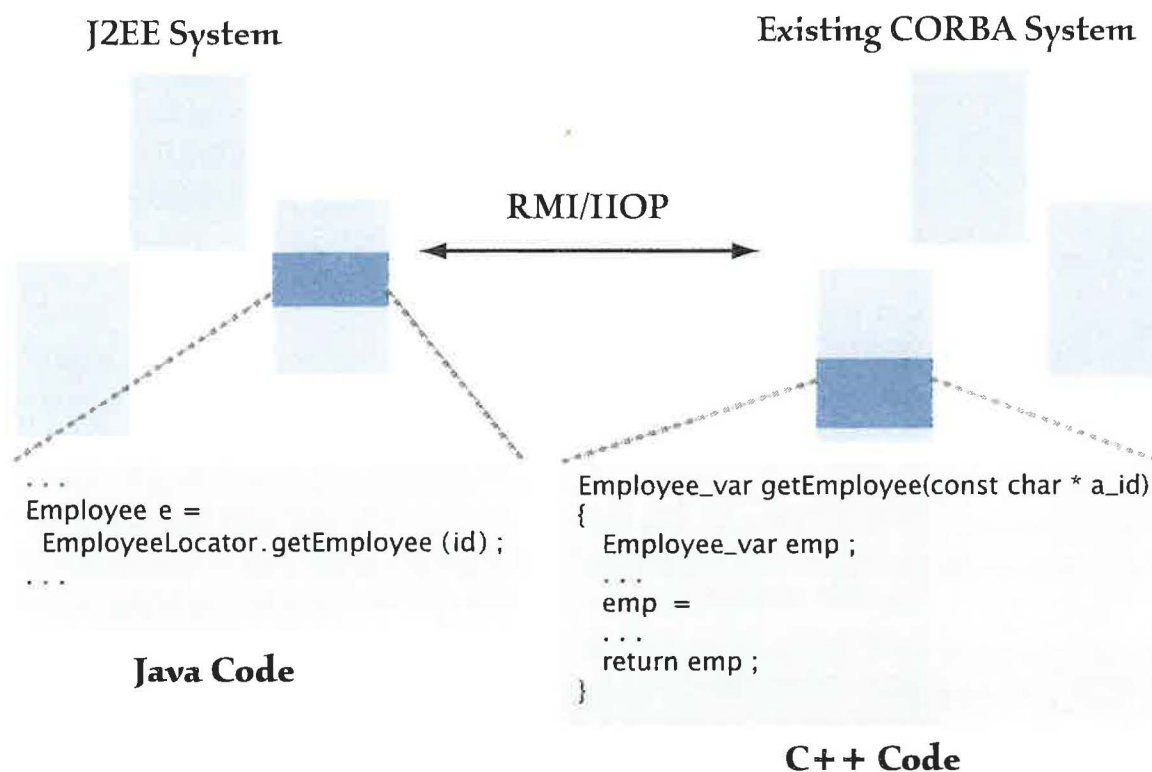
Component Communication via RMI-IIOP

A large part of J2EE centers around the idea of making sure that business logic is accessible from many types of clients and from other middleware standards, particularly the **Common Object Request Broker Architecture (CORBA)**. CORBA and the Internet Inter-ORB Protocol (IIOP) specify how distributed objects can be deployed and can communicate across network boundaries.

CORBA has its own huge specification, which consists of many core services (e.g., naming) and the Interface Definition Language (IDL). IDL is a platform-neutral language for specifying services. Its compilers generate “stub” and “skeleton” code in languages such as C++ and COBOL, which allow the functionality in those languages to be deployed as accessible (i.e., CORBA objects). Using the generated stub/skeleton code, objects can communicate with each other (more about this in Chapter 8).

An **object request broker (ORB)** is the mechanism by which objects locate and bind to each other. IIOP is simply a protocol for CORBA objects to communicate over a network. If you’re somewhat familiar with RMI (Remote Method Invocation) and EJB, but new to CORBA, you’re perhaps beginning to sense a similarity.

