

Enterprise Systems Development using Java-EE

Dr Fritz Solms and Craig Edwards
fritz@solms.co.za, craig@solms.co.za
<http://www.solms.co.za>

August 8, 2014

Contents

| | | |
|----------|---|-----------|
| I | The Java Platform, Enterprise Edition (Java EE) reference architecture | 15 |
| 1 | Introduction | 17 |
| 2 | The purpose of architecture | 19 |
| 3 | Definition of software architecture | 21 |
| 4 | Components of software architecture | 23 |
| 5 | Questions addressed by software architecture | 25 |
| 6 | Introduction | 27 |
| 6.1 | What is a reference architecture and a framework | 27 |
| 6.2 | Examples | 27 |
| 6.3 | Benefits and risks of using reference architectures | 28 |
| 7 | Java-EE: first level of granularity | 29 |
| 8 | What does the EJB container provider? | 33 |
| 8.1 | Introduction | 33 |
| 8.2 | Concurrency support | 33 |
| 8.3 | Component pooling | 33 |
| 8.4 | Network enabling | 34 |
| 8.5 | Persistence | 34 |
| 8.6 | Component location transparency | 34 |
| 8.7 | Transaction support | 35 |
| 8.8 | Security support | 35 |
| 8.9 | Session management | 36 |
| 8.10 | Interception | 36 |
| 8.11 | Resource connection pooling | 36 |
| 8.12 | Implicit monitoring | 36 |
| 9 | Enterprise beans | 39 |
| 9.1 | Bean species | 39 |
| 9.1.1 | Session beans | 39 |
| 9.2 | Elements of enterprise beans | 39 |
| 9.2.1 | EJBObject | 39 |

| | | |
|-----------|---|-----------|
| 9.2.2 | Local and remote interfaces | 40 |
| 9.3 | Bean restrictions | 40 |
| 9.3.1 | Beans can't give clients direct access to the bean instance | 40 |
| 9.3.2 | Enterprise beans may not accept network server connections | 40 |
| 9.3.3 | Enterprise beans should be single-threaded | 40 |
| 9.3.4 | Enterprise beans should not create a user interface | 41 |
| 9.3.5 | Stateful session beans can't have persistent class fields | 41 |
| 9.3.6 | Enterprise beans may not use any native libraries | 41 |
| 10 | Entity objects | 43 |
| II | Maven | 45 |
| 11 | Introduction | 47 |
| 11.1 | Guiding principles of Maven | 47 |
| 11.2 | What does Maven provide? | 48 |
| 11.3 | Maven versus Ant | 48 |
| 11.4 | What is Maven really? | 49 |
| 12 | Maven's Project Object Model (POM) | 51 |
| 12.1 | POM structure | 51 |
| 12.2 | Project Identifiers | 52 |
| 12.2.1 | Group Identifier | 53 |
| 12.2.2 | Artifact Identifier | 53 |
| 12.2.3 | Version | 53 |
| 12.2.4 | Packaging | 53 |
| 12.3 | POM inheritance | 53 |
| 12.3.1 | Declaring the parent POM | 53 |
| 12.3.2 | The Super POM | 54 |
| 12.3.3 | The effective POM | 55 |
| 12.4 | Project dependencies | 55 |
| 12.4.1 | Specifying project dependencies | 55 |
| 12.4.2 | Transitive dependencies | 57 |
| 12.4.3 | Dependency management | 57 |
| 12.4.4 | Exclusions | 58 |
| 12.5 | Project modules | 59 |
| 12.6 | Build customization | 59 |
| 12.6.1 | Customizing/configuring plugin behaviour | 60 |
| 12.6.2 | Adding a goal to a life cycle phase | 60 |
| 12.7 | Distribution Information | 62 |
| 12.7.1 | Specifying distribution Info | 62 |
| 12.7.2 | Specifying login credentials | 62 |
| 13 | Maven's life cycles | 65 |
| 13.1 | Maven's default build life cycle | 66 |
| 13.2 | Maven's clean life cycle | 67 |
| 13.3 | Default goal bindings for the site life cycle | 68 |
| 13.4 | Package-based goal bindings | 68 |
| 13.5 | Project based life cycle goals | 68 |

| | |
|--|------------|
| 14 Executing Maven | 71 |
| 14.1 Executing a plugin goal | 71 |
| 14.2 Executing a life cycle phase | 71 |
| 15 Maven repositories | 73 |
| 15.1 Repository structure | 73 |
| 15.2 Repository locations | 73 |
| 15.2.1 Remote repositories | 73 |
| 15.2.2 The local repository | 74 |
| 15.3 How Maven uses Repositories | 75 |
| 15.4 Repository tools | 75 |
| 16 Hello World via Maven | 77 |
| 16.1 Creating a project via the Archetype plugin | 77 |
| 16.2 Executing default life cycle phases | 78 |
| 16.2.1 Deploying onto a server | 79 |
| 16.3 Executing specific plugin goals | 81 |
| 16.3.1 Executing a program from Maven | 81 |
| 16.4 Generating documentation | 82 |
| 16.5 Cleaning the project | 82 |
| 17 Maven JAXB Sample | 83 |
| 17.1 The schema | 83 |
| 17.2 The Test Application | 84 |
| 17.3 Java-6 POM | 85 |
| 17.4 Executing goals | 87 |
| 18 Maven JAXWS Sample | 89 |
| 18.1 Java-6 POM | 89 |
| 18.2 The Test Application | 90 |
| 18.3 Executing the web service test | 91 |
| III Context and Dependency Injection (CDI) | 93 |
| 19 Dependency Injection | 95 |
| 19.1 The problem | 95 |
| 19.2 The solution provided by dependency injection | 96 |
| 19.3 Implementations | 96 |
| 20 Introduction to CDI | 99 |
| 20.1 What does CDI provide? | 99 |
| 20.2 Decoupling through CDI | 100 |
| 21 Understanding CDI | 101 |
| 21.1 What is a CDI bean? | 101 |
| 21.2 Specifying the state retention period With bean scope | 101 |
| 21.3 Assigning Beans EL Names | 102 |
| 21.4 Injecting a CDI bean | 102 |
| 21.5 Using qualifiers | 102 |

| | | |
|-----------|---|------------|
| 21.6 | A simple example | 103 |
| 21.7 | How are bean references obtained? | 104 |
| IV | The Java Persistence Api (JPA) | 105 |
| 22 | Overview | 107 |
| 22.1 | What is JPA? | 107 |
| 22.2 | Why use JPA? | 107 |
| 22.3 | What does JPA provide? | 108 |
| 22.4 | JPA providers | 108 |
| 23 | Persistence context | 109 |
| 23.1 | Overview | 109 |
| 23.1.1 | What is a persistence context? | 109 |
| 23.1.2 | What is an entity manager? | 109 |
| 23.1.3 | Optimistic concurrency control | 109 |
| 23.1.4 | Life span of entity manager | 110 |
| 23.1.5 | Transaction management | 110 |
| 23.1.6 | The scope of a persistence context | 111 |
| 23.1.7 | How are entity managers obtained? | 111 |
| 23.1.8 | Detaching objects from a persistence context | 112 |
| 23.2 | Persistence context configuration | 112 |
| 23.2.1 | Configuring the persistence context for a non-managed application | 112 |
| 23.2.2 | Configuration of persistence context for managed applications | 113 |
| 24 | Entities | 115 |
| 24.1 | Simple entities | 115 |
| 24.1.1 | Declaring entities | 115 |
| 24.1.2 | Requirements for entities | 115 |
| 24.1.3 | What is persisted? | 116 |
| 24.1.4 | Embeddable classes | 117 |
| 24.1.5 | Primary keys | 118 |
| 24.1.6 | Specifying column mappings | 121 |
| 24.1.7 | Column constraints | 121 |
| 24.1.8 | Primitive collections and maps | 121 |
| 24.2 | Relationships | 122 |
| 24.2.1 | Summary of UML relationships | 122 |
| 24.2.2 | Composition relationships between entities | 125 |
| 24.2.3 | Relationship types | 125 |
| 24.2.4 | Fetching strategies | 128 |
| 24.2.5 | Specialization | 129 |
| 25 | The Java Persistence Query Language (JPQL) | 135 |
| 25.1 | JPQL versus SQL | 135 |
| 25.1.1 | Result collections in JPQL | 135 |
| 25.1.2 | Selecting entity attributes | 136 |
| 25.2 | Statement types | 136 |
| 25.2.1 | Elements of JPQL query statement | 136 |
| 25.2.2 | Elements of update and delete statements | 137 |

| | | |
|-----------|--|------------|
| 25.3 | Polymorphism | 137 |
| 25.4 | Navigating object graphs | 137 |
| 25.4.1 | Simple paths | 137 |
| 25.4.2 | Single-valued versus multi-valued paths | 138 |
| 25.5 | Specifying the source of a query | 138 |
| 25.5.1 | Selecting from multiple domains | 139 |
| 25.5.2 | Joins | 139 |
| 25.6 | Collapsing multi-valued paths into Single-valued paths | 140 |
| 25.7 | Constraining a result set via a WHERE clause | 141 |
| 25.7.1 | Comparison operators | 141 |
| 25.7.2 | Calculation and logical operators | 141 |
| 25.7.3 | Using collection variables in WHERE clauses | 141 |
| 25.8 | Constructing result objects | 142 |
| 25.9 | Nested queries | 142 |
| 25.10 | Ordering | 142 |
| 25.11 | Grouping | 142 |
| 25.12 | Query parameters | 143 |
| 25.12.1 | Positional parameters | 143 |
| 25.12.2 | Named parameters | 143 |
| 26 | Constructing and executing queries | 145 |
| 26.1 | Named queries | 145 |
| 27 | JPA converters | 147 |
| 27.1 | Defining custom converters? | 147 |
| 27.2 | Applying custom converters | 147 |
| 27.3 | Default converters | 148 |
| 28 | Calling stored procedures | 149 |
| 29 | The criteria API | 151 |
| 29.1 | Overview | 151 |
| 29.1.1 | Benefits of using the criteria API for specifying queries | 151 |
| 29.1.2 | Query tree | 152 |
| 29.2 | Generating the JPA metamodel | 152 |
| 29.2.1 | Maven build declarations to generate a canonical metamodel | 152 |
| 29.2.2 | Generated metamodel classes | 154 |
| 29.3 | Simple example | 154 |
| 29.4 | Query operators | 155 |
| 29.5 | Composite predicates | 155 |
| 29.6 | Ordering | 156 |
| 29.7 | Joins | 156 |
| V | Enterprise Java Beans | 157 |
| 30 | Session beans | 161 |
| 30.1 | Remote interfaces | 161 |
| 30.1.1 | Defining remote interfaces | 161 |
| 30.1.2 | Access path when using a remote interface | 162 |

| | | |
|-----------|--|------------|
| 30.2 | Local interfaces | 163 |
| 30.2.1 | Defining local interfaces | 163 |
| 30.2.2 | Access path when using a local interface | 163 |
| 30.3 | Switching between local and remote interfaces | 164 |
| 30.3.1 | Parameter handling for plain Java objects | 164 |
| 30.3.2 | Parameter handling for remote Java objects (RMI) | 164 |
| 30.4 | Automatically generated interfaces | 164 |
| 30.4.1 | Automatic interface naming | 164 |
| 30.4.2 | Which services are published in the automatically generated interface? . . | 165 |
| 30.4.3 | Automatically generated local interfaces | 165 |
| 30.4.4 | Automatically generated remote interfaces | 165 |
| 30.4.5 | Should one automatically generate interfaces? | 165 |
| 30.5 | Stateless session beans | 165 |
| 30.5.1 | Life cycle of a stateless session bean | 165 |
| 30.6 | Stateful session beans | 166 |
| 30.6.1 | Life cycle of a stateful session bean | 166 |
| 30.7 | Asynchronous session bean services | 166 |
| 30.7.1 | Why should one not simply create a thread? | 167 |
| 30.7.2 | Simple asynchronous requests | 167 |
| 30.7.3 | Deferred asynchronous requests | 168 |
| 30.7.4 | Asynchronous beans | 168 |
| 30.8 | Registering a connection pool to an email server | 169 |
| 30.9 | Sending an email from an enterprise bean | 169 |
| 31 | Interceptors | 171 |
| 31.1 | When should one use interceptors? | 171 |
| 31.2 | Defining interceptors | 171 |
| 31.2.1 | The interception context | 171 |
| 31.2.2 | Interceptor methods | 172 |
| 31.2.3 | Interception classes | 172 |
| 31.3 | Interception order | 174 |
| 31.4 | Default interceptors | 174 |
| 31.5 | Exercises | 175 |
| 32 | The Java Messaging Service (JMS) | 177 |
| 32.1 | Introduction | 177 |
| 32.2 | Styles of Messages: Point-Point vs Publish-Subscribe Domains | 177 |
| 32.2.1 | The Point-to-Point Messaging Domain | 177 |
| 32.2.2 | The Publish-Subscribe Messaging Domain | 178 |
| 32.3 | Message Types | 178 |
| 32.4 | Using JMS queues and topics | 179 |
| 32.4.1 | General Algorithm for Connecting to a Queue or Topic | 179 |
| 32.4.2 | Using JMS queues | 179 |
| 32.4.3 | Using JMS Topics | 183 |
| 32.5 | The Structure of a JMS Message | 186 |
| 32.5.1 | JMS Message Headers | 187 |
| 32.6 | Durable Subscribers | 187 |
| 32.7 | Messages Participating in Transactions | 188 |
| 32.8 | Selected message retrieval | 188 |

| | |
|---|------------|
| 33 Singleton beans | 189 |
| 33.1 Declaring singletons | 189 |
| 33.2 Using singleton beans | 189 |
| 33.3 Application startup and shutdown callbacks | 190 |
| 34 The Java-EE timer service | 191 |
| 34.1 What can the Java-EE timer service be used for? | 191 |
| 34.2 Architecture of timer service | 192 |
| 34.3 Specifying the schedule | 192 |
| 34.3.1 Specifying schedule expressions | 192 |
| 34.4 Automatic timer creation | 192 |
| 34.5 Programmatic timer creation | 193 |
| 35 Transactions | 195 |
| 35.1 The Java Transaction API (JTA) | 195 |
| 35.1.1 Overview of JTA | 195 |
| 35.1.2 Examples of using JTA directly | 197 |
| 35.2 Declarative transaction demarcation | 199 |
| 35.2.1 Transaction attributes | 200 |
| 35.2.2 Selecting a transaction attribute | 201 |
| 35.2.3 Annotating Session Beans with transaction attributes | 202 |
| 35.2.4 Transaction boundaries on method boundaries | 202 |
| 35.3 Potential problems caused by concurrent access | 202 |
| 35.3.1 Introduction | 202 |
| 35.3.2 Update loss | 202 |
| 35.3.3 Dirty Reads | 202 |
| 35.3.4 Non-repeatable reads | 203 |
| 35.3.5 Phantom reads | 203 |
| 35.4 Transaction isolation levels | 203 |
| 35.4.1 ReadUncommitted | 203 |
| 35.4.2 ReadCommitted | 203 |
| 35.4.3 Repeatable reads | 204 |
| 35.4.4 Serializability | 204 |
| 35.4.5 Setting the transaction isolation level | 204 |
| 36 Security | 205 |
| 36.1 Declarative authorization | 205 |
| 36.1.1 Specifying authorization requirements | 205 |
| 36.1.2 Run-as | 206 |
| 36.2 Programmatic security | 206 |
| VI JSF | 209 |
| 37 Overview of JSF and Facelets | 211 |
| 37.1 Java-EE Architecture | 213 |
| 37.2 What do JSF/Facelets provide? | 214 |

