

Expert One-on-One J2EE™ Design and Development

Rod Johnson



Expert One-on-One J2EE™ Design and Development

Rod Johnson



Expert One-on-One J2EE™ Design and Development

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2003 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

Library of Congress Card Number: 2003107067

ISBN: 0-7645-4385-7

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

1B/RQ/QW/QT/IN

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, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8700. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-Mail: permcoordinator@wiley.com.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: WHILE THE PUBLISHER AND AUTHOR HAVE USED THEIR BEST EFFORTS IN PREPARING THIS BOOK, THEY MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK AND SPECIFICALLY DISCLAIM ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES OR WRITTEN SALES MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR YOUR SITUATION. YOU SHOULD CONSULT WITH A PROFESSIONAL WHERE APPROPRIATE. NEITHER THE PUBLISHER NOR AUTHOR SHALL BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (800) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Trademarks: Wiley, the Wiley Publishing logo, Wrox, the Wrox logo, the Wrox Programmer to Programmer logo and related trade dress are trademarks or registered trademarks of Wiley in the United States and other countries, and may not be used without written permission. J2EE is a trademark of Sun Microsystems. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Trademark Acknowledgments

Wrox has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Wrox cannot guarantee the accuracy of this information.

Credits

Author

Rod Johnson

Commissioning Editor

Craig A. Berry

Technical Editors

Kalpana Garde

Niranjan Jahagirdar

Project Managers

Cilmara Lion

Abbas Rangwala

Author Agent

Nicola Phillips

Index

Adrian Axinte

Andrew Criddle

John Collin

Proof Reader

Chris Smith

Technical Reviewers

Simon Brown

John Carnell

Eric Eicke

Meeraj Kunnumpurath

Todd Lauinger

Eric Paul Schley

Andrew Smith

Tobin Titus

Tom Watkins

David Whitney

Dave Writz

Production Coordinators

Abbie Forletta

Manjiri Karande

Illustrations

Santosh Haware

Manjiri Karande

Cover

Dawn Chellingworth

Natalie O'Donnell

About the Author

Rod Johnson is an enterprise Java architect specializing in scalable web applications. Rod spent two years designing and delivering a J2EE solution for FT.com, Europe's largest business portal, before trekking to Everest Base Camp, parenting a baby, and writing this book. He would like to thank Tristan for providing the toughest of these challenges.

Rod completed an arts degree majoring in music and computer science at the University of Sydney. He obtained a PhD in musicology before returning to software development. With a background in C and C++, Rod has worked with both Java and J2EE since their release. He is currently a member of JSR 154 Expert Group defining the Servlet 2.4 specification.

Rod has contributed several chapters to other Wrox publications including *Professional Java Server Programming* (J2EE and J2EE 1.3 editions) and *Professional Java Server Pages* (2nd edition), and is a reviewer for Wrox Press. He has presented at international conferences including Times Java, Mumbai (2001), and his writing has been featured on java.sun.com.

An Australian, Rod is currently living and working in London.

He can be contacted at expert@interface21.com.

Although all authors seem to thank their families, I had no idea how richly such gratitude was deserved until I wrote this book.

Above all, I would like to thank my wife, Kerry, for her love and support throughout a lengthy and stressful process.

Others to give generous support include my parents-in-law, Don and Dana Jamieson.

Of those who have given practical help, I would like to thank my editor, Craig Berry, for helping me to distill a huge topic into a book; and my former colleague, Dominic Fisher, for his business insight and valuable help in formulating realistic requirements for the sample application.



Table of Contents

Introduction	1
J2EE Myths	2
How is this Book Different?	5
My Approach	6
Who this Book is for	7
Aims of this Book	7
What this Book Covers	7
Assumed Knowledge	8
Recommended Reading	9
What You Need to Use this Book	9
Chapter 1: J2EE Architectures	15
Goals of an Enterprise Architecture	16
Deciding Whether to Use a Distributed Architecture	18
New Considerations in J2EE Design	19
When to Use EJB	20
Implications of Using EJB	20
Questionable Arguments for Using EJB	22
Compelling Arguments for Using EJB	22
Arguments for Using EJB to Consider on a Case-by-Case Basis	23
Accessing Data	24
J2EE Data Access Shibboleths	24
Entity Beans	25
Java Data Objects (JDO)	25
Other O/R Mapping Solutions	25
JDBC	26
State Management	26
J2EE Architectures	26
Common Concepts	27
Architectural Tiers in J2EE Applications	27
The Importance of Business Interfaces	28