Figure 3-5
Integrating
Java and C++
CORBA
objects



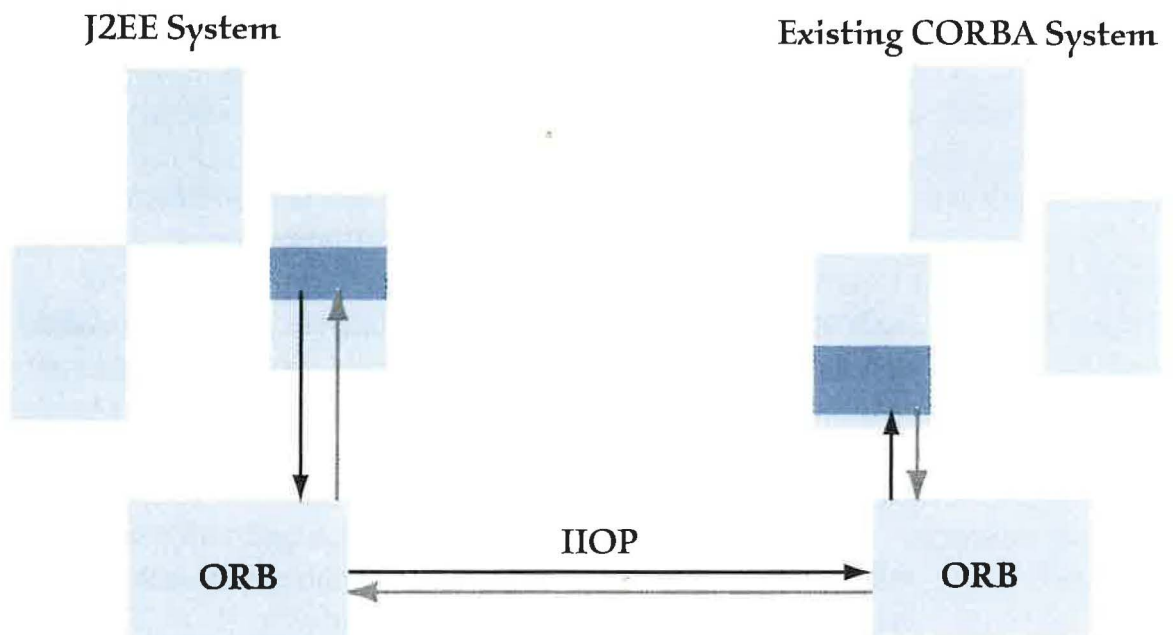
In the Java 2 Standard Edition (J2SE) version 1.3, RMI was fused with IIOP to create RMI-IIOP. Although it uses IIOP as its means of communication, RMI-IIOP retains an RMI-style API. This enables IIOP-style communication without forcing Java developers to learn yet another API.

For a J2EE system, RMI-IIOP is the primary method of communication across tiers. In addition, it essentially allows Java developers to construct CORBA services without having to learn IDL. In the old days, you had to buy a CORBA product produced by companies such as Visigenic or IONA. Now J2SE comes with a fully functional CORBA ORB and support for stub code generation through its RMI compiler.

What all this means for the application system architect is that Java developers can easily access existing CORBA application systems and vice versa. Thus, it allows legacy functionality to be incorporated easily in a J2EE deployment.* Figure 3-5 shows Java objects communicating with remote CORBA objects over RMI-IIOP, and Figure 3-6 shows how IIOP works. Notice that it's simply a layer through which objects associated with different ORBs can be coupled.

*An important restriction, which we don't address in detail here, has to do with legacy CORBA objects, which must be deployed by vendors who support the recent Objects-by-Value extension of the CORBA specification. More about this at <http://cgi.omg.org/cgi-bin/doc?formal/99-10-07>.

Figure 3–6
IIOP among
ORBs



Figures 3–5 and 3–6 show a J2EE system talking to an existing CORBA System. I do this to show why you would connect CORBA objects written in two languages—in this case, something new (a J2EE application system written using Java)—to something that already exists (a CORBA application system written in, say, C++ or Smalltalk and deployed using a different ORB vendor). In reality, however, these are two CORBA systems. J2EE and RMI-IIOP allow Java programmers to represent their services as CORBA objects, without the additional work of coding an IDL file, and so on.

The main message here is that in addition to allowing you to use RMI to communicate easily with existing Java-based RMI objects, RMI-IIOP allows you to extend object sets to include CORBA objects. Since RMI-IIOP is the language spoken between components in J2EE containers, it's important to be aware of what it is, and how it generally works.

Transaction Management Using the Java Transaction API

A J2EE transaction enables a series of operations, perhaps distributed, to be executed in an orderly way that guards against errors with any one of them. It also protects the integrity of these operations across program boundaries.

For example, suppose your server-side application system is an automatic conference organizer that allows one-touch registration. A user enters her name, address, conference options (i.e., tutorials she wants to enroll in), and travel options. She

provides a credit card account number and the system does the rest. The conference organizer, in turn, does the following:

- Register participant for conference
- Register participant for hotel
- Register participant for rental car
- Debit user's credit card for the total amount

Each operation may be handled by a different application server. There's no telling what can happen with any one operation; for example, the hotel registration may fail or return an error. If there are errors, we certainly don't want to continue the series of operations and we want to undo actions that occurred in earlier operations. This is just the kind of thing transactions are good at.