| | |
|---|------------|
| 38 Understanding JSF | 215 |
| 38.1 Introduction | 215 |
| 38.2 Core JSF concepts | 215 |
| 38.2.1 Facelets | 215 |
| 38.2.2 JSF UI components (controls) | 215 |
| 38.2.3 JSF component tree | 216 |
| 38.2.4 JSF renderers | 216 |
| 38.2.5 Converters | 216 |
| 38.2.6 JSF validators | 216 |
| 38.2.7 Backing beans and binding components (managed beans) | 216 |
| 38.2.8 The Unified Expression Language | 217 |
| 38.2.9 JSF events | 217 |
| 38.2.10 Navigation in JSF | 217 |
| 38.2.11 Messaging | 217 |
| 38.3 Minimalist example | 217 |
| 38.3.1 The Project Object Model (POM) | 218 |
| 38.3.2 The configuration files | 220 |
| 38.3.3 Project organization | 220 |
| 38.3.4 The model | 220 |
| 38.3.5 The managed bean (backing and binding bean) | 222 |
| 38.3.6 The view | 222 |
| 38.3.7 Building, deploying and running the web app | 223 |
| 38.3.8 Executing our minimalistic web app | 224 |
| 38.3.9 Exercise | 224 |
| 38.4 The JSF Request Life-Cycle | 225 |
| 38.4.1 The JSF Request Life-Cycle | 225 |
| 38.5 Navigation | 227 |
| 38.5.1 Navigation rules | 227 |
| 38.5.2 Backing bean based navigation | 229 |
| 38.5.3 Implicit navigation | 229 |
| 38.6 Event processing | 230 |
| 38.6.1 Value change events | 230 |
| 38.6.2 Action events | 233 |
| 38.6.3 Phase events | 233 |
| 38.7 Messaging | 236 |
| 38.7.1 Messages | 236 |
| 38.7.2 Inserting messages into a message queue | 236 |
| 38.7.3 The message element | 236 |
| 38.7.4 Example | 237 |
| 39 The Unified Expression Language | 241 |
| 39.1 Overview | 241 |
| 39.2 Immediate and deferred expression evaluation | 242 |
| 39.3 Value expressions | 242 |
| 39.3.1 Navigating across object graphs | 242 |
| 39.3.2 Accessing array or list elements | 243 |
| 39.3.3 Accessing enumeration values | 243 |
| 39.4 Method expressions | 243 |
| 39.4.1 Passing parameters to methods | 243 |

| | | |
|-----------|--|------------|
| 39.5 | EL operators | 244 |
| 39.6 | Accessing context objects | 244 |
| 40 | JSF components | 245 |
| 40.1 | Overview | 245 |
| 40.2 | JSF standard components | 245 |
| 40.2.1 | Standard tag libraries | 245 |
| 40.3 | Observer (event listener) registration | 246 |
| 40.3.1 | Generic component attributes | 247 |
| 40.3.2 | UIViewRoot | 248 |
| 40.3.3 | UIComponent | 248 |
| 40.3.4 | UIForm | 249 |
| 40.3.5 | UIPanel | 250 |
| 40.3.6 | UICommand | 252 |
| 40.3.7 | UIOutput | 253 |
| 40.3.8 | UIGraphic | 254 |
| 40.3.9 | UIInput | 254 |
| 40.3.10 | Tables | 257 |
| 40.3.11 | Messaging components | 258 |
| 40.4 | JSF tag libraries | 259 |
| 40.5 | Custom components | 259 |
| 40.5.1 | Custom composite components | 260 |
| 40.5.2 | Published custom components | 265 |
| 41 | Converters | 275 |
| 41.1 | Standard converters | 275 |
| 41.1.1 | The number converter | 275 |
| 41.1.2 | Date/Time conversion | 276 |
| 41.2 | Custom converters | 276 |
| 41.3 | Examples | 277 |
| 41.3.1 | Names class | 277 |
| 41.3.2 | Book details class | 278 |
| 41.3.3 | FullNamesConverter | 279 |
| 41.3.4 | CaptureBookDetailsBinding | 280 |
| 41.3.5 | captureBookDetails | 280 |
| 41.3.6 | bookDetailsSavedConfirmation | 281 |
| 41.4 | Exercises | 282 |
| 42 | Validators | 283 |
| 42.1 | UI-based validators | 283 |
| 42.1.1 | Standard validators | 283 |
| 42.1.2 | Custom validators | 284 |
| 42.1.3 | Simple validation example | 285 |
| 42.1.4 | Exercises | 287 |
| 42.2 | Bean Validation | 287 |
| 42.2.1 | Standard constraint annotations | 288 |
| 42.2.2 | Specifying custom bean constraints | 289 |
| 42.2.3 | Annotating fields of backing bean | 290 |
| 42.2.4 | Annotating a data class (value object or entity) | 293 |

| | |
|---|------------|
| 42.2.5 Bean Validation Example | 296 |
| 43 Internationalization | 301 |
| 43.1 Overview | 301 |
| 43.2 Specifying supported locales | 301 |
| 43.3 Creating the resource bundles for the different locales | 302 |
| 43.4 Abstracting facelet texts | 302 |
| 43.5 Abstracting text generated in the binding bean (e.g. messages) | 303 |
| 44 Templating | 305 |
| 44.1 Overview | 305 |
| 44.1.1 Example use case | 305 |
| 44.2 Specifying a template | 306 |
| 44.3 Partial template population | 307 |
| 44.4 Adding the main content | 307 |
| 44.5 Multi-level templates and template parameters | 308 |
| 44.6 Repeated template application | 308 |
| 45 Reusable components | 311 |
| 45.1 Overview | 311 |
| 45.1.1 Business versus presentation layer processes | 311 |
| 45.1.2 Assembling higher level views from lower level views | 312 |
| 45.1.3 Binding components controlling presentation layer processes | 312 |
| 45.2 Defining reusable components across levels of granularity | 312 |
| 45.2.1 Composite views | 312 |
| 45.2.2 Composite backing beans | 314 |
| 45.3 Binding beans | 316 |
| 45.4 Forwarding information to managed beans | 317 |
| 46 Ajax | 319 |
| 46.1 Overview | 319 |
| 46.2 What does the AJAX framework provide? | 319 |
| 46.3 How does AJAX work? | 320 |
| 46.4 JSF Ajax support | 320 |
| 46.4.1 JSF AJAX JavaScript Library | 320 |
| 46.4.2 Server-Side Processing of AJAX Request | 321 |
| 46.5 Partial AJAX Request Life Cycle | 321 |
| 46.6 The JSF JavaScript API for AJAX | 321 |
| 46.6.1 The JSF <code>ajax</code> tag | 322 |
| 46.7 Using an AJAX request | 322 |
| 46.8 Example | 325 |
| 46.8.1 The facelet with AJAX requests | 325 |
| 46.8.2 The supporting backing bean | 326 |
| VII JAX-RS | 329 |
| 47 Overview | 331 |
| 47.1 Introduction | 331 |
| 47.2 Core Principles | 331 |

| | |
|---|------------|
| <i>CONTENTS</i> | 13 |
| 47.3 Java Rest Frameworks | 331 |
| 48 Introduction | 333 |
| 49 JAX-RS Resources | 335 |
| 49.1 Paths | 335 |
| 49.1.1 How is the absolute resource path assembled? | 335 |
| 49.1.2 Basic paths | 335 |
| 49.1.3 Embedded path variables | 336 |
| 49.2 HTTP method support | 336 |
| 50 Supported inputs and outputs | 337 |
| 50.1 How is the output type selected? | 341 |
| 51 Parameters | 343 |
| 51.1 Path Parameters | 343 |
| 51.2 Request parameters | 344 |
| 51.3 Form parameters | 344 |
| 51.4 Cookie parameters | 345 |
| 51.5 Header parameters | 345 |
| 51.6 Default parameter values | 345 |
| 52 Responses | 347 |
| 52.1 Building URIs and Responses | 347 |

Part I

The Java Platform, Enterprise Edition (Java EE) reference architecture

Chapter 1

Introduction

Java EE, the *Java Platform, Enterprise Edition* architecture is one of the most widely used reference architectures for large, interactive enterprise systems.

Java-EE supports, amongst other things, standard access channels for enterprise systems, a solid process execution environment, a range of standard integration channels, hot deployment and clustering.

The main aim is to provide a reference architecture for scalable, reliable, secure, integrable and maintainable systems.

Chapter 2

The purpose of architecture

Chapter 3

Definition of software architecture

Chapter 4

Components of software architecture

Chapter 5

Questions addressed by software architecture

1. What are the architectural components and what are their responsibilities?
2. How do the architectural components communicate?
3. How will the system handle the number of concurrent users?
4. How will authentication, authorization, confidentiality be enforced?
5. How will we ensure reliability and fail-over safety?
6. Does the software architecture support pluggability, to what extent and how?
7. Is logging supported by the software architecture, and if so, how?
8. What frameworks, technologies, protocols, . . . , will be used?
9. How does the software architecture support monitorability and auditability?
10. How does is the system deployed or ported?
11. How is maintainability supported. Can the system support life maintenance, and if so, how?
12. How is high-performance achieved?

Chapter 6

Introduction

6.1 What is a reference architecture and a framework

A reference architecture is defined as a *best-practices based template architecture which has been proven to address the typical challenges for a particular domain*. It provides a specific combination of architectural patterns, and supports a range of architectural strategies in order to concretely realize quality requirements. In addition a reference architecture commonly either relates to or specifies a set of standards.

A framework, on the other hand, is defined as a *partial or complete implementation of a reference architecture*.

A framework thus provides an implementation of the infrastructure of the reference architecture based on the architectural patterns used by the reference architecture, and each of the architectural strategies specified in the reference architecture.

6.2 Examples

A number of reference architectures are widely used. Some are reference architectures for enterprise systems, but there are reference architectures for fields as disparate as vehicle control systems and gaming. Some of the more widely used reference architectures include

- Layered reference architectures for enterprise systems like *Java-EE* and Python *Django*. These reference architectures typically emphasise scalability, security, reliability, and, to a lesser extent, integrability. They introduce concepts for presentation, services and persistence layer business components. *Java-EE* has many implementing frameworks like *JBoss*, *Apache Geronimo*, and *Glassfish* as well as a range of commercial Java-EE implementations.
- The *Services-Oriented (reference) Architecture* which places the core focus on integrability, flexibility (time to market) and reuse. It is based on the microkernel pattern and introduces the concept of a stateless, discoverable and self-healing service as core concept for specifying application logic. In addition it provides an infrastructure for “*service orchestration*”. Once again, there are many implementing frameworks like *Apache Axis*, *OpenESB*, *Mule* and so on.
- *Space-Based Architectures* which focus on extreme reliability and scalability, integrability and the ability to have workers auto-orchestrate a process amongst themselves. This

reference architecture is based on the blackboard architectural pattern.

- *AUTOSAR* is a widely used reference architecture for automotive systems. It is based on the layered and microkernel architectural pattern and focuses on addressing reliability, performance, integrability and monitorability. Once again, there are many implementing frameworks, most of which are proprietary implementations of this open reference architecture.

6.3 Benefits and risks of using reference architectures

Benefits of using a reference architecture include

1. *Reduced risk*

- Community contributes to solution
- Typical architectural challenges for domain addressed
- Well understood by architects and developers
- Often enforces good standards compliance.

2. *Lower cost*

- Lower analysis and research costs.
- Implementing frameworks commonly exist.
- Lower maintenance costs because frameworks often maintained by community.
- Lower training costs for staff and shorter time to productivity.

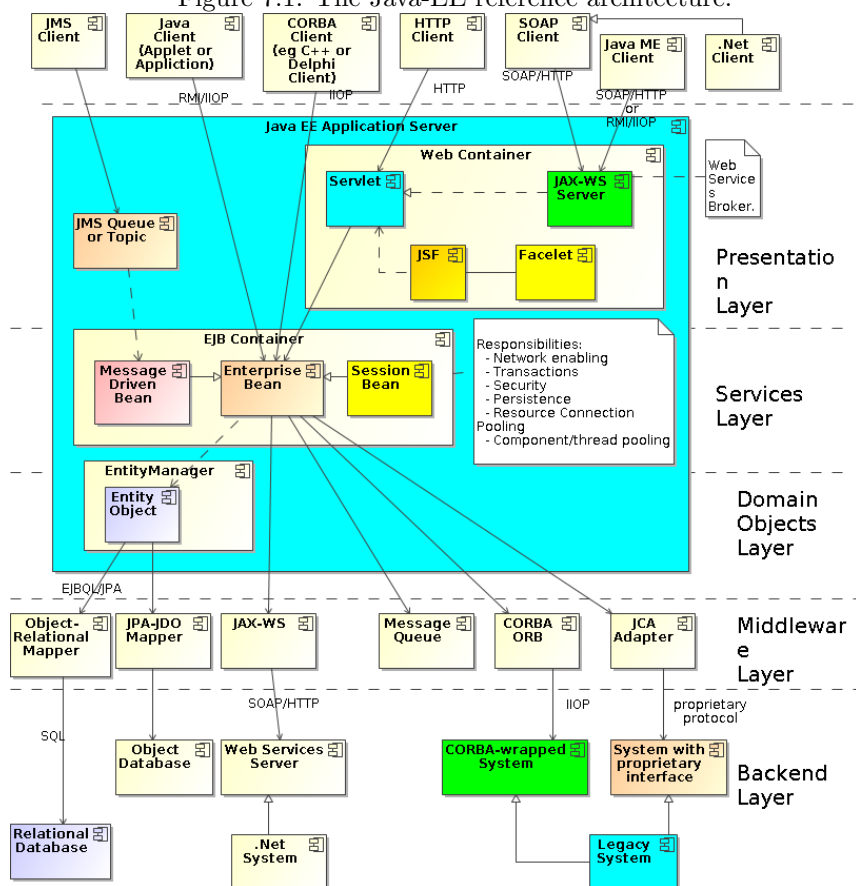
However, having a software architecture on a reference architecture also introduces some risks. In particular that of

- jumping to reference architecture prior to solid architecture analysis,
- and that it may curb innovation/competitive advantage.

Chapter 7

Java-EE: first level of granularity

Figure 7.1: The Java-EE reference architecture.



The application server is the architectural component at the first level of granularity. It is based on the *layered architectural pattern* with the following layers

1. Access layer → web container, message queues, ...
2. Business processes layer → EJB container
3. Persistence layer → Persistence context

A number of tactics or strategies are used to concretely address the quality requirements of a typical enterprise system. For example, clustering of all architectural components including all layers, JNDI repositories, load balancers, firewalls and databases is used to address scalability, availability and reliability.

At the first level of granularity no application components are hosted and hence the architecture thus does not introduce concepts or constraints for application components at this level of granularity.

The EJB container is the business logic container.

At this lower level of granularity a further range of architectural tactics is applied to address EJB container quality requirements. For example, the flyweight pattern and resource pooling (including object, thread and connection pooling) are used to address scalability and performance. Similarly interception is used to address for auditability and security, and queueing to address scalability and reliability. In addition, integrability is addressed through seamless integration through RMI, SOAP-based and REST-ful web services, and messaging.

The EJB container does host application components. It uses the controller pattern for application components and introduces stateless and stateful session beans and entities as the concepts within which applications are developed. Message driven beans should really be used for messaging adapters and are thus conceptually part of the access layer.

In addition EJB introduces a number of constraints for application components (session beans) including

- may not provide no direct access to service components
- may not directly access resources (threads, DB connections, ...)
- may not directly access native libraries

Finally EJB introduces a range of strategies to improve flexibility, reliability and extensibility including

- decoupling via contracts
- dependency injection & service provider lookup
- declarative authorization
- declarative transaction management
- interception for extensibility and customizability
- **Scalability**
 - Typically good through resource (thread, connection, object, ...) reuse, load-balancing, caching and clustering.
- **Reliability**
 - Typically good through transaction support, clustering, session replication and support for messaging.

- **Flexibility**

- Average as processes not explicit.
- Improved through interceptors, removal of infrastructure/plumbing logic, hosting process logic only in stateless session beans and annotations.

- **Performance**

- Average because of layers, and communication overheads.
- RMI/IIOP is reasonably efficient protocol.
- Improved through JPA-based object cache.

- **Auditability**

- Not directly supported. Needs to be implemented via interceptors.

- **Security**

- Good support for authentication, authorization and confidentiality.

- **Integrability**

- Quite good with support for CORBA, SOAP-based & Restless web services, DB integration.
- Integration to systems using proprietary protocol via JCA.

Chapter 8

What does the EJB container provider?