Table of Contents

Non-distributed Architectures	28
Web Application with Business Component Interfaces	28
Web Application that Accesses Local EJBs	30
Distributed Architectures	32
Distributed Application with Remote EJBs	32
Web Application Exposing Web Services Interface	34
Web Tier Design	36
The Model View Controller (MVC) Architectural Pattern	36
Connectivity Between the Web Tier and Business Objects	38
Designing Applications for Portability	38
Summary	40
Chapter 2 J2EE Projects: Choices and Risks	43
Developing a Policy on Specification Versions	44
Choosing an Application Server	45
When to Choose an Application Server	46
Defining the Requirements	47
Evaluation Criteria	48
Supported Specifications	48
Sun Resources	48
Cost	50
Vendor Relationship	50
Vendor Viability	51
Development and Deployment	51
Value-added Features	52
Quality of Documentation	52
Availability of Skills	53
User Experience	53
Choice Process	54
Top Mistakes in Choosing an Application Server	54
The "Neat Technology" Trap	55
When to Use Alternative Technologies to Supplement J2EE	56
Portability Issues	56
What does Portability Mean?	57
A Pragmatic Approach to Portability	58
Staging Environments and Release Management	58
Building a Team	60
Team Structure	60
Who Owns the Architecture?	60
Vertical or Horizontal Responsibility	61

Choosing Development Tools	63
Visual Modeling Tools	64
IDEs	64
Build Utilities	64
Code Generators	66
Version Control	67
Identifying and Mitigating Risks	67
Summary	71
Chapter 3: Testing J2EE Applications	73
What Can Testing Achieve?	74
Definitions	75
Testing Correctness	76
The XP Approach to Testing	76
Writing Test Cases	78
What Makes a Good Test Case?	78
Recognizing Test Case Authoring and Maintenance as a Core Task	78
Unit Testing	78
main() Methods	79
Using JUnit	79
Test Practices	86
Should Testing Strategy Affect How We Write Code?	90
Integration and Acceptance Testing	92
Testing Business Objects	92
Testing Business Objects Implemented Without Using EJB	92
Testing EJBs	93
Testing Database Interaction	99
Testing Web Interfaces	100
Unit Testing Web-Tier Components	101
Acceptance Testing Web Interfaces	104
Design Implications	105
Testing Performance and Scalability	105
Load Testing EJBs and Other Business Objects	106
Load Testing Web Interfaces	107
Automating Tests	107
Complementary Approaches to Testing	108
Summary	110

Chapter 4: Design Techniques and Coding Standards for J2EE Projects 113

OO Design Recommendations for J2EE Applications 114

Achieving Loose Coupling with Interfaces	115
Prefer Object Composition to Concrete Inheritance	115
The Template Method Design Pattern	117
The Strategy Design Pattern	119
Using Callbacks to Achieve Extensibility	120
The Observer Design Pattern	122
Consider Consolidating Method Parameters	124
Exception Handling – Checked or Unchecked Exceptions	125
Good Exception Handling Practices	128
Exceptions in J2EE	130
Making Exceptions Informative	131
Using Reflection	132
Reflection Idioms	133
Using JavaBeans to Achieve Flexibility	138
Avoid a Proliferation of Singletons by Using an Application Registry	139
Refactoring	142

Coding Standards 142

Start from the Standard	143
Allocation of Responsibilities	145
Avoid Code Duplication	146
Avoid Literal Constants	147
Visibility and Scoping	149
Public Instance Variables	149
Protected and Package Protected Instance Variables	150
Method Visibility	151
Variable Scoping	151
Inner Classes and Interfaces	152
Using the final Keyword	153
Method Overriding and Final Methods	153
Final Classes	155
Final Instance Variables	155
Implementing toString() Methods Useful for Diagnostics	155
Defensive Coding Practices	156
Handle Nulls Correctly	156
Consider the Ordering of Object Comparisons	156
Use Short-circuit Evaluation	157
Distinguish Whitespace in Debug Statements and Error Messages	157
Prefer Arrays to Collections in Public Method Signatures	157
Documenting Code	158
Logging	160
Choosing a Logging API	162
Logging in the EJB Tier	165

Why (and How) Not to Reinvent the Wheel	166
Help! API Overload	167
Using Frameworks	167
What Makes a Good Framework?	168
Benefits of Using Existing Frameworks	168
Evaluating Existing Frameworks	169
Implementing your own Framework	170
Summary	171
 Chapter 5: Requirements for the Sample Application	 179
Overview	180
User Populations	180
Public Internet Users	180
Box Office Users	181
Administrators	182
Assumptions	182
Scope Limitations	183
Delivery Schedule	183
Internet User Interface	183
Basic Workflow	184
Error Handling	185
Application Screens	185
Welcome Screen	188
Display Show Screen	189
Book Seats Screen	191
Show Reservation Screen	192
Payment Details Screen	194
Confirm Reservation Screen	196
Box Office User Interface	198
Non-Functional Requirements	198
Hardware and Software Environment	199
Summary	200
 Chapter 6: Applying J2EE Technologies	 203
When is a Distributed Architecture Appropriate?	204
Distributed Applications and Scalability	205
Distributed Applications and Reliability	206
Scalable and Robust Architectures	207
High-level Architecture for the Sample Application	209