With transactions, there are two important terms to remember: **commit** and **rollback**. When one operation is completed, it is still not finalized. Instead, all such operations wait until the last one has completed. At that point, the transaction can explicitly issue a commit operation, which finalizes all of them. If an error or failure occurred during the execution of any of these operations, a rollback can be issued to signal to the other operations that finalization will never occur and that uncommitted actions should be undone.

The Java Transaction API (JTA) supports transaction management among various J2EE containers. The operations that take place during a transaction are typically JDBC or JMS related, although the J2EE spec also supports external adapters that conform to the XA transaction standard. Note that the `javax.transaction` package specifies a very general form of transactions that has nothing directly to do with using JDBC or JMS in isolation—it merely ties together operations for a logical operation, including those (e.g., database updates) that involve external resources.

The J2EE platform offers transaction services because of the complexity and possibility of errors when developers code their own transaction logic, especially between tiers. In fact, a single client interaction might cause many components in many tiers to be invoked and create confusion about who starts the transaction and who is responsible for committing or rolling back that transaction. By offering transaction services at the platform level, J2EE allows a collection of distributed components to easily coordinate execution common to a single, logical client operation.

Transactions and Web Components

When designing Web components, such as Java servlets, you can always get a handle to the current transaction. This is guaranteed by the specification and is a responsibility of the platform. In particular, the spec states that Web components can obtain

access to a transaction that implements the `javax.transaction.UserTransaction` interface. Web components are allowed this access via the JNDI API, which we discuss in more detail shortly.

Transactions in Web components don't span client sessions. More specifically, for servlets, they are started in the servlet's `service()` method and must be completed (committed or rolled back) before this method returns. If not, the Web component is in an error state.

Web components can consist of multiple threads, but transaction resource objects should not be shared by threads. This means that threads should not use associate transaction resources with common memory addresses. Thus, resource objects should not be stored in static variables or in instance variables.

Since servlets often interact with EJBs, which interact with the database, it is important to know how a transaction propagates. The specification declares that when a Web component contacts an EJB, the platform must be responsible for propagating that transaction to the EJB's context.

Transactions and EJBs

Transaction management gets a bit more complicated when it comes to EJBs, partially because J2EE assumes that the bulk of application logic resides in EJBs. It's also because management of persistent data in the application is abstracted by EJBs, specifically in the form of entity beans. In contrast, session and message-driven beans (the other two types of EJBs) represent session logic, not data.

It's useless to speak about the details of EJBs and transactions here, since we haven't discussed EJBs in detail. For now, the important things to remember about EJBs and transactions are:

- For some types of bean (session and message-driven beans), transaction management may be handled by the bean itself; that is, the developer writes the logic to start, commit, and possibly roll back a transaction.
- Session and message-driven beans can be configured so that transaction management is automatic—by the container. In fact, details about their transaction management (e.g., does a transaction get passed from one bean to another when the first calls the second?) can be defined per method.
- In contrast to session and message-driven beans, entity beans can't manage transactions explicitly. Everything is handled by the container, regardless of whether that bean interacts with the database (bean-managed persistence) or the container does (container-managed persistence).

Many of these details, especially the types of EJBs that can be developed, are covered in Chapter 8.

JNDI for Resource Location

Although J2EE supposes a distributed application system, it's often necessary to maintain a common state among the system's components. For example, resources, names, or addresses may be common among components (i.e., location independent). Access and management of this state are provided by the Java Name and Directory Interface (JNDI).

For those familiar with directory services, JNDI is the means with which to access such services as LDAP. For most other J2EE developers, JNDI is simply an easy way to manage resources shared by tiers.

From a component standpoint, JNDI gives location-independent access to a common application state, or **context**. The J2EE specification also views the JNDI context as a mechanism for making application components dynamic. Instead of altering the source code for a component, it bases a conditional part of component execution on the values obtained by access to the context. Then, at runtime, when the component accesses its context, such variable settings (i.e., variable values or rules) are resolved. This can be done on a deployment basis. That is, the deployer can set the proper value in the deployment descriptor, and the component can obtain that value via the JNDI. All of this occurs without recompilation.

For example, suppose we wanted to customize the multiple to use as the basis for salary raises. If we assume that this value is deployment specific or otherwise dynamic, then our application component needs to read this value from the JNDI and act appropriately.

The following snippet of application code does just that for a mythical function called `calculateNewSalary()`. The method shown requires the prior salary value as a parameter and uses the value of `raiseMultiple` to determine the new salary.

```
public double calculateNewSalary(double a_origSalary) {  
  
    /* Obtain handle to our naming context. */  
    Context initCtx = new InitialContext();  
    Context localCtx = (Context)initCtx.lookup("java:comp/env");  
  
    /* Look up the salary multiple */  
    double raiseMultiple = ((Double)  
        localCtx.lookup("raise_multiple")).doubleValue();  
  
    /* Raise salary per the multiple */  
    return raiseMultiple * a_origSalary;  
  
}
```

Thus, we first initialize the context and locate the desired node within the context tree structure. Once found, we read one leaf of that directory tree—`raise_multiple`—for its value. Thus, for example, a 10 percent raise means that this value is 1.10. We then use that value as the basis for calculating the new salary.