8.1 Introduction

As the EJB container is responsible for hosting shared business objects within the application server, it is responsible for transparently applying a number of enterprise and middleware services to beans. Had the developer needed these services in a traditional CORBA/DCOM or Java RMI deployment, several explicit API calls would have to be made from within the business objects. In other words, the business logic would be polluted with a number of complex technicalities.

8.2 Concurrency support

The application server automatically supports concurrent service requests via multiple bean instantiation, i.e. concurrent service requests are processed by different bean instances. Bean developers need thus not worry about writing multi-threaded servers – in fact they are forbidden to do so because that would interfere with the containers concurrency support.

Furthermore, in the case where the different client bean instances share common data resources the container takes over the responsibility of synchronizing the different data views.

8.3 Component pooling

Component pooling is possible because, as we shall see, clients do not obtain a direct reference to the enterprise bean instance.

As one deploys enterprise beans within a container, the container typically creates a pool of instances of the bean. The algorithm used is specific to the container – this is one of the many areas where application server vendors compete.

Pooling is particularly simple for stateless session beans which provide a set of client services but which do not maintain information across service requests. In this case the session bean is returned to the pool upon completion of the service and if the same or another client requests a service from that same enterprise bean one of the bean instances in the pool is allocated to the client.

Containers also provide component pooling and life-cycle management for other enterprise beans. For message-driven beans it is basically as simple as for stateless session beans. For stateful session beans and entity beans the state has to be persisted before the same bean instance can be used for another client.

If, say during peak hours, the demand for a particular enterprise bean increases, the container can dynamically increase the pool size and at a later stage, when the load decreases, the pool size can be reduced again.

A relatively small number of beans can thus serve a large number of clients.

8.4 Network enabling

Bean developers need not make the beans network-enabled. The container automatically supports distributed architectures by wrapping the bean instances by bean objects which are network enabled. These bean objects *intercept* bean service requests, in order to provide the other services like transaction, security and persistence. Every EJB is published as both an RMI and a CORBA object, and can easily (through the use of Java annotations) be published as a Web Service as well.

8.5 Persistence

The application server automatically loads bean information from persistent storage upon bean activation and saves the bean state automatically onto persistent storage upon deactivation.

Furthermore, the bean developer need not know the details of the structure of the persistent storage (e.g. which object fields are stored in which columns of which tables or whether the persistent storage is a relational or object database). The mapping to persistent storage can be done declaratively by a database administrator who need not be a Java developer. It can be even left to the EJB container to create the required database tables and to define the mapping implicitly. **Note:** *If one uses JPA based persistence, then one is locked into object-relational mapping and hence to relational databases.*

EJB also maps specialization relationships onto relational databases and allows you to choose from a set of standard mappings for specialization.

8.6 Component location transparency

EJB Application Servers must provide a naming and directory service which implements the Java Naming and Directory Interface (JNDI). The JNDI is a generic API for interfacing with general naming and directory services (such as, for example, LDAP).

As is shown can be seen in 8.1, JNDI wraps concrete naming and directory services. Some of the naming and directory services which can currently be accessed through JNDI are

- **Local File Systems:**

Files and directories

- **COSNaming:**

CORBA's standard naming service is meant to enable CORBA clients to look up a reference to a CORBA object from a name.

- **RMI registry:**

Java's RMI naming service which fulfills the same purpose as the CORBA naming service for Java RMI objects.

- **LDAP:**

The *Lightweight Directory Access Protocol* was developed in the early 1990's as a standard directory protocol which would be used by a wide range of applications. It facilitated, for example, that the particulars (e.g. personal details, phone numbers, e-mail addresses, network and device access particulars, ...) of a new employee could be entered into one central location. Applications like phone and e-mail number search applications, answering machine services, network administration applications etc. would all obtain the required information from one central LDAP. LDAP data is structured as a hierarchical database which allows multiple entries for a specific item. Sun's iPlanet directory server and OpenLDAP are the most well known LDAP servers but most other directory services provide an LDAP interface. LDAP version has support for referrals – i.e. it makes it possible that LDAP service requests are referred on to other service providers. This enables large-scale clustering and distribution.

- **DNS:**

Internet *domain name servers* which map a domain name (the host name) onto a IP address (the message path) can also be accessed through JNDI.

- **NIS:**

NIS, Sun's *Network Information Service* , which acts as a yellow pages service for network resources.

- **NDS:**

The *Novell Directory service* .

JNDI thus provides a standard interface which decouples the application from the physical naming and directory service provided by the environment. Naming services are typically used to obtain a reference to an object in a distributed environment. Directory services are really sophisticated naming services which include metadata describing the objects they reference. This enables clients to make more sophisticated searches for objects – i.e. for example to query all printers in a particular building which can print color onto A3 sized paper.

8.7 Transaction support

A transaction is a set of operations that must be processed as a single unit and if that unit was not successful in its entirety the entire transaction must be rolled back. Transaction boundaries are used as instants where object states are synchronized with the database.

The EJB application server/container provides implicit support for distributed transaction management freeing developers of the burden of either including API calls to managing transactions themselves.

8.8 Security support

When you open your systems across an intranet or even internet, security becomes of vital importance. The EJB application server facilitates a declarative support for authentication

and authorization which removes the burden of making calls to a security API from the bean developer.

The authentication is usually done at the persistence layer using normal JAAS (Java Authentication and Authorization Service) based security.

Confidentiality is usually transparently configured within the architecture resulting in secure communication (via, for example, SSL).

At the services layer, the main focus is on authorization. The bean deployer defines security roles for entire enterprise beans or for individual services supplied by enterprise beans. A security administrator maps users and user groups onto security roles.

8.9 Session management

The client session is solidly managed across presentation and services layers (the session beans). The session context and state is propagated across session beans.

8.10 Interception

Enables you to intercept both, business logic and bean management services in order to add further responsibilities around a set of base responsibilities addressed by the bean logic itself and by the application server. This makes enterprise beans externally extensible.

8.11 Resource connection pooling

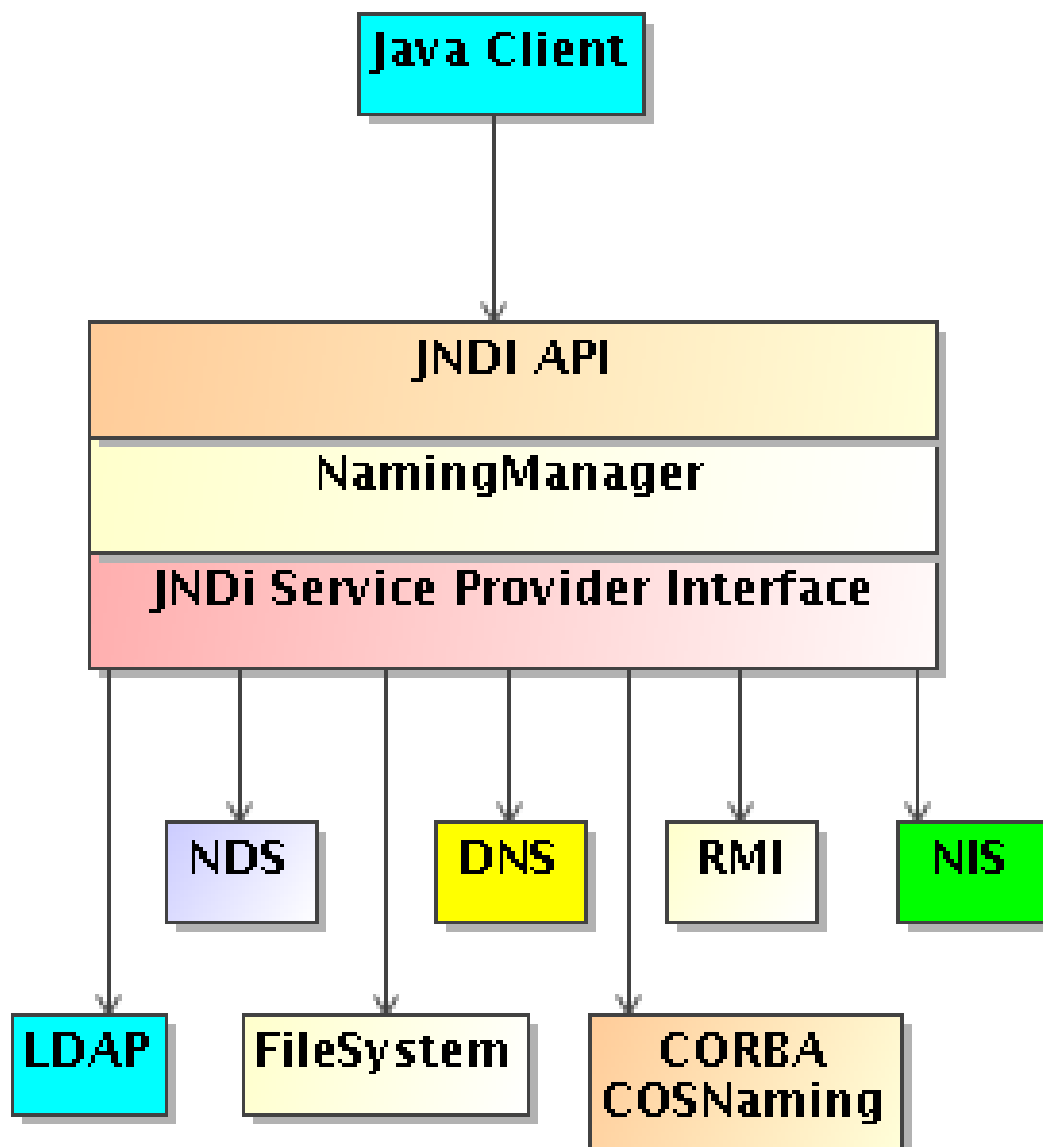
In a similar way, the container is typically responsible for resource connection pooling like, for example, database connection pooling. Establishing these connections is typically expensive. Enterprise beans should typically not establish connections themselves to resources. Instead they obtain a connection from the container who maintains a connection pool.

8.12 Implicit monitoring

EJB containers typically monitor the usage of beans in order to

- Optimize bean and thread pools.
- Perform load balancing across multiple machines
- Support reporting for administration and maintenance purposes.

Figure 8.1: The JNDI Architecture



Chapter 9

Enterprise beans

Enterprise beans are meant to be *pure server side business logic components which can be deployed on application servers within different business processes (work flows) requiring different security and transactional support, as well as different persistence mappings* . To achieve this a bean implementation should not contain any deployment information and should focus purely on business logic.

9.1 Bean species

9.1.1 Session beans

9.2 Elements of enterprise beans

9.2.1 EJBObject

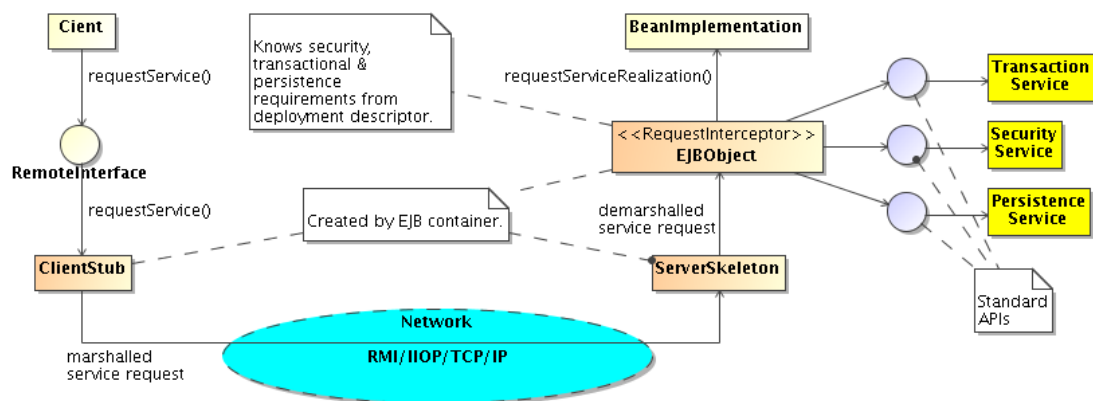
If incoming service requests were dispatched directly to a bean, all of the enterprise services provided by the EJB container would be by-passed. To this end, beans which are deployed in a EJB container are never directly accessible: The EJB container *generates* a class (which implements both the remote interface of the bean, as well as the `javax.ejb.EJBObject` interface). The EJB container uses the information specified either as Java annotations, or in XML deployment descriptor, to generate an EJBObject which enforced the qualities desired by the developer (security, transactional behaviour, web services publication, etc).

The EJB object thus intercepts the call to the enterprise bean, enabling the container to provide standard support to your beans, such as

- concurrency,
- transactions,
- security, and
- persistence.

The way in which a service request is intercepted and ultimately processed is shown in the following figure. **Note:** *Since EJB uses RMI/IIOP/TCP/IP as protocol, enterprise beans can be accessed from CORBA through the standard CORBA protocol, IIOP. This means they are directly accessibly to, for example, CORBA or C++ clients.*

Figure 9.1: Bean Service Request Processing



9.2.2 Local and remote interfaces

9.3 Bean restrictions

The container takes over a lot of responsibilities which would otherwise have resided with the bean developer. Consequently the bean itself is prevented from doing certain things which interfere with the container operations.

9.3.1 Beans can't give clients direct access to the bean instance

The client should never interface with the bean instance directly, but should instead interface only with the EJB object which is generated by the container and acts as a portal to the bean. This architecture enables the container to intercept bean service requests and take over the responsibility of container managed services like transactions, concurrency, security and persistence.

Though you can't pass a handle of the bean instance to the client, you can, of course pass the handle to bean helper classes (which are, from the client's point of view part of the bean implementation).

9.3.2 Enterprise beans may not accept network server connections

Beans should not act as servers themselves. Once again, this is the responsibility of the EJB object. The bean can, however, open client connections to other network servers (e.g. CORBA servers or other enterprise beans).

9.3.3 Enterprise beans should be single-threaded

In order to be able to effectively handle concurrent service requests from a large number clients the application server uses thread pooling. For this reason the application classes should not create their own threads. If they do want to have a piece of work done in a separate thread, then they should submit the piece of work to the application server which will assign a thread from a thread pool for processing that piece of work in a managed way.

The bean should also not use synchronization itself. This too will be handled by the application server within the transactions framework.

9.3.4 Enterprise beans should not create a user interface

The whole idea behind Enterprise Java Beans is to separate the presentation layer from the business logic layer. No direct interaction via a GUI (AWT or Swing) or keyboard input is allowed for enterprise beans. Of course, the latter follows from the fact that enterprise beans may not use the `java.io` package. Bean developers may also not assume that the bean host has any form of GUI support.

9.3.5 Stateful session beans can't have persistent class fields

The idea of maintaining persistent class information (static fields) for an EJB goes largely against OO concepts and warrants a redesign with perhaps introducing further entity beans.

There are several problems with maintaining class state for an EJB. Firstly, the container will not manage concurrent access to such fields and you as bean developer may not. Furthermore, the EJB service requests may be distributed by the container across Java Virtual Machines (JVMs) and the class state would not be available across these JVMs.

Thus, you should only use constant (`final`) class attributes for EJBs.

9.3.6 Enterprise beans may not use any native libraries

The reasons for this are firstly security and secondly portability. If you really need to obtain access to native libraries you should wrap them as a CORBA component and use them via standard CORBA service requests.

Chapter 10

Entity objects

The Java EE specification supports the concepts of entity objects, which are managed by an entity manager. Entity objects exist from when they are created until they are deleted. The entity manager manages the persistence to durable storage (some form of database).

Entity objects have persistent object identity and can be looked up on that object identity. Hence, these objects exist, from the user's side, across client sessions and server restarts. The entity manager will typically use JPA (the Java Persistence API) to interface with an object-relational mapper (such as Hibernate or Eclipselink) in order to persist the state of objects to a relational database.

Note: *Though possible (via JDBC), Java EE proposes that developers no longer work with relational databases using relational elements such as SQL and result sets. Instead, it is proposed to stay within a fully object-oriented realm, and let the infrastructure manage the mapping between the two worlds.*

Part II

Maven

Chapter 11

Introduction

Apache Maven has established itself as the primary build tool for *Java* projects. It is used to manage

- a project's build,
- project reporting, and
- project documentation

from a single project descriptor, the *Project Object Model* (POM).