Table of Contents

Deciding When to Use EJB	209
Using EJB to Implement a Distributed Architecture	209
Transaction Management	210
Transaction Management in J2EE Applications	210
Transaction Management and EJB	211
Transaction Management in the Sample Application	213
EJB and Authorization	213
EJB and Multi-threading	214
Declarative Configuration Management	214
The Downside of EJB	214
So Much Infrastructure	214
Programming Restrictions Applying to EJBs	215
The Singleton Problem in EJB	220
Timer Functionality	222
EJBs in the Sample Application	222
Deciding How to Use EJB	223
What Should EJBs Do?	223
When to Use Local or Remote Interfaces	223
Does it Make Sense for a Bean to Have Both Local and Remote Interfaces?	224
Phony Remote Interfaces	225
EJB Interface Summary	226
Using EJBs in the Sample Application	227
Deciding when to Use Asynchronous Calling with JMS	228
Message-Oriented Middleware (MOM) and JMS	228
Producing Messages	228
Consuming Messages	229
Consuming Messages without Using EJB	229
Consuming Messages with Message-Driven Beans (MDB)	229
When to Use Asynchronous Calling	232
Indications for Using Messaging	232
Disadvantages of Using Messaging	233
JMS and Performance	233
Alternatives to JMS Messaging	234
JMS in the Sample Application	234
Authentication and Authorization	235
The Standard Security Infrastructure	235
The Server Implementation	237
Deciding When to Use XML	238
Using XSLT in J2EE Applications	238
“Deep” Use of XML	239
Converting Between JavaBeans and XML	241
J2EE and XML in the Future	244
XML in the Sample Application	245
Caching to Improve Performance	245
Caching Options	245
A Caching Strategy for the Sample Application	247
Summary	248

Chapter 7: Data Access in J2EE Applications	251
Data Access Goals	252
Business Logic and Persistence Logic	252
Object-Driven and Database-Driven Modeling: A Philosophical Debate	253
O/R Mapping and the "Impedance Mismatch"	255
The Data Access Object (DAO) Pattern	257
Working with Relational Databases	259
Referential Integrity	259
Stored Procedures, Triggers, and Views	259
RDBMS Performance Issues	262
RDBMS Performance Tuning	262
Denormalization	263
Portability Versus Performance	263
Exchanging Data in Distributed Applications	265
The Value Object J2EE Pattern	265
"Generic" Value Objects	267
"Disconnected" Data Access Using JDBC Rowsets	267
Common Data Access Issues	268
Transaction Isolation	268
Pessimistic and Optimistic Locking	269
Primary Key Generation Strategies	269
Sequence Entity Bean	271
Unique ID Generation in Java	271
Database-Specific ID Generation	271
JDBC 3.0	273
Where to Perform Data Access	273
Data Access in the EJB Tier	273
Entity EJBs	274
Session EJBs and Helper Classes	275
Data Access in the Middle Tier without Using EJB	275
Data Access in the Web Tier	276
Servlets and Web-Specific Classes	276
Data Access from JSP Pages	276
Summary	278
Data Modeling in the Sample Application	278

Chapter 8: Data Access Using Entity Beans	285
Entity Bean Concepts	286
Definition	287
How Should We Use Entity Beans?	288
The Granularity Debate	288
The Business Logic Debate	290
Session Beans as Mediators	291
CMP Versus BMP	292
Entity Beans in EJB 2.0	294
Local Interfaces	294
Home Interface Business Methods	295
EJB 2.0 CMP	296
Basic Concepts	296
Container-Managed Relationships (CMR)	297
EJB QL	298
Limitations of O/R Modeling with EJB 2.0 Entities	299
Custom Entity Behavior with CMP/BMP Hybrids	299
Entity Bean Caching	300
Entity Bean Locking Strategies	301
Exclusive Locking	301
Database Locking	302
Read-only and "Read-mostly" Entities	302
Transactional Entity Caching	304
Entity Bean Performance	305
Tool Support for Entity Beans	306
Summary	306
Chapter 9: Practical Data Access	311
Data Access Technology Choices	312
SQL-Based Technologies	312
JDBC	312
SQLJ	313
O/R Mapping Technologies	315
Established Commercial Products	315
Java Data Objects (JDO)	317
Choosing a Data Access Strategy for the Sample Application	319
JDBC Subtleties	320
Correct Exception Handling	320
Extracting Information from SQLExceptions	322
The PreparedStatement Question	323