In the deployment descriptor, `raise_multiple` might be stored as follows:

```
...
<env-entry>
  <env-entry-name>raise_multiple</env-entry-name>
  <env-entry-type>java.lang.Double</env-entry-type>
  <env-entry-value>1.10</env-entry-value>
</env-entry>
...
```

Another use of the JNDI is to provide access to various shared J2EE resources, such as the following:

- Databases
- Message queues
- EJBs
- Transactions
- Mail servers
- URLs

Using the JNDI to locate these objects provides a simple and consistent interface for obtaining access to a common resource.

As an example of how these other objects can be fetched from the context via the JNDI, consider how a well-known Java Message Service (JMS) message queue is located. Although its name is known (i.e., the queue was registered to JNDI under that name), its location is not. This is the value of the JNDI: Instead of forcing every application component to know where every resource is physically located, J2EE provides a logical handle to that distributed resource. This makes coding application components simpler and allows the component to be relocated without having to recode or recompile the application components that use that resource.

The following code is an example of how to obtain a handle to a well-known message queue using the JNDI, the following code might suffice:

```
/* Obtain handle to our naming context. */
Context initCtx = new InitialContext();
Context localCtx = (Context)initCtx.lookup("java:comp/env");
```

```
/* Look up the message queue we want */  
Queue myQueue = (Queue)localCtx.lookup("jms/MyQueue");
```

Just as the `raise_multiple` value was defined in the deployment descriptor, so too is the information corresponding to the JMS `MyQueue` object. Notice that in both cases, the resource type and description can be included in the descriptor entry, thus making the environment self-describing.

```
<resource-env-ref>  
  <description>  
    My message queue  
  </description>  
  <resource-env-ref-name>jms/MyQueue</resource-env-ref-name>  
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>  
</resource-env-ref>
```

J2EE and Your Architecture

At this point, you may be wondering, do I need to use all of J2EE? If not, how much of its technology should I use? The answer is, *as much as you want*.

To be sure, most Java development efforts already use popular parts of the J2EE architecture, such as servlets, JSPs, and EJBs. JDBC has already become a popular mechanism for integrating databases into Java applications. More recently, JMS is gaining converts as people (re)discover the joys of asynchronous communication. However, one of the nice things about the J2EE model is that you don't have to embrace all of it to use some of it. For example, you don't have to use EJBs; you can simply use servlets and JSPs. Or, you can use servlets and EJBs, but not JSPs. With the exception of the platform technologies, which tie everything together, the component technology you use depends on your needs.

When you do use J2EE, however, it means choosing a vendor. Again, the J2EE is merely a specification. Unless you are planning to implement it yourself, you should settle on a vendor that is not only J2EE compliant but offers an implementation that suits your needs. For example, as we will discuss in Chapter 8, vendors differ on how they implement container-managed persistence (automatic data management) for EJB entity beans. This is the kind of detail you should know before choosing a vendor.

Finally, regardless of your path, if you're interested in building highly scalable Web applications, you should strongly consider an *n*-tier approach. In building those tiers, you may find that part or all of the J2EE specification fits your needs and plans.

Summary

In this chapter, we learned that the J2EE specification describes how *n*-tier applications can be built and deployed. Although these applications are often thought of as Web applications, they don't need to be. In fact, Web integration is only one facet of the specification.

J2EE is a set of application components, containers to manage those components, and integration or platform “glue” to tie everything together in a secure, location-independent manner. Component management is enforced via interface contracts. Simply put, a container has guarantees that the component has certain methods (i.e., lifecycle methods), so that it can be managed as necessary.

We covered the key services provided by the J2EE platform and gave special attention to transactions and resource management. From the component perspective, the container provides APIs that not only allow access to these technologies, but also enable other J2EE functionality—such as Java Mail—to be easily accessed.

It's unlikely that you will use every J2EE feature in a single application. For example, your application may be such that messaging services, such as that provided by JMS, do not make sense. Still, I strongly suggest that if you plan on deploying any application system for the Web, you consider J2EE. It is portable and is integrated with a number of Web/non-Web, asynchronous/synchronous technologies that give a great flexibility and freedom to the architect designing an application solution. For our purposes, J2EE provides a great example of how some scalability and performance techniques (e.g., caching and redundancy) have crept into a practical and enterprise-level application system.