11.1 Guiding principles of Maven

Apache Maven is more than just a build tool. It suggests and enforces proven work and structural patterns for developers, testers, project managers and users to be able to effectively collaborate in order to develop, test, deploy and distribute the development artifacts and to facilitate project reporting for observation/monitoring and project documentation.

Apache Maven was developed in the context of the following guiding principles:

1. Sensible defaults.
2. Encourage proven patterns including patterns for
 - file/project structure,
 - development process,
 - dependency management and installation.

making it easier for role players (developers, testers, project managers & system administrators) to move between projects.

3. Increase visibility including
 - code and component visibility for project tracking and reuse,
 - test visibility for bug tracking and fixing,
 - visibility of any other metadata like project vision, participating developers, ...
4. Effectively manage dependencies and resources

5. Integration between development teams

- local and global

11.2 What does Maven provide?

- A **Project Object Model** (POM) within which one specifies
 - project metadata with direct dependency specification,
 - build customizations with enforced unit testing,
 - reporting customizations, distribution information, ...
- **Uniform project structures** favouring convention over configuration.
- **Dependency management:** with support for resolving dependency chains/webs and retrieval of dependencies from local and remote repositories.
- **Standard processes** for building, reporting, deploying, distribution, ...
- Remote and local **repositories** with versioned artifacts and metadata facilitating *universal reuse of plugins and projects artifacts*.
- A **plugin framework** which enables plugins to publish goals which can be bound to life cycle phases of either the standard built-in life cycle or to phases of its own life cycle,
- Support for **modularization of projects** with commonality inheritance from parent POM.

11.3 Maven versus Ant

- **Declarative versus operational approach:**
- **Project structure / metadata versus explicit build instructions:**
- **Convention per default versus configure all**
- **Dependency management:**
- **Project inheritance:**
- **Modularization:**
- **Project reporting and measurement:**
- **Repositories:** Global and local repositories for plugins, libraries and applications.

11.4 What is Maven really?

Maven is a *project management tool* which facilitates software project

- building,
- reporting,
- deploying, and
- distribution

in the context of a *globally integrated developer community* with *projects constructed from artifacts developed by different teams*.

Chapter 12

Maven's Project Object Model (POM)

The *Project Object Model* (POM) encapsulates the project metadata. It may contain

- the project identifier, name, version, ...
- information on the project contributors and licensing,
- the structural organization (directories) of the project,
- the project dependencies,
- relationships between this project and other projects, and
- information about the build environment for the project.

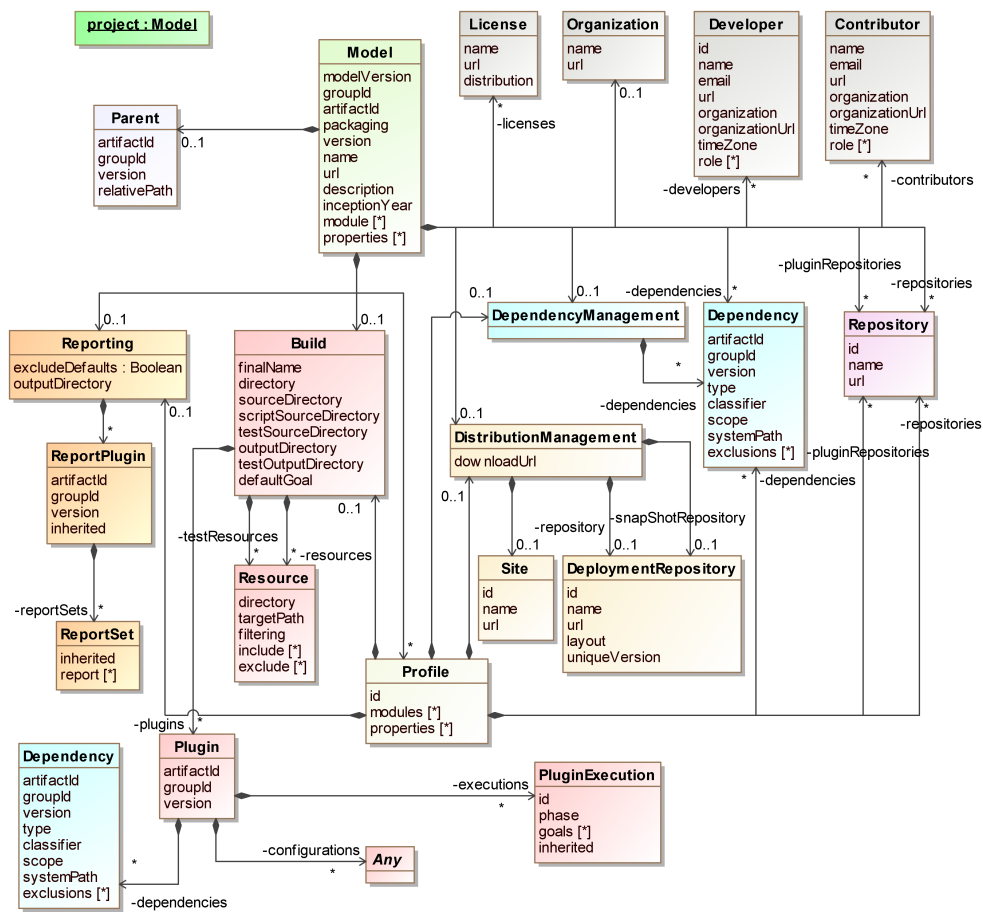
12.1 POM structure

The POM is encoded in XML and the structure of the POM is defined by an XML schema. The UML class diagram of figure 12.1.

At the high level the project object model may specify

- a project identifier assembled from artifactId, groupId and version,
- the version of the POM model used,
- general project metadata like project name and description, licensing and contributor information and information about the organization which claims ownership of the project,
- relationships the project has with other projects,
- the project dependencies,
- repositories from which dependencies and plugins are to be sourced,
- build information,
- information on how the outputs need to be distributed,

Figure 12.1: The structure of Maven's POM



- information of different build profiles like development, testing and production profiles,
- information on how the project measurement and reporting should be done,
- as well as a collection of properties which can be used throughout the POM.

12.2 Project Identifiers

Projects artifacts, their dependencies on other projects and plugins are uniquely identified by *project coordinates* which take the form of

```
1 <groupId>:<artifactId>:<packaging>:<version>
```

Strictly speaking the packaging itself is not part of the project identifier, but it is still used to source the appropriate packaged version of a resource.

For example, a minimal POM

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-  
  instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-  
  v4.0.0.xsd">  
2   <modelVersion>4.0.0</modelVersion>  
3  
4   <groupId>za.co.solms.mavenCourse</groupId>  
5   <artifactId>simplestExample</artifactId>  
6   <packaging>jar</packaging>  
7   <version>1.0-SNAPSHOT</version>  
8  
9   <name>simple</name>  
10  <url>http://www.solms.co.za</url>  
11 </project>
```

which has as project identifier

```
1 za.co.solms.mavenCourse:simplestExample:1.0-SNAPSHOT
```

It would, by default, be packaged in a jar named

```
1 simplestExample-1.0-SNAPSHOT.jar
```

12.2.1 Group Identifier

The `groupId` is to identify the group or organization which takes ownership of the project. The convention is to use the domain name of the group/organization in reverse.

12.2.2 Artifact Identifier

The `artifactId` is a project identifier which should be unique within the group or organization which owns the project.

12.2.3 Version

This represents an identifier for the version of a project. During active development the version is usually designated as a `SNAPSHOT` version.

12.2.4 Packaging

This specifies the packaging to be used. Examples include `jar`, `war`, `ear`, `pom`, and `zip`.

12.3 POM inheritance

In order to ensure some level of uniformity across projects you typically will want to define common project parameters in a parent POM which will be inherited across the various child POMs.

12.3.1 Declaring the parent POM

A POM declares another POM as its parent by specifying the project identifier and optionally a relative path. For example

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi
  :schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
2   <modelVersion>4.0.0</modelVersion>
3
4   <parent>
5     <groupId>za.co.solms.mavenCourse</groupId>
6     <artifactId>simpleParent</artifactId>
7     <version>1.0</version>
8   </parent>
9
10  <groupId>za.co.solms.mavenCourse</groupId>
11  <artifactId>simplestExample</artifactId>
12  <packaging>jar</packaging>
13  <version>1.0-SNAPSHOT</version>
14
15  <name>simple</name>
16  <url>http://www.solms.co.za</url>
17 </project>

```

A child project will inherit all definitions defined in the parent project and will only have to either

- add additional definitions which are not inherited,
- override inherited definitions with its own definitions.

Note: *The inheritance is not at class but at instance level. Thus, a particular instance of a POM inherits from another instance of a POM which has been declared it's parent POM.*

12.3.2 The Super POM

Maven enforces a common ultimate parent POM from which all projects inherit. This POM is called the Super POM and comes packaged with any Maven installation.

The Super POM encapsulates some defaults shared across all projects including

- The default remote Maven repository from which dependencies are, by default, obtained.
- The default remote Maven plugin repository from which plugins are sourced.
- Default build information including
 - a default directory structure for Maven projects including the source and output directories for the code and tests which are set to

```

1 src/main/java
2 src/test/java
3 src/main/scripts
4 target
5 target/test-classes

```

- the default target directory and default build file name which is

```
1 ${pom.artifactId}-${pom.version}
```

which is ultimately appended with the packaging type.

- A default list of plugins which are available for the default Maven build cycle including plugins for
 - compiling,

- creating jars, wars, ears, rars and javadoc,
 - resolving dependencies,
 - deploying and installing, and
 - cleaning the project.
- the default report output directory which is set to

```
1 target/site
```

12.3.3 The effective POM

The effective POM is the sum-total of all inherited POM elements, some of which may be over-written at some level of the parent-child hierarchy for the POM. You can use the **effective-pom** goal of the *help* plugin to query the effective POM for a specific child/concrete POM by executing

```
1 mvn help:effective-pom
```

from within the directory which contains the child POM.

12.4 Project dependencies

The POM requires that the direct dependencies of a project are explicitly specified. Project dependencies may be either

- *external dependencies* on libraries or other resources developed by other groups or organizations,
- libraries or other resources developed within other internal projects.

12.4.1 Specifying project dependencies

12.4.1.1 Specifying project dependencies

Project dependencies are specified in a separate dependencies section. The dependency is identified through the standard

```
1 groupId:artifactId:version
```

project identifier:

```
1 <project>
2   ...
3   <dependencies>
4     <dependency>
5       <groupId>solms.co.za.mavenCourse</groupId>
6       <artifactId>testLibrary</artifactId>
7       <version>1.4.2</version>
8     </dependency>
9   </dependencies>
10  ...
11 </project>
```

12.4.1.1.1 Specifying version ranges Instead of locking the dependency into a particular version, Maven allows you to specify a dependency on a version range, specifying the lowest and highest version numbers which would be acceptable. This can be done inclusively via square brackets [1.51,1.9] or exclusively via round brackets (1.5,2).

You can leave one boundary open ended. For example,

```
1 <dependency>
2   <groupId>junit</groupId>
3   <artifactId>junit</artifactId>
4   <version>[,4)</version>
5 </dependency>
```

defines a dependency on any version of JUnit up to, but excluding version 4.0.

12.4.1.1.2 Specifying the scope of a dependency By default dependencies are available in all classpaths for all build phases and are packaged within the resultant archive. Maven allows you, however, to reduce the scope of a dependency via the **scope** attribute which can have the following values

- **compile:** The **compile** scope is the default scope. A dependency which is assigned the **compile** scope is available in all class paths and is included in the package for the project (main) artifact. For example, a library which you are compiling against and whose classes are required at run time would typically be specified using the

```
1 <project>
2   ...
3   <dependencies>
4     <dependency>
5       <groupId>solms.co.za.mavenCourse</groupId>
6       <artifactId>testLibrary</artifactId>
7       <version>1.4.2</version>
8       <scope>compile</scope>
9     </dependency>
10  </dependencies>
11  ...
12 </project>
```

- **provided:** The **provided** dependency is used to specify a dependency which would be provided by the deployment/execution environment for the project artifact. For example, the EJB, JTA, JPA, ... APIs would be provided by the application server into which an application is to be deployed into. Dependencies with **provided** scope are still included in the class paths for compilation purposes, but are not packaged within the resultant project artifact.

```
1 <project>
2   ...
3   <dependencies>
4     <dependency>
5       <groupId>javax.servlet</groupId>
6       <artifactId>jervlet-api</artifactId>
7       <version>[2.2,2.3)</version>
8       <scope>provided</scope>
9     </dependency>
10  </dependencies>
11  ...
12 </project>
```


- **runtime:** Dependencies specified with `runtime` scope are required during execution and testing, but not for compilation. For example, you might need a particular API/contract jar at compile time, but the actual implementation classes for that API only at run time.
- **test:** Dependencies declared with `test` scope are only required for the compilation and execution of tests. For example, a JUnit dependency would typically be scoped as a *test* dependency:

```

1 <project>
2   ...
3   <dependencies>
4     <dependency>
5       <groupId>junit</groupId>
6       <artifactId>junit</artifactId>
7       <version>[4,]</version>
8       <scope>test</scope>
9     </dependency>
10  </dependencies>
11  ...
12 </project>

```

- **system:** The `system` scope is meant to be used to specify native system dependencies. If you declare a dependency with `system` scope, you will have to specify the `systemPath` element. System scope should be used only in very exceptional cases.

12.4.2 Transitive dependencies

Maven resolves transitive dependencies, i.e. recursively dependencies of dependencies. Maven does this by building a dependency graph, resolving any conflicts if possible.

For example, your project may have a dependency on the HP *Jena* framework for processing RDF and OWL from Java. *Jena* itself depends on *jena.arq* which is its implementation of the SparQL query language for semantic knowledge repositories. *jena.arq* in turn depends on projects like *Lucene* and *Xerces* and so on. Maven resolves the recursive dependencies and includes them in the application assembly.

12.4.3 Dependency management

Across the various projects and modules one may accumulate dependencies across large version ranges of other projects. One often would like to standardize the dependencies to particular versions and then manage a controlled process to evolve ones dependencies to later versions. This one does not, however, want to do project for project or even module for module. It is for this purpose that Maven introduces the concept of dependency management.

Dependencies themselves are, of course, not inherited by child projects. One can, however, define in the `<dependencyManagement>` section of the parent project the preferred versions for libraries which are used across child projects. In the child project one still needs to specify the dependency on a particular library, but one can omit to specify a version. In such cases the version will be determined by the version specified in the `<dependencyManagement>` element of the parent project.

Consider, for example, the following top-level POM used by an organization:

```

1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>za.co.solms</groupId>
4   <artifactId>approvedLibs</artifactId>

```

```

5  <version>2.1</version>
6  ...
7  <dependencyManagement>
8    <dependencies>
9      <dependency>
10        <groupId>junit</groupId>
11        <artifactId>junit</artifactId>
12        <version>4.0.1</version>
13      </dependency>
14      <dependency>
15        <groupId>com.hp.hpl.jena</groupId>
16        <artifactId>jena</artifactId>
17        <version>2.6.0</version>
18      </dependency>
19    </dependencies>
20  </dependencyManagement>
21  ...
22 </project>

```

All projects within the organization could now inherit the approved versions for standard libraries used within the organization from the parent POM. The child POMs for the actual projects would not specify the version numbers for these standard libraries (except if they need to override the approved version numbers with project specific version numbers):

```

1  <project>
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>za.co.solms</groupId>
4    <artifactId>courseNotesAssembly</artifactId>
5    <version>0.8-SNAPSHOT</version>
6    ...
7    <dependencies>
8      <dependency>
9        <groupId>com.hp.hpl.jena</groupId>
10       <artifactId>jena</artifactId>
11     </dependency>
12   </dependencies>
13   ...
14 </project>

```

12.4.4 Exclusions

At times one would like to explicitly exclude particular library versions. The reason for that would be to resolve a conflict between two dependencies caused typically by a transitive dependency being on a different version of a library than what is required by the project itself. This is typically caused by an unnecessary locking into a particular version, i.e. when a specific version is specified in a dependency but the dependency should have really been a dependency range as later and/or earlier versions of that library would be substitutable.