A Generic JDBC Abstraction Framework	324
Motivation	325
Aims	326
Exception Handling	327
A Generic Data-Access Exception Hierarchy	328
Converting JDBC Exceptions to Generic Exceptions	331
Two Levels of Abstraction	333
A Framework to Control JDBC Workflow and Error Handling	333
"Inversion of Control" Revisited	334
The com.interface21.jdbc.core package	334
Using the JdbcTemplate Class	340
A Higher Level of Abstraction: Modeling RDBMS Operations as Java Objects	342
Implementation of the com.interface21.jdbc.object Package	342
Using the JDBC Object Abstraction	347
JDBC Abstraction Summary	351
Implementing the DAO Pattern in the Sample Application	353
Summary	360
Chapter 10: Session Beans	363
Using Stateless Session Beans	364
Benefits of Stateless Session Beans	364
Stateless Session Beans and Internal State	365
Implications of Stateless Session Bean Pooling	365
Using Stateful Session Beans	366
Why Not to Use Stateful Session Beans	366
Performance and Scalability Issues	366
Reliability Issues	368
When to Use Stateful Session Beans	370
Session Synchronization	370
Protecting Stateful Session Beans from Concurrent Calls	371
Patterns for Achieving Stateful Functionality with SLSBs	371
Object Parameter	372
Using a "Required Workflow Exception" to Mimic an SFSB State Machine	372
Using a Stateful Session Bean as Controller	373
J2EE Design Patterns Applicable to Session Beans	373
The Session Faade Pattern in Distributed J2EE Applications	374
The EJB Command Design Pattern	374
Implementing the EJB Command Design Pattern	375
Advantages and Disadvantages of the EJB Command Design Pattern	377
Using Commands without Adopting the Command Design Pattern	378
Session Bean Implementation issues	379
Error Handling in EJBs	379
The EJB Container's Behavior on Exceptions	380
Understanding EJB API Exceptions	381
Transaction Attributes for EJBs using CMT	382
The Business Methods Interface "Pattern"	386

Table of Contents

Session Beans in the Sample Application	389
Summary	389
Chapter 11: Infrastructure and Application Implementation	393
Infrastructure	394
Goals of a Strong Infrastructure	395
Using a Framework to Configure Application Components	396
The Problem	396
Using JavaBeans	397
Using a "Bean Factory"	401
The Application Context	406
Testing Implications	408
Summary of Application Configuration Infrastructure	409
Managing API Complexity	410
Implementing EJBs	410
Accessing EJBs	417
Using JMS	425
Implementing Business Logic	428
Implementing the Sample Application	428
Defining Business Interfaces	429
Determining Implementation Strategy	432
Implementing the BoxOffice	433
Using JMS to Propagate Data Updates	435
Pulling It All Together	436
Summary	437
Chapter 12: Web-Tier MVC Design	441
The Challenges of Web Development	442
Lessons Learned in Java Web Development	443
The Shortcomings of Servlet-only Solutions	443
JSP: Promise and Temptation	444
"JSP Model 1" Architecture	444
The Temptation of the JSP Standard Infrastructure	445
Striking a Balance	446
Web-Tier Design Goals	447
A Clean Web Tier	447
A Thin Web Tier	447
MVC Concepts and the Front Controller J2EE Pattern	448
Concepts	449
The MVC Triad	449
Control Flow	452

Pattern Variants	453
Template Selection Servlet	453
How Many Controller Servlets?	454
JSP or Servlet Controller?	454
Should a Request Cause the Creation of a Command?	455
Implementation Goals	455
Web Application Frameworks	456
Common Concepts	456
Available Frameworks	457
Struts	457
Maverick	461
WebWork	463
Integrating a Web Application Framework into Overall Application Architecture	465
The Web Application Framework Used in the Sample Application	467
Design Goals	468
Basic MVC Control Flow	469
Controller Servlet	471
Request to Controller Mapping (com.interface21.web.servlet.HandlerMapping)	473
Request Controller (com.interface21.web.servlet.mvc.Controller)	474
Models	475
Views	476
ViewResolver	477
ContextLoaderServlet	478
Custom Tags	479
Workflow Refinements	479
Examples	480
A Basic Controller Implementation	481
A Controller Exposing Bean Properties	482
A Multi-Action Controller	484
Web-Tier Session Management	488
Session State Managed by the J2EE Server	488
Clustering and Replication	488
Simple Optimizations	489
Session State Held in the Browser	490
Session State Management with Cookies	490
Session State Management with Hidden Form Fields	491
Processing User Input	492
Data Binding and Displaying Input Errors for Resubmission	492
Approaches to Data Binding in MVC Frameworks	493
JSP Custom Tags	494
Data Validation	496
Where Should Data Validation be Performed?	496
Data Validation in the Framework Described in this Chapter	498
Implementing the Web Tier in the Sample Application	506
Overview	506
Handling a Seat Reservation Request	508
Implementation Review	511
Summary	512