For example, the following POM excerpt specifies a dependency on a *testLibrary*, but

```

1  <project>
2    ...
3    <dependencies>
4      <dependency>
5        <groupId>solms.co.za.mavenCourse</groupId>
6        <artifactId>testLibrary</artifactId>
7        <version>1.4.2</version>
8        <exclusions>
9          <exclusion>
10            <groupId>xerces</groupId>
11            <artifactId>xercesImpl</artifactId>
12          </exclusion>
13        </exclusions>
14      </dependency>
15    </dependencies>

```

```

16 ...
17 </project>

```

12.5 Project modules

Whilst project inheritance is used to inherit metadata from parent projects, modules are used to specify a part-of or aggregation relationship between projects. This is used to aggregate builds of sub-projects into a single higher-level build which builds all sub-projects of a project.

One specifies a POM for the aggregate object which lists a number of sub-projects as modules. These sub-projects are, however, not identified via the standard groupId-artifactId-version identifiers but are assumed to be of the same groupId. The module name maps onto the artifactId of the sub-project and the sub-project needs to be contained in a sub-directory of the directory for the aggregate project.

Whilst the inheritance relationship is specified from the child projects perspective, aggregation is done from the perspective of the aggregate projects with the sub-projects being unaware that they are modules of an aggregate project.

For example, one could specify different modules for the web client, swing client, web services portal, business logic layer and the reporting module of an enrollment system:

```

1 <project>
2   <modelVersion>4.0.0</modelVersion>
3
4   <groupId>za.co.solms</groupId>
5   <artifactId>enrollmentSystem</artifactId>
6   <version>2.3</version>
7
8   <modules>
9     <module>webClient</module>
10    <module>swingClient</module>
11    <module>webServicesPortal</module>
12    <module>businessLogicLayer</module>
13    <module>reporting</module>
14  </modules>
15  ....
16 </project>

```

these sub-projects would be contained in corresponding sub-directories of the enrollment system base directory. They would each have their own POM which could be unaware of the higher-level aggregate enrollment-system project.

The aggregate project need not be the parent of the sub-projects. It is, however, very common to declare the aggregate project the parent project of the sub-projects as that enables one to specify commonalities across sub-projects and in particular introduce some uniformity around the version dependencies via the *dependencyManagement* construct.

12.6 Build customization

The build section of the POM is used to customize the standard Maven build life cycle. The customization usually involves adding additional build steps to the standard build life cycle for the artifact type built by the project.

One typically defines for a custom build step

- the source and output directories for the sources and tests,
- the directories containing any resources required for the build or for the tests,

- the plugins to be used, the goals (services) to be executed and the build phase to which they should be bound, and any dependencies and configurations.

12.6.1 Customizing/configuring plugin behaviour

One can use the build section of the POM to customize the behaviour of a defined goal. This can be done in the **configuration** sub-element for the **plugin** element of the POM.

For example, it is relatively common to have to customize the behaviour of the *clean* plugin, having to specify additional directories and/or file patterns which need to be removed during the clean phase.

```

1 <project>
2   ....
3
4   <plugins>
5     <plugin>
6       <artifactId>maven-clean-plugin</artifactId>
7       <configuration>
8         <filesets>
9           <fileset>
10            <directory>temp/generatedClasses</directory>
11            <includes>
12              <include>*.java</include>
13            </includes>
14          </fileset>
15        </filesets>
16      </configuration>
17    </plugin>
18  </plugins>
19 </project>

```

12.6.2 Adding a goal to a life cycle phase

For example, in order to add a step which creates Java classes from XML schemas one can specify a build customization which requests the **generate** goal of the *jaxb2* plugin to generate java classes from all XML schemas located in a **src/main/resources/schemas** directory.

In order to be able to use the Java 6 built-in JAXB support we specify the source and target Java version for the Maven compile step to 1.6.

Finally we also need to include the repositories from which the jaxb2 libraries can be sourced from. The resultant POM is shown below:

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/maven-v4_0_0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>za.co.solms.example</groupId>
8   <artifactId>jaxb-maven-sample-java6</artifactId>
9   <packaging>jar</packaging>
10  <version>0.1</version>
11
12  <name>JAXB / Maven Sample (Java 6+)</name>
13
14  <description>
15    A Sample Maven 2.x project that illustrates usage of the
16    JAXB Maven plugin to compile XML Schema resources to Java.
17    This is suitable for isolated work on XML documents. For
18    web services, usage of the JAX-WS is recommended.
19  </description>
20

```

```

21 <developers>
22   <developer>
23     <organization>Solms TCD</organization>
24     <organizationUrl>http://www.solms.co.za/</organizationUrl>
25     <email>info@solms.co.za</email>
26   </developer>
27 </developers>
28
29 <build>
30   <plugins>
31     <!-- Configuration to compile all schemas in the
32          resources/schemas directory. Automatically
33          invoke during the 'generate-sources' phase -->
34     <plugin>
35       <groupId>org.jvnet.jaxb2.maven2</groupId>
36       <artifactId>maven-jaxb2-plugin</artifactId>
37       <executions>
38         <execution>
39           <goals>
40             <goal>generate</goal>
41           </goals>
42         </execution>
43       </executions>
44       <configuration>
45         <schemaDirectory>
46           src/main/resources/schemas
47         </schemaDirectory>
48         <schemaIncludes>
49           <include>*.xsd</include>
50         </schemaIncludes>
51       </configuration>
52     </plugin>
53     <!-- Assume a Java SE 6 environment, which includes JAXB.
54          For Java 5, extra dependencies on the JAXB Implementation
55          should be specified -->
56     <plugin>
57       <groupId>org.apache.maven.plugins</groupId>
58       <artifactId>maven-compiler-plugin</artifactId>
59       <configuration>
60         <source>1.6</source>
61         <target>1.6</target>
62       </configuration>
63     </plugin>
64   </plugins>
65 </build>
66
67 <dependencies>
68   <dependency>
69     <groupId>junit</groupId>
70     <artifactId>junit</artifactId>
71     <version>[4.1,</version>
72     <scope>test</scope>
73   </dependency>
74 </dependencies>
75
76 <repositories>
77   <repository>
78     <id>maven2-repository.dev.java.net</id>
79     <name>Java.net Maven 2 Repository</name>
80     <url>http://download.java.net/maven/2</url>
81   </repository>
82 </repositories>
83 <pluginRepositories>
84   <pluginRepository>
85     <id>maven2-repository.dev.java.net</id>
86     <url>http://download.java.net/maven/2</url>
87   </pluginRepository>
88 </pluginRepositories>
89
90
91 </project>

```

12.7 Distribution Information

One can include distribution information in the `distributionManagement` section of the project's object model. This provides information which is typically used for deployment or to make the resultant artifact available for other projects.

In addition one can specify the URL which should be used by other projects when downloading the deployed artifact. This is done in the `downloadURL` element. Maven itself adds a *status* element which specifies the status of the distribution.

12.7.1 Specifying distribution Info

The distribution information contains an id, name and url of the distribution repositories for the project's artifacts as well as a flag whether unique versions should be created by adding a time stamp to the file names. The distribution management section allows for a main and a snapshot repository.

For example,

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven
  -4.0.0.xsd">
2   ...
3   <distributionManagement>
4     <repository>
5       <uniqueVersion>false</uniqueVersion>
6       <id>solmsMain</id>
7       <name>Solms Main Repository</name>
8       <url>scp://solms.co.za/repository/main</url>
9       <layout>default</layout>
10    </repository>
11    <snapshotRepository>
12      <uniqueVersion>true</uniqueVersion>
13      <id>solmsDev</id>
14      <name>Solms Development Repository</name>
15      <url>sftp://propellers.net/maven</url>
16      <layout>default</layout>
17    </snapshotRepository>
18    ...
19  </distributionManagement>
20  ...
21 </project>

```

specifies a main and a development repository. The development repository adds time stamp information to the artifact names in order to ensure unique versions.

12.7.2 Specifying login credentials

Obviously one should not distribute the server login credentials (username, password, path to private key, ...) with the POM. These should remain on the build server and are specified in the `settings.xml` file:

```

1 <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
4     http://maven.apache.org/xsd/settings-1.0.0.xsd">
5   ...
6   <servers>
7     <server>
8       <id>solms.co.za</id>
9       <username>myUserName</username>
10      <password>myPassword</password>

```

```
11     <privateKey>${user.home}/.ssh/id_dsa</privateKey>
12     <passphrase>myPassphrase</passphrase>
13     <filePermissions>664</filePermissions>
14     <directoryPermissions>775</directoryPermissions>
15 </server>
16 </servers>
17 ...
18 </settings>
```

The permissions are used when creating new files and directories, i.e. these are the permissions to be used for any new files.

Chapter 13

Maven's life cycles

Maven provides a declarative approach to project builds, specifying the build requirements in the projects object model (POM) and then using this information to build, clean or document the project. To this end Maven defines three standard life cycles which are not plugin specific:

- The *build life cycle* is used to compile, test, package, install and deploy a project.
- The *clean life cycle* is used to remove any generated directories and files so that only the sources remain.
- The *site life cycle* measures the project (e.g. collect results of test) and generates a documentation site which contains general project metadata and reporting on project status.

The project is built following a life cycle which is a pipeline of phases which is executed sequentially. Maven defines a standard build life cycle, which is generally followed when building a project. The default build life cycle contains the following core phases (this is not a complete list of phases):

1. validate
2. compile
3. test
4. package
5. deploy
6. install

The build process follows the phases in the order of the life cycle, executing any goals/operations which are bound to these phases.

Plugin goals are bound to life cycle phases in one of the following ways

1. The plugin for the project packaging (project type) binds goals/operations to its life cycle phases which are either phases of one of the standard life cycles or phases of its custom life cycle.
2. In your build customization you can bind different goals/operations to phases of either of the three life cycles.

The life cycle which is used is determined by the plugin for the specified package type (e.g. jar or war). A plugin may hence choose to either

- bind goals to the standard life cycle phases,
- inherit a standard life cycle and add phases, or
- define a new life cycle with its own phases.

In either case, the plugin will bind its operations (goals) to life cycle phases which are either phases of its own life cycle or phases of the standard build life cycle.

13.1 Maven's default build life cycle

Maven defines a default build life cycle. This is a generic life cycle which is abstract enough to work for the vast majority of builds. Instead of modifying the life cycle itself, different plugins and project may bind different goals/operations to the phases of the default life cycle.

Maven's default build life cycle contains the following phases:

1. **validate:** The *validate* phase is meant to be used to validate that the project has all information required to build the project.
2. **generate-sources:** The *generate-sources* phase is used if sources need to be generated from other artifacts. This can include goals like generation of Java classes from XML schemas and the generation of code from a design model.
3. **process-sources:** The *process-sources* phase is used to pre-process the sources before compilation. This could be used, for example, for filtering and obfuscation.
4. **generate-resources:** The *generate-resources* phase is meant to be used if one needs to generate any resources which should be included in the final assembly.
5. **process-resources:** The *process-resources* phase for any processing of the generated resources including copying, renaming, setting permissions and so forth.
6. **compile:** During the *compile* phase the source code is meant to be compiled.
7. **process-classes:** The *process-classes* phase is meant for any post-compile processing like byte-code enhancements, annotation processing, and so forth.
8. **generate-test-sources:** The *generate-test-sources* phase is meant to be used if one generates test sources from other artifacts. Typically this would be done from some form of contract specifications (e.g. the UML-based services contracts coming from an URDAD analysis and design process).
9. **generate-test-resources:** The *generate-test-resources* phase can be used to generate resources for the testing. This could include generating test data.
10. **process-test-resources:** The *process-test-resources* phase is used to do any post-creation processing. This typically may involve filtering the resources or copying them into appropriate locations for packaging.
11. **test-compile:** The *test-compile* phase is meant for operations which compile the test sources.

12. **test:** The *test* phase is the phase where the actual testing of the compiled sources is to be done.
13. **prepare-package:** The *prepare-package* phase is a pre-processing phase which is meant to perform operations preparing the organization of resources for packaging before the actual packaging is done. This may include un-packaging certain packaged resources.
14. **package:** The *package* phase is meant to create the distributable package, e.g. the jar or war.
15. **pre-integration-test:** The *pre-integration-test* phase is a preparation phase for the *integration-test* phase. This phase is commonly used to set up the environment for the integration test.
16. **integration-test:** The *integration-test* phase is meant to perform the testing in an environment which mirrors the environment within which the artifact is to be deployed. It may include deploying the package into the integration testing environment.
17. **post-integration-test:** The *post-integration-test* phase is meant to be a clean-up phase which restores the environment within which the integration test was done.
18. **verify:** Whilst the *test* and *integration-test* phases verify the functionality of the package components and that they do not break the functionality offered by the environment, the *verify* phase provides the opportunity to verify whether the package meets certain quality criteria.
19. **install:** The *install* phase is meant to install the package in a local repository so that other projects which have a dependency on it can use it.
20. **deploy:** The *deploy* phase is used to copy the final package to a remote repository for sharing with other groups and projects. Usually only production-ready packages are deployed.

13.2 Maven's clean life cycle

Maven's *clean* defines the following phases

1. pre-clean
2. clean
3. post-clean

By default only the *clean* plugin's `clean` goal/operation is bound to the *clean* phase.

The *site* life cycle is used for generating project documentation. It has the following phases:

1. pre-site
2. site
3. post-site
4. site-deploy

13.3 Default goal bindings for the site life cycle

The *site* plugin binds the

- `site:site` goal to the *site phase*, and the
- `site:deploy` goal to the *site-deploy* phase.

13.4 Package-based goal bindings

The POM enables one to specify, for a project, the packaging. This identifies the main plugin which determines the build for the project type. Many plugins simply specify goal bindings to standard life cycle phases. Others may define their own life cycle and corresponding goal bindings.

For example, the *jar* *ejb* and *war* plugins all bind goals to a subset of the default build life cycle. The goal bindings for each of these package/project types are, in fact, the same except for the bindings for the package phase:

1. *process-resources* \longrightarrow `resources:resources`
2. *compile* \longrightarrow `compiler:compile`
3. *process-test-resources* \longrightarrow `resources:testResources`
4. *test-compile* \longrightarrow `compiler:testCompile`
5. *test* \longrightarrow `surefire:test`
6. *package* \longrightarrow respectively to `jar:jar`, `ejb:ejb` and `war:war`
7. *install* \longrightarrow `install:install`
8. *deploy* \longrightarrow `deploy:deploy`

13.5 Project based life cycle goals

The initial goal binding is determined by the packaging which represents the project type. Additional goal bindings can be specified on a per project or a per parent project level. In the latter case these bindings are inherited by all child projects.

The project or parent-project specific goal bindings are specified in the `<<build>>` customization element of the POM:

```

1 <project>
2   ...
3   <build>
4     <plugins>
5       <plugin>
6         <groupId>pluginGroupId</groupId>
7         <artifactId>pluginArtifactId</artifactId>
8         <executions>
9           <execution>
10            <phase>somePhase</phase>
11          </execution>
12          <goals>
13            <goal>
14              someGoal1
15            </goal>

```

```
1         <goal>
2             someGoal12
3         </goal>
4     </goals>
5
6     </executions>
7 </plugin>
8 </plugins>
9 </build>
10 </project>
```


Chapter 14

Executing Maven

The syntax for executing Maven is

```
1 mvn [option]* [plugin:goal]* [phase]*
```

specifying at least one plugin goal or life cycle phase and zero or more options.

14.1 Executing a plugin goal

When executing a plugin goal only the operation/service representing that goal for the plugin is executed. This is done via

```
1 mvn [option]* pluginName:goalName
```

For example,

```
1 mvn eclipse:eclipse
```

executes the *eclipse* goal of the *eclipse* plugin which creates an eclipse project for the Maven project.

14.2 Executing a life cycle phase

When executing Maven against a life cycle phase, all preceding phases of the chosen life cycle are executed. The life cycle which is used is determined by the main plugin which is chosen on the package type for the project (e.g. jar or war). It may be either one of the standard life cycles (default build, clean or site) or its own custom life cycle. In either case, the plugin will bind its own goals as well as the goals of other plugins to the life cycle phases.

Maven thus

1. determines the package type from the POM,
2. finds the corresponding plugin for that package type,
3. obtains from the plugin the life cycle phases and the goals/operations bound to each of the phases up to the requested life cycle phase, and

4. executes the life cycle phases up to the requested phase in sequence by executing any bound plugin goals for each of the life cycle phases.

For example,

```
1 maven test
```

for a `jar` project executes all phases prior to and including the test phase, i.e. all goals which were bound to these build cycle phases.

In general it is preferable to execute life cycle phases over plugin goals as the dependencies for the build will be resolved.

Chapter 15

Maven repositories

The Maven repository is a central component of Maven, facilitating universal reuse of project artifacts including Maven plugins.

A Maven repository is simply a storage space for reusable artifacts generated from various projects and the Project Object Model (POM) describing how the artifacts can be built and what the artifact dependencies are.

15.1 Repository structure

The Maven repository is simply a file system with a tree structure conforming to

```
1 <groupId>.<artifactId>.<version>
```

hierarchy. That directory will contain

- the actual artifact,
- the POM for the project with which the artifact was created,
- and a hashing key which is used to verify that the received artifact is not corrupted.

15.2 Repository locations

Maven uses both, remote and local repositories. The local and remote repositories are both structured in the same way and hence can be processed using the same logic. Furthermore, since the local and remote repositories have the same structure, they can be simply synchronized.

15.2.1 Remote repositories

The remote repositories one uses typically include

- the central Maven repository containing the core maven plugins and globally published maven projects,
- an internal remote repository where one publishes one's internal projects across the organization, or

- group/organization specific repositories which publish their projects separately from the central Maven repository.

There is no structural difference between internal and external remote repositories. Hence the same logic is used to process them and they can be synchronized. Typically the central repository is very large and one does not usually want to synchronize the full repository.

15.2.1.1 Accessing remote repositories

Files can be retrieved from remote repositories either via the HTTP or via a mechanism like SCP using the `file://URL` protocol. In the former case the directory needs to be, of course, served by the web server. Security can be done via HTTPS or simply by restricting the user access to the directory.

Files can be uploaded to a repository using SCP, FTP or other file copy mechanisms.

The location of the official central Maven artifact and plugin repositories are specified in the Super-POM:

```

1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <name>defaultProject</name>
4
5   <repositories>
6     <repository>
7       <id>central</id>
8       <name>Maven Repository Switchboard</name>
9       <layout>default</layout>
10      <url>http://repo1.maven.org/maven2</url>
11      <snapshots><enabled>>false</enabled></snapshots>
12    </repository>
13  </repositories>
14
15  <pluginRepositories>
16    <pluginRepository>
17      <id>central</id>
18      <name>Maven Plugin Repository</name>
19      <url>http://repo1.maven.org/maven2</url>
20      <layout>default</layout>
21      <snapshots><enabled>>false</enabled></snapshots>
22    </pluginRepository>
23  </pluginRepositories>
24
25  ...
26 </project>

```

You can add further repository and plugin repository specifications in either the POM for your project or in a common parent POM for your projects.

15.2.2 The local repository

The local repository

- acts as a cache for remote repositories, and
- hosts the artifacts of local projects.

It has the same structure as the remote repositories and can be synchronized with them.

The location of your local repository is

```
1 ${home}/.m2/repository
```

15.3 How Maven uses Respositories

When Maven resolves dependencies, it sources the metadata (POM) for the dependency from either the local cache repository or one of the remote repositories. From the POM it obtains the transitive dependencies. This is done recursively building up a dependency tree.

Maven will attempt to resolve any conflicts and then will source the actual artifacts for the dependencies, looking first in the local cache before trying to source them remotely. The integrity of any sourced dependency is validated using the hash which is also obtained from the repository.

15.4 Repository tools

To create a Maven repository, it is sufficient to simply create the appropriate directory structure containing the artifacts, poms and hash keys and then to serve the file system by some mechanism (typically via HTTP).

However, Maven repositories typically provide additional functionality like

- searching,
- providing tools for conveniently navigating the repository, and
- publishing the metadata in a convenient way.

Chapter 16

Hello World via Maven

A hello-world application is commonly used to test and demonstrate an infrastructure without introducing any significant functionality. The purpose is to start getting comfortable with the environment and tools.

16.1 Creating a project via the Archetype plugin

When creating a project via the `generate` goal of the *archetype* plugin, one needs to minimally specify the `groupId` and `artifactId`:

```
1 mvn archetype:generate -DgroupId=za.co.solms.training.maven
2   -DartifactId=helloWorld
```

Maven will ask you which archetype it should generate. You will see that there are templates for a wide variety of project types like, for example, JavaEE, JSF and Spring projects. For a basic Java application you can use `maven-archetype-quickstart` which is the default option.

This generates a project directory of the same name as the artifact identifier defines a POM with the corresponding group and artifact identifiers using the default `jar` packaging and defaulting the version to `1.0-SNAPSHOT`. It defaults the project name to the artifact id and sets the url to `http://www.maven.org`. In addition Maven adds a dependency on JUnit:

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/maven-v4_0_0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>za.co.solms.training.maven</groupId>
7   <artifactId>helloWorld</artifactId>
8   <packaging>jar</packaging>
9   <version>1.0-SNAPSHOT</version>
10  <name>helloWorld</name>
11  <url>http://maven.apache.org</url>
12  <dependencies>
13    <dependency>
14      <groupId>junit</groupId>
15      <artifactId>junit</artifactId>
16      <version>3.8.1</version>
17      <scope>test</scope>
18    </dependency>
19  </dependencies>
20 </project>
```

In addition to the POM, the create **generate** of the *archetype* plugin creates the expected directory structures for the sources

```

1  -src
2    -main
3      -java
4        -za
5          -co
6            -solms
7              -training
8                -maven
9                  *App.java

```

and test sources of the project:

```

1  -src
2    -test
3      -java
4        -za
5          -co
6            -solms
7              -training
8                -maven
9                  *AppTest.java

```

as well as the source files for a minimal (hello-world) application and its corresponding test application.

16.2 Executing default life cycle phases

One can now go ahead and execute any of the default life-cycle phases. For example,

```
1 mvn compile
```

will execute all life cycle phases up to and including the compile phase. The compiled classes are stored in

```
1 target/classes
```

Executing the compile phase will result in executing all phases prior to the compile phase in sequential order. This includes, for example, the **process-resources** phase which attempt to copy any resources from

```
1 src/main/resources
```

into the target

```
1 target/classes
```

so that they are in the class path and that they will be included in the packaging.

Similarly we can use

```
1 mvn test
```

to run the stages up to and including that of compiling and executing the test programs. The test results are saved in

```
1 target/surefire-reports
```

To execute the life cycle phases up to and including that of creating the jar one can execute

```
1 mvn package
```

The resultant jar package is simply stored in

```
1 target
```

If you run

```
1 mvn install
```

it will execute all the preceding phases including the package phase and will then go ahead and install the jar into the local repository.

16.2.1 Deploying onto a server

The last phase of the default build life cycle is the deploy phase. In order to be able to execute the deploy phase one needs to set up the distribution information in the POM and specify the authentication details in a separate `settings.xml` which is not distributed when the project is deployed.

16.2.1.1 Specifying distribution information

In order to be able to deploy the jar onto a remote server, the server details need to be specified in the `distributionManagement` section of the POM:

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi
  :schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
2   <modelVersion>4.0.0</modelVersion>
3
4   <groupId>za.co.solms.training.maven</groupId>
5   <artifactId>helloWorld</artifactId>
6   <packaging>jar</packaging>
7   <version>1.0-SNAPSHOT</version>
8
9   <name>helloWorld</name>
10  <url>http://maven.apache.org</url>
11
12  <dependencies>
13    <dependency>
14      <groupId>junit</groupId>
15      <artifactId>junit</artifactId>
16      <version>3.8.1</version>
17      <scope>test</scope>
18    </dependency>
19  </dependencies>
20
21  <distributionManagement>
22    <snapshotRepository>
23      <uniqueVersion>true</uniqueVersion>
24      <id>solmsRepository</id>
25      <name>Solms Internal Repository</name>
26      <url>scp://solms.co.za/var/maven/repository</url>
27    </snapshotRepository>
28  </distributionManagement>
29
30 </project>
```

16.2.1.2 SSH key for login without password

It is common for systems to have to log automatically into an ssh server without requesting a user to provide a password. This is, for example, required for cron-scheduled backups via rsync or even via scp and for automated deploys within build scripts.

This can be achieved by generating a private-public key pair and appending the public key to the server's authorized keys.

To do this, log into the client machine (the one which should log in and generate a pair of authentication keys specifying the encryption algorithm to be used (here DSA)):

```

1 ssh-keygen -t dsa
2 Generating public/private dsa key pair.
3 Enter file in which to save the key (/home/clientUser/.ssh/id_dsa):
4 Enter passphrase (empty for no passphrase):
5 Enter same passphrase again:
6 Your identification has been saved in /home/clientUser/.ssh/id_dsa.
7 Your public key has been saved in /home/clientUser/.ssh/id_dsa.pub.
8 The key fingerprint is:
9 4e:c4:73:12:a0:5e:de:40:85:df:4d:bc:1c:f5:15:b3 clientUser@clientMachine
10 The key's randomart image is:
11 +--[ DSA 1024]-----+
12 |o+o...oo|
13 |o.. .+.+|
14 |.o.=..+ o E.|
15 |.o+.+.+ |
16 |..S |
17 |o |
18 |. |
19 ||
20 ||
21 +-----+

```

Now use ssh to create, on the server, a `.ssh` in the user's home directory:

```

1 ssh serverUser@serverMachine mkdir -p .ssh
2 Password:

```

where `serverMachine` is the URL of the server machine. It will still ask for the user's password.

Finally append the client's public key to `authorized_keys` file in the `.ssh` directory in the server user's home directory. You will be once again asked for the password, but this is the last time:

```

1 cat .ssh/id_dsa.pub | ssh serverUser@serverMachine 'cat
2 >> .ssh/authorized_keys'
3 Password:

```

From now onwards the `clientUser` can log into the `serverMachine` under user `serverUser` without having to supply a password. You can test this via

```

1 ssh serverUser@serverMachine

```

16.2.1.3 Specifying authentication settings

Finally the authentication settings can be added to the `settings.xml` file, enabling the deploy process to authenticate itself via the public key:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"

```



```

3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
5   http://maven.apache.org/xsd/settings-1.0.0.xsd">
6
7   <servers>
8     <server>
9       <id>repository</id>
10      <username>fritz</username>
11      <privateKey>~/.ssh/id_dsa</privateKey>
12    </server>
13  </servers>
14
15 </settings>

```

16.3 Executing specific plugin goals

When building a project you will generally want to execute Maven specifying the target life cycle phase. This ensures that all preceding phases are executed so that all requirements for your target phase are met. Maven will execute any plugin goals which are bound to the executed phases of the resolved life cycle. However, at times you may want to execute a particular plugin goal. The reasons for this may be that you want to

- execute a goal which is not bound to any of the default life cycle phases, or
- you want to only execute a particular goal without any of the goals bound to previous life cycle phases.

For example, you could want to just execute the compiler's compile goal you can run

```
1 mvn compiler:compile
```

This will execute only the `compile` operation of the *compiler* plugin and not any other goals bound to either the *compile* phase or any of the preceding phases.

16.3.1 Executing a program from Maven

One can use the `java` goal of the *exec* plugin to execute a Java application, specifying the main class in the `exec.mainClass` parameter:

```
1 mvn exec:java -Dexec.mainClass=za.co.solms.training.maven.App
```

In a similar way one can execute a binary executable via

```

1 mvn exec:exec -Dexec.executable="ls" -Dexec.args="-l -a"
2   -Dexec.workingdir="~/temp" -Dexec.args="-arg1 -arg2"

```

For example, the following asks Maven to provide a directory listing using long info and showing the hidden files:

```

1 > mvn exec:exec
2
3   -Dexec.executable="ls" -Dexec.args="-l -a"
4 [INFO] Scanning for projects...
5 [INFO] Searching repository for plugin with prefix: 'exec'.
6 [INFO] -----
7 [INFO] Building helloWorld
8 [INFO] task-segment: [exec:exec]

```

```

 9 [INFO] -----
10 [INFO] [exec:exec]
11 [INFO] total 8
12 [INFO] drwxr-xr-x 5 fritz fritz 176 Dec 13 22:19 .
13 [INFO] drwxr-xr-x 3 fritz fritz 80 Dec 13 06:28 ..
14 [INFO] -rw-r--r-- 1 fritz fritz 936 Dec 13 21:51 pom.xml
15 [INFO] -rw-r--r-- 1 fritz fritz 470 Dec 13 22:07 settings.xml
16 [INFO] drwxr-xr-x 4 fritz fritz 96 Dec 13 14:24 src
17 [INFO] drwxr-xr-x 6 fritz fritz 216 Dec 13 06:39 target
18 [INFO] drwxr-xr-x 3 fritz fritz 72 Dec 13 22:19 ~
19 [INFO] -----
20 [INFO] BUILD SUCCESSFUL
21 [INFO] -----
22 [INFO] Total time: 1 second
23 [INFO] Finished at: Sun Dec 13 22:20:45 SAST 2009
24 [INFO] Final Memory: 5M/82M
25 [INFO] -----

```

16.4 Generating documentation

The *site* plugin can be used to generate documentation for a project:

```
1 mvn site
```

The `index.html` file for the project is stored in the `target/site` directory. The documentation can be deployed onto the project documentation server via

```
1 mvn site-deploy
```

This does, however, require that the site distribution information is provided in a `site` sub-element of the `distributionManagement` section of the POM:

```

1 <project>
2   ...
3   <distributionManagement>
4     <site>
5       <id></id>
6       <name></name>
7       <url></url>
8     </site>
9   </distributionManagement>
10 </project>

```

16.5 Cleaning the project

The *clean* plugin is used to remove all temporary and output files and directories, leaving only the sources including any resources, the POM and any settings and properties files. It is executed via

```
1 mvn clean
```

Chapter 17

Maven JAXB Sample

This simple project uses the JAXB compiler to generate the Java binding classes for an XML schema and then uses a JUnit test to execute the bindings.

This example shows

- adding further dependencies,
- adding and customizing plugins,
- specifying additional repositories and plugin repositories,
- adding further metadata to the project,
- and generating project documentation.

17.1 The schema

The schema is saved in

```
1 src/main/resources/schemas/accounts.xsd
```

It defines a few classes with relationships between them:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema
3   targetNamespace="http://example.solms.co.za/accounts"
4   xmlns:a="http://example.solms.co.za/accounts"
5   xmlns:xs="http://www.w3.org/2001/XMLSchema"
6   elementFormDefault="qualified"
7   attributeFormDefault="unqualified">
8
9   <xs:complexType name="Account" abstract="true">
10     <xs:sequence>
11       <xs:element name="balance" type="xs:double" minOccurs="0"/>
12     </xs:sequence>
13     <xs:attribute name="accountNumber"
14       type="a:AccountNumber" use="required"/>
15   </xs:complexType>
16
17   <xs:simpleType name="AccountNumber">
18     <xs:annotation>
19       <xs:documentation>
20         10-digit account number
```

```

21     </xs:documentation>
22   </xs:annotation>
23   <xs:restriction base="xs:string">
24     <xs:pattern value="\d{10}" />
25   </xs:restriction>
26 </xs:simpleType>
27
28 <xs:complexType name="CreditAccount">
29   <xs:complexContent>
30     <xs:extension base="a:Account">
31       <xs:sequence>
32         <xs:element name="minBalance" type="xs:double"
33           minOccurs="0" />
34       </xs:sequence>
35     </xs:extension>
36   </xs:complexContent>
37 </xs:complexType>
38
39 <xs:complexType name="ChequeAccount">
40   <xs:complexContent>
41     <xs:extension base="a:Account">
42       <xs:sequence>
43         <xs:element name="chequeFee" type="xs:double"
44           minOccurs="0" />
45       </xs:sequence>
46     </xs:extension>
47   </xs:complexContent>
48 </xs:complexType>
49
50 <xs:complexType name="Client">
51   <xs:sequence>
52     <xs:element name="name" type="xs:string"
53       minOccurs="0" />
54     <xs:element name="account" type="a:Account"
55       minOccurs="0" maxOccurs="unbounded" />
56   </xs:sequence>
57 </xs:complexType>
58
59 <xs:element name="client" type="a:Client">
60   <xs:annotation>
61     <xs:documentation>
62       An instance document containing a single client.
63       Accounts must be unique by account number.
64     </xs:documentation>
65   </xs:annotation>
66   <xs:key name="uniqueAccounts">
67     <xs:selector xpath="a:account" />
68     <xs:field xpath="@accountNumber" />
69   </xs:key>
70 </xs:element>
71
72 </xs:schema>