Chapter 12: Views in the Web Tier	515
Decoupling Controllers and Views	517
Constructing the View for the Reservations Page	519
Information Presented and Required Formatting	519
The Model Behind this View	521
Model Principles	524
JSP Views	526
What We Want to Avoid	527
How to Use JavaBeans in JSP Pages	531
JSP Custom Tags	532
The Java Standard Tag Library	533
Other Third-Party Tag Libraries	535
Implementing Your Own Tag Libraries	535
Guidelines for Custom Tag Use	536
Guidelines for JSP Use	536
Looking Ahead: Implications of JSP 2.0	538
A JSP View for the Example	538
JSP Summary	543
Dedicated Template Languages	544
Common Concepts	544
WebMacro	545
Velocity	546
Velocity Concepts	546
A Velocity Template for our Example	548
Velocity Summary	550
FreeMarker	551
XSLT	551
When to Use XSLT	552
What Do We Want from XSL?	553
How to Use XSLT in Views	553
Using XSLT Instead of JSP	553
Using XSLT from JSP Custom Tags	554
Implementing our Example Using a "Pure" XSLT Approach	555
Alternative Approaches to Markup Generation	561
HTML Generation Libraries	562
XMLC	563
An XMLC Template for Our Example	564
Compiling the Template	567
Manipulating the XMLC Object Generated from the Template	568
Further Reading on XMLC	570
Generating Binary Content	571
Generating PDF with iText	571
View Composition and Page Layout	575
Summary	579

Chapter 14: Packaging and Application Deployment	583
Packaging	584
Deployment Units	584
Expanded Deployment Units	585
Understanding J2EE Class Loading	585
Java Class Loading Concepts	585
Class Loading in J2EE	586
Server Check List	590
Recommendations	591
Further Information	592
Packaging the Sample Application	592
Application Deployment: Common Concepts	596
Configuring a Server to Run the Application	596
Creating Connection Pools	597
Creating JMS Destinations	597
Setting up Authentication	597
Installing Libraries	598
Writing Proprietary Deployment Descriptors for an Application	598
EJB-Specific Configuration	599
Web-Tier Configuration	599
Deploying an Application	600
Deployment Parameters for the Sample Application	600
Deploying the Sample Application on JBoss 3.0	601
Understanding the JBoss Directory Structure	601
Configuring a JBoss Server to Run the Sample Application	602
Creating a Connection Pool	602
Creating JMS Destinations	603
Installing the Service Definition File	604
Reviewing Configuration	604
Writing JBoss Deployment Descriptors for the Sample Application	605
Deploying the Application	607
Summary	608
Chapter 15: Performance Testing and Tuning an Application	611
Strategic Issues and Definitions	612
Performance and Scalability	613
Setting Clear Goals for Performance and Scalability	614
Design Versus Code Optimization	614
Tools for Testing Performance and Throughput	615
Preparing to Benchmark	616
Web Test Tools	617
Microsoft Web Application Stress Tool	617
Non-Web Testing Tools	619

Table of Contents

Locating Performance or Scalability Problems	622
Testing in Layers	623
Profiling Tools	623
JVM Profiling Options	624
The JProbe Profiler	625
Addressing Performance or Scalability Problems	630
Server Choice and Server Configuration	630
Dispensing with Redundant Container Services	631
Caching	632
When to Cache	632
Where to Cache	633
Third-party Caching Products for Use in J2EE Applications	637
Code Optimization	638
Case Study: The "Display Show" Page in the Sample Application	643
Performance in Distributed Applications	653
The Overhead of Remote Method Invocation (RMI)	653
Minimizing Remote Calls	655
Application Partitioning	655
Consolidating Remote Calls	658
Moving Data Efficiently	658
Serialization Optimizations	659
Other Data Transfer Strategies	662
Collocating Components in the Same JVM	663
Web-Tier Performance Issues	663
View Performance	663
Web Caching Using HTTP Capabilities	666
Cache Control HTTP Headers	666
Using the Servlet API to Control Caching	668
Implications for MVC Web Applications	669
The Welcome Page in the Sample Application	670
Edged Side Caching and ESI	671
The Primary Causes of Performance and Scalability Problems in J2EE Applications	672
Summary	673
Chapter 16: Conclusion: Making J2EE Work for You	675
General Principles	676
Projects	679
Appendix A: Implementing View Technologies	683
Decoupling Controllers from View Technologies Using a View Interface	684
View Implementations	686

JSP	688
Configuring the JSTL	688
The InternalResourceView View Implementation	689
Defining JSP Views for Use in an Application	691
Velocity	691
Installing and Configuring Velocity	691
Implementing the View Interface for Velocity	692
Exposing Model Data to a Velocity Template	693
Providing Support for Date and Currency Formatting	694
Defining Velocity Views for Use in an Application	695
XSLT	696
Installing Domify	696
Implementing the View Interface for XSLT	697
Performing XSLT transforms	697
Date and Currency Formatting Support	700
Defining XSLT Views for Use in an Application	701
XMLC	703
Installing and Configuring XMLC	703
Implementing the View Interface for XMLC	704
Defining XMLC Views for Use in an Application	705
Generating PDF with iText	705
Installing iText	705
Implementing the View Interface for PDF Generation with iText	705
Defining PDF Views for Use in an Application	707
Additional Views	707
Custom Views	708
<u>Index</u>	<u>711</u>



Introduction

I believe that J2EE is the best platform available for enterprise software development today. It combines the proven merits of the Java programming language with the lessons of enterprise software development in the last decade.

Yet this promise is not always fulfilled. The return on investment in many J2EE projects is disappointing. Delivered systems are too often slow and unduly complex. Development time is often disproportionate to the complexity of business requirements.

Why? Not so much because of shortcomings in J2EE as because J2EE is often used badly. This often results from approaches to architecture and development that ignore real world problems. A major contributing factor is the emphasis in many J2EE publications on the J2EE specifications rather than the real world problems people use them to address. Many issues that commonly arise in real applications are simply ignored.

When reading J2EE discussion forums, I'm struck by how little guidance and direction many developers find, and how much time and effort they waste as a result. In many cases, these developers have years of IT experience, and yet are finding it hard to come to grips with J2EE.

The problem is not a lack of information about J2EE components. Many books and web sites do a good job describing servlets, EJBs etc. Enabling technologies such as JNDI, RMI, and JMS are equally well served.

The problem is in getting to the next level – taking these construction materials and using them to build applications that meet real business requirements in a reasonable period of time. Here, I feel that much of the existing literature on J2EE is a hindrance rather than help. There is a gulf between the world of J2EE books – the world as it perhaps should be – and the real world of enterprise software projects.

This book aims to address this problem and provide clear guidance and direction on using J2EE effectively in practice. I'll help you to solve common problems with J2EE and avoid the expensive mistakes often made in J2EE projects. I will guide you through the complexity of the J2EE services and APIs to enable you to build the simplest possible solution, on time and on budget. I'll take a practical, pragmatic approach, questioning J2EE orthodoxy where it has failed to deliver results in practice and suggesting effective, proven approaches.

I feel that no existing book delivers this. The closest is probably *Core J2EE Patterns* from Prentice Hall (ISBN: 0-130648-84-1), which generated much excitement on its release. Here at last was a book that addressed *how* to use J2EE components. *Core J2EE Patterns* is a good book and a valuable resource for J2EE architects and developers. In particular, the terminology it uses has become widely accepted, but it's a Sun publication, and can't help reflecting the "party line".

It also deals purely with the J2EE standards, paying little attention to issues encountered in working with real application servers. It fails to provide clear guidance: too often, it sits on the fence, presenting a variety of very different alternative "patterns". Readers able to choose confidently between them have little to learn from the book.

The more I considered the available publications, sample applications, and discussion forums, the more convinced I became that J2EE needed a healthy dose of pragmatism. J2EE is a great platform; unfortunately, many of the architectural practices promoted for it are not, and don't help to solve many common problems. Many J2EE sample applications, such as Sun's Java Pet Store, are disappointing. They don't face real world problems. They perform poorly, and their code often contains sloppy practices, providing a poor model.

I was also struck by the difference in outlook between developers new to J2EE and those who had actually used J2EE to build enterprise systems. A former colleague used the wonderfully evocative word "gnarly" to describe developers who've come to grips with practical challenges of working with a technology and bear the scars. While those new to J2EE sounded like J2EE evangelists, the "gnarly" developers told a different story. They had had to jettison some of the ideological baggage of the innocents to implement necessary functionality or achieve adequate performance. Like my colleagues and myself, they'd found that reality intruded harshly on the initial vision.

In this book I'll draw on my experience and industry knowledge to help you design and develop solutions that work in practice, without the need for you to go through a painful process of discovering the difference between J2EE theory and reality.

J2EE Myths

I believe that the causes of disappointing outcomes with J2EE can usually be traced to a few common myths, which underpin many explicit and implicit assumptions in development projects:

- ❑ J2EE is about portability, between application servers and databases.
- ❑ J2EE is the best answer to all the problems of enterprise development. If a problem that would traditionally have been solved using non-J2EE technologies, such as RDBMS stored procedures, can be solved with standard J2EE technology, it's always best to use the "pure" J2EE approach.

- ❑ J2EE servers take care of performance and scalability, leaving developers to get on with implementing business logic. Developers can largely ignore the performance implications of J2EE "patterns" and rely on acceptable performance in production.
- ❑ J2EE enables developers to forget about low-level problems such as data access and multi-threading, which will be handled transparently by the application server.
- ❑ All J2EE applications should use EJB, which is the essential J2EE technology for developing enterprise-class applications.
- ❑ Any problems with J2EE will soon be addressed by more sophisticated application servers.