```

17.2 The Test Application

```

1 package za.co.solms.example;
2
3 import java.io.StringWriter;
4 import javax.xml.XMLConstants;
5 import javax.xml.bind.JAXBContext;
6 import javax.xml.bind.Marshaller;
7 import javax.xml.validation.Schema;
8 import javax.xml.validation.SchemaFactory;
9 import org.junit.Test;
10 import org.xml.sax.SAXException;
11 import za.co.solms.example.accounts.ChequeAccount;
12 import za.co.solms.example.accounts.Client;
13 import za.co.solms.example.accounts.CreditAccount;

```

```

14 import za.co.solms.example.accounts.ObjectFactory;
15
16 /**
17  * Illustrates XML marshalling
18  */
19 public class JAXBTest
20 {
21     @Test
22     public void testGenerateXML() throws Exception
23     {
24         // Create some data using the generated classes
25         Client client = new Client();
26         client.setName("Jack Black");
27
28         ChequeAccount a1 = new ChequeAccount();
29         a1.setAccountNumber("1007657643");
30         a1.setBalance(100.00);
31         a1.setChequeFee(0.75);
32         client.getAccount().add( a1 );
33
34         CreditAccount a2 = new CreditAccount();
35         a2.setAccountNumber("1007657642");
36         a2.setBalance(-735.18);
37         a2.setMinBalance(-10000.00);
38         client.getAccount().add( a2 );
39
40         // Using JAXB we marshal the client to XML (in this case, just to a
41         // string – we validate the output during marshalling)
42         JAXBContext ctx
43             = JAXBContext.newInstance("za.co.solms.example.accounts");
44         Marshaller marshaller = ctx.createMarshaller();
45         // Pretty-print output
46         marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
47         // Perform validation on marshalling using our schema
48         marshaller.setSchema( getAccountsSchema() );
49         // Marshal to string
50         StringWriter writer = new StringWriter();
51         marshaller.marshal( new ObjectFactory().createClient(client), writer);
52
53         // Display output
54         System.out.println( writer.getBuffer() );
55     }
56
57     /** Gets the accounts Schema */
58     private Schema getAccountsSchema()
59     {
60         {
61             try
62             {
63                 SchemaFactory sf
64                     = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
65                 return sf.newSchema(getClass().getClassLoader().getResource
66                     ("schemas/accounts.xsd") );
67             }
68             catch (SAXException e)
69             {
70                 throw new RuntimeException("Failed to read accounts schema", e);
71             }
72         }
73     }

```

17.3 Java-6 POM

JAXB is included in Java 6:

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5

```

```

6  <groupId>za.co.solms.example</groupId>
7  <artifactId>jaxb-maven-sample-java6</artifactId>
8  <packaging>jar</packaging>
9  <version>0.1</version>
10
11  <name>JAXB / Maven Sample (Java 6+)</name>
12
13  <description>
14    A Sample Maven project that illustrates usage
15    of the JAXB Maven plugin to compile XML Schema
16    resources to Java. This is suitable for isolated work
17    on XML documents. For web services, usage of
18    the JAX-WS is recommended.
19  </description>
20
21  <developers>
22    <developer>
23      <organization>Solms TCD</organization>
24      <organizationUrl>
25        http://www.solms.co.za/
26      </organizationUrl>
27      <email>info@solms.co.za</email>
28    </developer>
29  </developers>
30
31  <build>
32    <plugins>
33      <!-- Configuration to compile all schemas in the
34           resources/schemas directory. Automatically
35           invoke during the 'generate-sources' phase -->
36      <plugin>
37        <groupId>org.jvnet.jaxb2.maven2</groupId>
38        <artifactId>maven-jaxb2-plugin</artifactId>
39        <executions>
40          <execution>
41            <goals>
42              <goal>generate</goal>
43            </goals>
44          </execution>
45        </executions>
46        <configuration>
47          <schemaDirectory>
48            src/main/resources/schemas
49          </schemaDirectory>
50          <schemaIncludes>
51            <include>*.xsd</include>
52          </schemaIncludes>
53        </configuration>
54      </plugin>
55      <plugin>
56        <groupId>org.apache.maven.plugins</groupId>
57        <artifactId>maven-compiler-plugin</artifactId>
58        <configuration>
59          <source>7</source>
60          <target>7</target>
61        </configuration>
62      </plugin>
63    </plugins>
64  </build>
65
66  <dependencies>
67    <dependency>
68      <groupId>junit</groupId>
69      <artifactId>junit</artifactId>
70      <version>[4.8,]</version>
71      <scope>test</scope>
72    </dependency>
73  </dependencies>
74
75  <repositories>
76    <repository>
77      <id>maven2-repository.dev.java.net</id>

```

```
79     <name>Java.net Maven 2 Repository</name>
80     <url>http://download.java.net/maven/2</url>
81   </repository>
82 </repositories>
83 <pluginRepositories>
84   <pluginRepository>
85     <id>maven2-repository.dev.java.net</id>
86     <url>http://download.java.net/maven/2</url>
87   </pluginRepository>
88 </pluginRepositories>
89 </project>
```

17.4 Executing goals

We running the tests via `mvn test`

- any dependencies are downloaded,
- the XJC compiler is used to compile the schema, generating Java binding classes,
- all generated and written Java classes are compiled,
- and the tests are executed.

Running

```
1 mvn site
```

generates documentation for the site showing

- the project description,
- the project dependencies,
- the test results,
- the project team,

and a range of other information.

Chapter 18

Maven JAXWS Sample

This is an example of a Maven script for a simple Java web client in the form of a unit test. It

- sources the web services contract (WSDL) from the service host,
- generates the Java adapter and binding classes (JAX-WS runs JAXB under the hood),
- compiles the test class, and
- executes the test.

18.1 Java-6 POM

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/maven-v4_0_0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>za.co.solms.example</groupId>
8   <artifactId>jaxws-maven-sample-java6</artifactId>
9   <packaging>jar</packaging>
10  <version>0.1</version>
11
12  <name>JAX-WS / Maven Sample (Java 6)</name>
13
14  <description>
15    A Sample Maven 2.x project that illustrates usage
16    of the JAX-WS Maven plugin to compile and use
17    web services from Java. In Java SE 6, JAX-WS is
18    part of the platform, so this project only requires
19    the JAX-WS WSDL compiler plugin to do initial
20    WSDL to Java compilation.
21  </description>
22
23  <developers>
24    <developer>
25      <organization>Solms TCD</organization>
26      <organizationUrl>
27        http://www.solms.co.za/
28      </organizationUrl>
29      <email>info@solms.co.za</email>
30    </developer>
31  </developers>
32
33  <build>
```

```

34 <plugins>
35 <plugin>
36 <groupId>org.codehaus.mojo</groupId>
37 <artifactId>jaxws-maven-plugin</artifactId>
38 <executions>
39 <execution>
40 <goals>
41 <goal>wsimport</goal>
42 </goals>
43 </execution>
44 </executions>
45 <configuration>
46 <target>2.1</target>
47 <wsdlUrls>
48 <wsdlUrl>http://www.webservices.net/CurrencyConvertor.asmx?WSDL</wsdlUrl>
49 </wsdlUrls>
50 <sourceDestDir>
51 target/generated-sources
52 </sourceDestDir>
53 </configuration>
54
55 <dependencies>
56 <dependency>
57 <groupId>com.sun.xml.ws</groupId>
58 <artifactId>jaxws-tools</artifactId>
59 <version>2.1.7</version>
60 </dependency>
61 </dependencies>
62 </plugin>
63 <plugin>
64 <groupId>org.apache.maven.plugins</groupId>
65 <artifactId>maven-compiler-plugin</artifactId>
66 <configuration>
67 <source>1.7</source>
68 <target>1.7</target>
69 </configuration>
70 </plugin>
71 </plugins>
72 </build>
73
74 <dependencies>
75 <dependency>
76 <groupId>junit</groupId>
77 <artifactId>junit</artifactId>
78 <version>[4.11,<
79 <scope>test</scope>
80 </dependency>
81 </dependencies>
82
83 <repositories>
84 <repository>
85 <id>maven2-repository.dev.java.net</id>
86 <name>Java.net Maven 2 Repository</name>
87 <url>http://download.java.net/maven/2</url>
88 </repository>
89 </repositories>
90 <pluginRepositories>
91 <pluginRepository>
92 <id>maven2-repository.dev.java.net</id>
93 <url>http://download.java.net/maven/2</url>
94 </pluginRepository>
95 </pluginRepositories>
96 </project>

```

18.2 The Test Application

```

1 package za.co.solms.example;
2
3 import net.webservices.Currency;
4 import net.webservices.CurrencyConvertor;

```

```

5 import net.webservicex.CurrencyConvertorSoap;
6 import org.junit.Test;
7
8 public class JAXWSTest
9 {
10     @Test
11     public void testCurrency()
12     {
13         Currency from = Currency.USD;
14         Currency to = Currency.ZAR;
15
16         System.out.printf
17             ("Getting conversion rate from %s to %s...\n", from, to);
18
19         CurrencyConvertorSoap cc
20             = new CurrencyConvertor().getCurrencyConvertorSoap();
21
22         double rate = cc.conversionRate(from, to);
23
24         System.out.printf
25             ("From web service: Conversion rate from %s to %s is %f\n", from, to, rate);
26     }
27 }

```

18.3 Executing the web service test

Running

```
1 mvn test
```

sources the WSDL, compiles the adapter and binding classes as well as the test and then runs the test:

```

1 [INFO] Scanning for projects...
2 [INFO] -----
3 [INFO] Building JAX-WS / Maven Sample (Java 6)
4 [INFO] task-segment: [test]
5 [INFO] -----
6 ...
7 -----
8 T E S T S
9 -----
10 Running za.co.solms.example.JAXWSTest
11
12 Getting conversion rate from USD to ZAR...
13
14 From web service:
15 Conversion rate from USD to ZAR is 7.431000
16
17 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
18 Time elapsed: 2.544 sec
19
20 Results :
21
22 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

```


Part III

Context and Dependency Injection (CDI)

Chapter 19

Dependency Injection

Dependency injection is a software design pattern which implements an inversion of control. Instead of the client requiring the resource determining which resource to use through either looking up or instantiating the resource, the resource reference is initialized by the client environment.

19.1 The problem

Assume an instance of an **OrderProcessor** requires some **InvoiceGenerator** to generate invoice for order

```
1 public interface InvoiceGenerator
2 {
3     public Invoice generateInvoice(Order order);
4 }
5
6 public class OrderProcessor
7 {
8     public OrderConfirmation processOrder(Order order)
9     {
10         ...
11         Invoice invoice = invoiceGenerator.generateInvoice(order);
12         ...
13     }
14 }
```

1. Tight coupling:

- Replace **InvoiceGenerator** implementation class.

2. Reusability of **OrderProcessor**:

- in environments where different class is used to generate invoices.

3. Breaking single responsibility principle:

- In addition to its core responsibility
- **OrderProcessor** creates/sources instances of its service providers.

4. Dependency inversion:

- low level class should *never* have dependency on high-level class
5. **Architecture/framework lock-in:**
 - Tightly couple application code to infrastructural classes.
 - e.g. when service provider is entity manager, adapter, ...
 6. **Dependent object initialization:**
 - May be complex and may change
 7. **Difficult to unit test:**
 - Unit testing environment: use mock object for `InvoiceGenerator`
 - Testing IoC can inject mock objects.

19.2 The solution provided by dependency injection

The solution provided by the dependency injection pattern is to introduce an *Inversion of Control* (IoC) container which manages and provides dependencies. In particular, the IoC scans for resources (including classes) one which a class can have dependencies, takes over life cycle management for these resources (e.g. instantiates and initializes resources) and initializes the references or pointers to the resources in client classes.

The elements of a dependency Injection include

- a) *Inversion of Control* (IoC) container which manages & provides dependencies
- b) an interface (contract) for the resource,
- c) implementation of a resource
- d) the client class depending on the service;
- e) an injector object responsible for injecting the resource into the client.

19.3 Implementations

Dependency injection implementations exist for many programming languages and frameworks.

- Widely implemented across the *Java* eco-system
 - CDI (Context and Dependency Injection)
 - * standard API spec
 - *SquareDagger*, *PicoContainer*
 - * minimalist/light-weight frameworks
 - *Spring DI*
 - * Very feature-rich DI framework implementation
 - * meant to be used within the *Spring* framework
 - *Googe Guice*

- * Framework independent feature rich DI framework
- *C++* DI frameworks:
 - *Walleroo* (released under Boost license)
- *JavaScript*
 - *AngularJS DI*
- *C#*
 - *Spring.Net*

Chapter 20

Introduction to CDI

Context and Dependency Injection (CDI) is a community-managed standard for dependency injection in *Java*. It provides

- a **CDI context** enabling one to bind lifecycle and interactions to extensible CDI life cycle contexts, and
- type-safe dependency injection of components managed by CDI context.

20.1 What does CDI provide?

CDI provides

1. A standard for context and dependency injection which supports
 - **Stand-alone & Java-EE embedded CDI containers**
 - **Type Safety:**
 - CDI uses interfaces to resolve injections
 - **Wide injection target:**
 - any Java object whose life-cycle can be managed by a CDI container.
 - e.g. enterprise beans, persistence contexts, Web service references, ...
 - **Decorators:**
 - Can decorate injected components.
 - **Events:**
 - Send and receive type-safe events with loose coupling.
 - **Interceptors:**
 - Can associate interceptors with components.
 - **Support for Unified Expression Language :**
 - facilitates injection into facelets, ...
 - **Service Provider Interface (SPI):**
 - enables 3rd party frameworks to integrate with CDI
2. A reference implementation.

20.2 Decoupling through CDI

CDI decouples clients and service providers in a couple of ways:

1. *Server implementation may vary* and CDI injects an instance of that class which currently implements the interface for the server, and satisfies certain characteristics.
2. *Decouples client and service life cycles* by making components contextual, with automatic life cycle management.
3. *Decoupling message producers and consumers* through an events mechanisms.
4. *Decoupling of orthogonal concerns* through interceptors.

Chapter 21

Understanding CDI

It helps to understand how CDI works under the hood.

21.1 What is a CDI bean?

A CDI bean is any injectable object. The following can be injected:

- Any concrete Java class which
 - Must have appropriate constructor
 - * Default constructor
 - * Constructor annotated with `@Inject`
 - is not a static inner class
- Examples
 - JSF managed beans,
 - Local & remote enterprise beans,
 - Persistence contexts, JSF managed beans,
 - JNDI resources (e.g. queues and topics, connection pools, ...)
 - Web service references, ...

21.2 Specifying the state retention period With bean scope

CDI Beans are singletons in some scope. The state of a CDI bean is maintained within its scope. The CDI bean scopes are

- `@RequestScoped`
 - State maintained single user interaction (e.g. single HTTP request)
- `@SessionScoped`
 - State maintained across interactions within user session.

- `@ApplicationScoped`
 - Shared instance (state) across all interaction with (web) application
- `@Dependent`
 - The injected bean shares the life cycle of the context it is injected into. This is the default scope.
 - This is the *default scope*.
- `@ConversationScoped`
 - Multiple cycles of a JSF request life cycle.
 - Allows for program determined conversation start and end.
- `@Singleton`
 - State shared among all clients.