Let's quickly consider each of these myths in turn.

Portability is a great bonus of the J2EE platform. As we'll see, portability *can* be achieved in real applications, but it's not the point of J2EE. The requirement of the vast majority of projects is to build an application that solves a particular problem well on one target platform. An application that runs badly on one platform will never be ported to other platforms (the application might be ported to another operating system that runs on more powerful hardware to gain adequate performance, but that's not the kind of portability that professional developers aspire to).

J2EE orthodoxy holds that an application should be portable across J2EE application servers and must be able to work with different databases. The distinction between these two goals is important, and sometimes missed. Portability between application servers may deliver business value and is usually a realistic goal. Portability between databases is much more fraught, and often provides no business value.

Portability is usually taken to mean *code portability*: the ability to take the application and run it on another platform without any change. I believe that this is an expensive misconception. Naïve emphasis on total code portability often leads to heavy costs in lost productivity and less satisfactory deliverables. **Write Once Run Anywhere (WORA)**, while a reality where Java itself is concerned, is a dangerous slogan to apply to enterprise development, which depends on a range of resources.

I'm not talking about the minority of projects to develop "shrink-wrapped" components (usually EJBs). This appealing concept is still to be proven in the market. Furthermore, I'm yet to see a non-trivial component that aimed for both, application server portability (which makes sense in this situation) and database portability (which will almost certainly be more trouble than it's worth).

I prefer **Design Once, Re-implement a Few Interfaces Anywhere (DORAFIA)**. I accept that this is not so catchy, and that people are unlikely to leave Java One chanting it. This more realistic approach is widely used in other domains, such as windowing systems.

The portability myth has led to wide acceptance that J2EE applications can't use the capabilities of today's relational databases, but should use them only as dumb storage. This does great harm in the real world.

This is not to say that I don't believe that J2EE applications can or should be portable. I'm just arguing for a more pragmatic and realistic view of portability. We *can* design J2EE applications to be ported easily; we can't do the same thing with a proprietary technology such as .NET.

It's pleasant to imagine that J2EE is the final stage of the evolution of enterprise architecture; that finally, the application of object technology and the Java language has cracked problems the industry has wrestled with for decades. Unfortunately, this is not the reality, although it's implicitly assumed in many approaches to J2EE development. J2EE builds on many of the technologies that preceded it. It's a step forward, but it won't be the last and it doesn't address all the issues of enterprise software development.

Exaggerated emphasis on portability, along with this J2EE-centric attitude, has led to the assumption that if something can't be done in standard J2EE, it's a design error to do it. This is even creeping into the EJB specification with the introduction of EJB QL: a portable but immature query language that's more complex but far less powerful than the familiar, mature, and largely standard SQL that is available to the great majority of J2EE applications.

I like to think of a J2EE server as the conductor of a group of enterprise resources such as databases. A good conductor is vital to any performance. However, a conductor doesn't attempt to play individual instruments, but leaves this to skilled specialists.

Perhaps the most dangerous myth is that J2EE is the easy route to good performance and scalability, and that efficiency is a lesser concern than approved J2EE "patterns". This leads to naïve and inefficient designs. This is unfortunate, as outside the Java community Java has always been dogged by fears of poor performance. Today, the evidence is that the Java language offers good performance, while some popular J2EE "patterns" offer very poor performance.

We cannot assume that the application server can take care of performance and scalability. In fact, J2EE gives us all the rope we need to tie up not only our J2EE application server, but the database as well. Had optimal performance been the main goal of software development, we'd have been writing web applications in C or assembly language. However, performance *is* vital to the business value of real-world applications. We can't rely on Moore's Law to allow us to solve performance problems with faster hardware. It's possible to create problems that prevent adequate performance, regardless of hardware power.

The idea that the J2EE server should transparently handle low-level details such as data access is appealing. Sometimes it's achievable, but can be dangerous. Again, let's consider the example of relational databases. Oracle, the leading enterprise-class RDBMS, handles locking in a completely different way compared to any other product. The performance implications of using coarse or fine-grained transactions also vary between databases. This means that "portability" can be illusory, as the same code may behave differently in different RDBMS products.

Oracle and other leading products are expensive and have impressive capabilities. Often we'd *want* (or need) to leverage these capabilities directly. J2EE provides valuable standardization in such infrastructure services as transaction management and connection pooling, but we won't be saying goodbye to those fat RDBMS product manuals any time soon.

The "J2EE = EJB" myth can lead to particularly expensive mistakes. EJB is a complex technology that solves some problems well, but adds more complexity than business value in many situations. I feel that most books ignore the very real downside of EJB, and encourage readers to use EJB automatically. In this book, I'll provide a dispassionate view of the strengths and weaknesses of EJB, and clear guidance on when to use EJB.

Allowing the technology used (J2EE or any other technology) to determine the approach to a business problem often leads to poor results. Examples of this mistake include determining that business logic should always be implemented in EJBs, or determining that entity beans are the one correct way to implement data access. The truth is that only a small subset of J2EE components – I would include servlets and stateless session EJBs – are central to most J2EE applications. The value of the others varies greatly depending on the problem in hand.

I advocate a problem-driven, not technology-driven, approach (Sun's "J2EE Blueprints" have probably done as much harm as good, by suggesting a J2EE technology-driven approach). While we should strive to avoid reinventing the wheel, the orthodoxy that we should never ourselves implement something that the server can (however inefficiently), can be costly. The core J2EE infrastructure to handle transaction management, etc., is a godsend; the same cannot be said for all the services described in the J2EE specifications.

Some will argue that all these problems will soon be solved, as J2EE application servers become more sophisticated. For example, ultra-efficient implementations of entity bean **Container-Managed Persistence (CMP)** will prove faster than RDBMS access using raw SQL. This is naïve and carries unacceptable risk. There is little place for faith in IT. Decisions must be made on what has been proven to work, and faith may be misplaced.

There are strong arguments that some features of J2EE, such as entity beans, can *never* be as performant in many situations as some alternatives. Furthermore, the Promised Land is *still* just around the corner. For example, entity beans were soon going to provide brilliant performance when they were first introduced into the EJB specification in 1999. Yet the next two years revealed severe flaws in the original entity bean model. Today, the radical changes in the EJB 2.0 specification are still to be proven, and the EJB 2.1 specification is already trying to address omissions in EJB 2.0.

How is this Book Different?

First, it's an independent view, based on my experience and that of colleagues working with J2EE in production. I don't seek to evangelize. I advocate using J2EE, but caution against J2EE orthodoxy.

Second, it has a practical focus. I want to help you to implement cost-effective applications using J2EE. This book aims to demystify J2EE development. It shows how to use J2EE technologies to reduce, rather than increase, complexity. While I don't focus on any one application server, I discuss some of the issues you're likely to encounter working with real products. This book doesn't shy away from real-world problems that are not naturally addressed by the J2EE specifications. For example, how do we use the Singleton design pattern in the EJB tier? How should we do logging in the EJB tier?

This book doesn't seek to cover the whole of J2EE. It aims to demonstrate effective approaches to solving common problems. For example, it focuses on using J2EE with relational databases, as most J2EE developers face O/R mapping issues. In general, it aims to be of most help in solving the most common problems.

We'll look at a single sample application throughout the book. Rather than use an unrealistic, abstract example as we discuss each issue, we'll look at a small part of a larger, more realistic, whole. The sample application is an online ticketing application. It is designed not to illustrate particular J2EE *technologies* (like many sample applications), but common *problems* facing J2EE architects and developers.

This book is about quality, maintainability, and productivity.

This is the book I wished I'd had as I worked on my first J2EE project. It would have saved me a lot of effort, and my employer a lot of money.

My Approach

This book is *problem*-oriented rather than *specification*-oriented. Unlike many books on J2EE, it doesn't aim to cover all the many services and APIs. Instead, it recognizes that not all parts of J2EE are equally useful, or of interest to all developers, and focuses on those parts that are used in building typical solutions.

Software design is as much art as science. The richness of J2EE means that it is often possible to find more than one good solution to a problem (and many bad solutions). While I make every effort to explain my views (or prejudices), this book naturally reflects my experience of and attitude towards working with J2EE. I present an approach that I've found to work well. However, this doesn't mean that it's the *only* valid approach.

The book reflects my attitudes towards software development in general:

- ❑ I try to avoid religious positions. I've never understood the energy and passion that so many developers devote to flame wars. This benefits no one.
- ❑ I'm a pragmatist. I care about outcomes more than ideology. When I work on a project, my primary goal is to deliver a quality result on time and budget. The technology I use is a tool towards that goal, not an end in itself.
- ❑ I believe that sound OO principles should underpin J2EE development.
- ❑ I believe that maintainability is crucial to the value of any deliverable.

*In keeping with this pragmatic approach, I'll frequently refer to the **Pareto Principle**, which states that a small number of causes (20%) are responsible for most (80%) of the effect. The Pareto Principle, originally drawn from economics, is highly applicable to practical software engineering, and we'll come across it repeatedly in approaching J2EE projects. For example, it can suggest that trying to solve all problems in a given area can be much harder (and less cost-effective) than solving just those that matter in most real applications.*

My approach reflects some of the lessons of **Extreme Programming (XP)**. I'm a methodology skeptic, and won't attempt to plug XP. This isn't an XP book, but I feel that XP offers a valuable balance to J2EE theory. In particular, we'll see the value of the following principles:

- ❑ Simplicity. XP practitioners advocate doing "the simplest thing that could possibly work".