21.3 Assigning Beans EL Names

Beans are readily injectable into and accessible from Java code. At times, need beans accessible also from EL expressions, e.g. from facelets. To this end we commonly annotate CDI beans as `@Named`. By default the class name is used, but we can assign different name via `@Named('BeanName')`.

21.4 Injecting a CDI bean

In order to inject one CDI bean into another, we annotate the field or setter with with `@Inject`. For example, below we inject a persistence context into a stateless session bean:

```

1 @Stateless
2 Class MyStatelessSessionBean
3 {
4
5     public list<Client> getOverdrawnClients()
6     {
7         ...
8         Query query = persistenceContext.creatNamedQuery("overdrawn");
9         return query.getResultList();
10    }
11
12    @Inject
13    private PersistenceContext persistenceContext;
14 }
```

21.5 Using qualifiers

CDI beans are singletons in some scope. Sometimes we require different implementations of a bean type. In such cases we need to define a qualifier. For example, we might want different types of message senders concurrently deployed (e.g. a `SmsMessageSender` and a `JabberMessageSender`).

We then require a qualifier annotation which enables users to specifically request a Jabber message sender:

```

1 import static java.lang.annotation.ElementType.FIELD;
2 import static java.lang.annotation.ElementType.METHOD;
3 import static java.lang.annotation.ElementType.PARAMETER;
4 import static java.lang.annotation.ElementType.TYPE;
5 import static java.lang.annotation.RetentionPolicy.RUNTIME;
6 import java.lang.annotation.Retention;
7 import java.lang.annotation.Target;
8 import javax.inject.Qualifier;
9
10 @Qualifier
11 @Retention(RUNTIME)
12 @Target({TYPE, METHOD, FIELD, PARAMETER})
13 public @interface Jabber {}

```

We can now qualify different bean implementation types using the qualification annotations. This enables us to

- to have different bean implementations concurrently deployed, and
- to specify which bean implementation should be injected.

```

1 package za.co.solms.training.cdi.MessageSender;
2
3 @Jabber
4 public class JabberSender implements MessageSender
5 {
6     public boolean sendMessage(String message, String recipient)
7     {
8         // code for sending message over Jabber
9     }
10 }

```

Now, if we want to inject a qualified bean, we annotate the bean field or setter with both, `@Inject` and the qualification annotation:

```

1 @Named
2 @RequestScoped
3 class QualifiedMessagingClient
4 {
5     public String processOrder()
6     {
7         // some fancy code
8         messageSender.sendMessage(recipient, message);
9
10        return "orderConfirmation";
11    }
12
13    @Inject @Jabber
14    private MessageSender messageSender;
15
16    private String messageText;
17    private String message;
18 }

```

Note that client still decoupled from implementation class.

21.6 A simple example

Below is a simple interface for a message sender. The `@Named` annotation is to make the CDI bean accessible from EL expressions and the `@Dependent` annotation is unnecessary as it would have been the default anyway.

```

1 package za.co.solms.training.cdi.MessageSender;
2

```

```

3 @Named // So that it can be accessed from EL expressions
4 @Dependent // could omit this as this is default
5 public interface MessageSender
6 {
7     public boolean sendMessage(String message, String recipient);
8 }

```

Below we have an implementation of a message sender which sends messages over email.

```

1 public class EmailSender implements MessageSender
2 {
3     public boolean sendMessage(String message, String recipient)
4     {
5         // code for sending message over Jabber
6     }
7 }

```

The messaging client has the CDI container inject a message sender.

```

1 @Named
2 @RequestScoped
3 class MessagingClient
4 {
5     public String processOrder()
6     {
7         // some fancy code
8         messageSender.sendMessage(recipient, message);
9
10        return "orderConfirmation";
11    }
12
13    @Inject
14    private MessageSender messageSender;
15
16    private String messageText;
17    private String message;
18 }

```

Should we replace the deployed message sender to a Jabber or SMS message sender, we would not have to make any changes to our messaging client. Furthermore, our client can be unit tested in a mocking environment.

21.7 How are bean references obtained?

CDI's `BeanProvider` implements Singleton per scope. It provides bean (resource) references via

```

1 T getReference(Bean<T> type, Qualifier... qualifiers);

```

The method is not called directly by the bean/resource client, but from annotations and Expression Language Processors. Qualifiers are required if multiple instances of same the type of CDI bean is required.

Part IV

The Java Persistence Api (JPA)

Chapter 22

Overview

The Java Persistence API (JPA) is a specification of a Java API for accessing, persisting, and managing data persisted in a relational database or in any other database for which there is a JPA provider.

JPA is used to persist POJOs (Plain Old Java Objects). These classes do not have to implement any particular interface or subclass any particular class.

22.1 What is JPA?

JPA, the *Java Persistence API* aims to provide a standard persistence framework to be used when persisting entities to a database. Initially JPA provided a standard API for Java-based object-relational mappers which enabled you to

- abstract from
 - any specific O/R framework (e.g. Hibernate, EclipseLink, ...), and
 - from any specific relational database and the flavour of SQL used by that database,
- use an object cache to improve scalability and performance.

In the mean time one can abstract fully from the database technology used using JPA/JDO bridges to persist to object databases and specific JPA adapters for different NOSQL databases.

Even though JPA is commonly used from within enterprise applications which often run in enterprise application servers, JPA can also be used within Java-SE applications.

22.2 Why use JPA?

1. Decoupling/abstraction
 - from JPA-provider, Technology-specific query language and Persistence Technology
 - no vendor lock-in
2. Reduce code bulk
 - plumbing code removed (code only business logic)

3. Reduce errors & improve consistency
 - mapping complex object graphs onto relational databases is error-prone
 - database schemas and structure created from object graphs
4. Improve performance
 - through object caching
5. Maintainability & Portability
 - Through code reduction, decoupling and JPA being a widely supported public standard.
6. JPA Criteria
 - Have powerful framework for dynamic query construction

22.3 What does JPA provide?

- O/R mapping
- Persisting, removing, querying, updating
- Object caching
 - with eager and lazy data retrieval
- Value objects and merging
- Object-Oriented query language with mapping onto technology-specific query language
 - including pre-compiled queries
- Concurrency support
- Constraint validation support
- Custom converters
- Dynamic query construction
- Calling stored procedures

22.4 JPA providers

There is a wide range of JPA implementations to choose from. Widely used examples include *EclipseLink*, *Hibernate*, *OpenJPA* and *DataNucleus*. These implementations compete on non-functional attributes (e.g. performance) and different implementation may be more suitable for different systems.

In addition, JPA implementations may provide non-standard extensions. It is advisable that these are either avoided or at least ring fenced. Otherwise the portability and flexibility of the application is compromised.

Chapter 23

Persistence context

23.1 Overview

Even though a persistence context is one of the central concepts of JPA, it is often not very well understood.

23.1.1 What is a persistence context?

A persistence context is a cache of objects whose persistence is managed. The cache maintains objects in memory which can be efficiently manipulated without every time having to consult the database.

The persistence context typically maintains a set of non-shared database connections. Only one instance with the same object identity exists within a persistence context.

23.1.2 What is an entity manager?

An entity manager is an entity resource manager which is associated with a persistence context. It maintains a cache for the persistence context and the life cycle of the entity instances contained within that persistence context. The entity manager interacts with the object-relational mapper and uses a connection pool to interact with the persistence provider (e.g. database).

The entity manager is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

Entities which are managed by an Entity Manager will automatically propagate these changes to the database when a transaction is committed. Entities which have been detached can be merged back into managed state resulting in any modifications made outside the persistence context being ultimately persisted to database upon commit.

23.1.3 Optimistic concurrency control

For highly scalable systems one usually requires optimistic concurrency control with versioning. Version or timestamp checking is used to detect

- conflicting updates across transactions, and to
- prevent lost updates within a transaction.

The behaviour is really the same as in concurrent version control systems like subversion or git or subversion. Different persistence contexts obtain their own detached copy of the entities. Which ever persistence context commits first will merge their changes into the global persistence context and persist its domain to the database.

Subsequent entity managers who commit may encounter a conflict when they merge their changes back into the global context. If this is the case, an exception is thrown. Otherwise the changes made within that transaction are persisted through to the database.

23.1.4 Life span of entity manager

JPA supports two types of persistence contexts. Transaction-scoped persistence contexts are specific to a single transaction whilst extended persistence contexts span across transactions.

23.1.4.1 Transaction-scoped persistence contexts

In the case of transaction-scoped persistence context, one will obtain (in a managed application) or have to create (in a non-managed application) a new persistence context per transaction. Any entities which have been enlisted within the cache will be detached at the end of the transaction and any changes made after detachment will no longer be propagated into the database.

Transaction-scoped persistence contexts do not support optimistic concurrency control. They are thus largely used in non-managed applications which usually do not have high concurrency demands.

23.1.4.2 Extended persistence contexts

Extended persistence contexts maintain a cache across transactions. They provide thus more efficient caching and support optimistic concurrency control. Extended transaction contexts are typically used in managed applications where the caching and optimistic concurrency control are important to achieve the required scalability.

23.1.5 Transaction management

23.1.5.1 The transaction management for a persistence context

The transaction type for a persistence context may be either `RESOURCE_LOCAL` or `JTA`. In the case of `RESOURCE_LOCAL` the transaction management is provided by JPA which typically delegates it to the local resource (e.g. database). In the case of `JTA` a transaction manager implementing the *Java Transaction API* is used. Such entity managers can enlist multiple resources within a transaction. In managed environments transaction boundaries are usually managed by the application server deducing the relevant transaction boundaries from the more abstract transaction requirements annotations (e.g. `requires`, `requires-new`, ...).

`RESOURCE_LOCAL` is often used in non-managed applications where transaction control may be required only for resources from a single database. When using `RESOURCE_LOCAL`, you must use the entity transaction API to begin commit around every call to your entity manager:

The entity manager for a managed environment is provided by the application server. It will usually use `JTA` transaction management, allowing for multiple resources (e.g. databases, message queues, external systems, ...) to be enlisted within a transaction.

23.1.6 The scope of a persistence context

The scope of a persistence context is the domain of entities which are managed by it, i.e. the collection of entitied managed by the persistence context.

The scope can be specified in one of the following ways:

1. A persistence unit may refer to a `orm.xml` file defining the entities and how they should be mapped onto a relational database. This is specified in a `<mapping-file>` element in the `persistence.xml`.
2. You can use one or more `<jar-file>` elements to specify that the entity classes in those jar files need to be included in the persistence context.
3. You can have a list of `<class>` elements listing the entity classes to be managed within the persistence context.
4. The annotated entities contained in the root of the persistence unit which is the `jar` file or directory, whose `META-INF` directory contains the `persistence.xml` file. This approach is the common approach when defining the persistence context for managed applications.

23.1.7 How are entity managers obtained?

Depending on whether one performs persistence from a managed or non-managed application, the entity manager is either provided/injected by the environment (i.e. by the application server) or needs to be created manually.

23.1.7.1 Obtaining an entity manager in a container managed environment

In a container managed environment the entity manager is provided by the container and is obtained either via dependency injection by annotating an `EntityManager` field or via a JNDI lookup.

To obtain a JTA based entity manager you need to annotate the entity manager field with a `@PersistenceContext` annotation

To obtain a entity manager using `RESOURCE_LOCAL` JTA provider, you annotate the entity manager field with a `@PersistenceUnit` annotation

23.1.7.2 Manual Creation of Entity Manager in JavaSE Applications

In a JavaSE application, the entity manager is not injected from a container, but must be created explicitly. For this you will

- define the persistence context descriptor, `persistence.xml` in a `META-INF` directory (or construct the persistence context properties in code),
- instantiate a entity manager factory for your persistence context from the general persistence environment, providing the entity manager factory a suitable name, and
- obtain an entity manager for your persistence context from entity manager factory.

Generally you should only have a single entity manager per persistence context active at any time. **Note:** *Calling `entityManagerFactory.createEntityManager()` twice results in two separate `EntityManager` instances and therefor two separate `PersistenceContexts/Caches`.*

```

1 EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnitName);
2 EntityManager em = emf.createEntityManager();

```

23.1.8 Detaching objects from a persistence context

If an object leaves the cache/persistence context, it is detached from it. The entity is then in a value object state. This will, for example, happen when an object is serialized (e.g. by being passed as parameter in a remote service request). An object will also become detached if it exists beyond the life span of the entity manager (and hence cache). In addition the entity can be manually detached via

```

1 entityManager.detach(myEntity);

```

Any updates made to a detached object are not reflected in the object cache and are not propagated to the database upon transaction commit.

You cannot call request an entity manager to persist or remove a detached entity (value object). Once the value object has been re-attached to the persistence context via

```

1 entityManager.merge(myEntity);

```

it can be persisted and removed again.

Note: Due to lazy loading, detached objects (e.g. serialized parameters) may not have all the information populated.

23.2 Persistence context configuration

The way in which a persistence context is configured depends on whether it is a managed or a non-managed application.

23.2.1 Configuring the persistence context for a non-managed application

For non-managed Java applications one needs to specify the database, database driver and login credentials which should be used as well as the set of entity classes which should be managed by the persistence context. The latter can be specified as a list of classes, in a separate `orm.xml` file or by specifying the jar-file(s) which contains the entity classes. The latter is often the most convenient approach:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
3
4 <persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
5   <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
6   <jar-file>myEntities.jar</jar-file>
7   <properties>
8     <property name="eclipselink.target-database" value="DERBY"/>
9     <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
10    <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
11    <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/myDB;create=true"/>
12    <property name="javax.persistence.jdbc.user" value="myApp"/>
13    <property name="javax.persistence.jdbc.password" value="myApp"/>
14  </properties>
15 </persistence-unit>
16
17 </persistence>

```


23.2.2 Configuration of persistence context for managed applications

The persistence unit configuration for a managed environment is specified in the `persistence.xml` file contained in the META-INF directory. It typically uses JTA-based transactions and refers to a data source defined for the container. In addition it can specify some properties for the object-relational mapper:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
4 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
6 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
7   <persistence-unit name="myEnterpriseApp" transaction-type="JTA">
8     <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
9     <jta-data-source>jdbc/myDataSource</jta-data-source>
10    <properties>
11      <property name="eclipselink.ddl-generation" value="create-tables"/>
12    </properties>
13  </persistence-unit>
14 </persistence>
```


Chapter 24

Entities

Entities are data classes which exist from when they are created up to the point where they are explicitly removed. They are persisted through to persistent storage (e.g. database) and may survive the life span of a session or application.

24.1 Simple entities

24.1.1 Declaring entities

An Entity is defined by annotating them with `javax.persistence.Entity`

```
1 import javax.persistence.Entity;
2
3 @Entity
4 public class Account
5 {
6     ...
7 }
```

One may customize the persistence by specifying, for example, the database table to which the entity should be persisted via

```
1 @Entity(name="ACCOUNTS")
2 public class Account
3 {
4     ...
5 }
```

24.1.2 Requirements for entities

Entities must satisfy a number of requirements:

- **Constructors:** Entities must have a `public` or `protected` default (no-argument) constructor. They may have other constructors. If the default constructor is declared `protected`, it is only available for the entity manager and users are forced to use the publicly available constructors.
- **Primary key:** Every entity requires a primary key, which may be a simple primary key represented by a bean field, or a composite key. The primary key is specified by annotating the relevant field with `javax.persistence.Id`

- **Support for serialization:** Entities which are meant to be detachable in order to pass them around as value objects (i.e. sent to a client through a remote interface) must be serializable. These temporarily detached value objects can be re-attached to the entity manager at a later stage.
- **Final:** Neither the class, nor any of its methods, may be final. The JPA provider must be able to subclass your class, in order to provide natural interception points and to access protected fields not published via public access methods.
- **Entities and inheritance:** Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- **Entities may be abstract:** Shared data may be encapsulated in abstract entities which are subclassed by various concrete entities.
- **Fields accessed only via accessors or business methods:** Persistent instance fields must be declared with private, package or protected scope (preferably private), and can only be accessed directly by the entity class's methods. Clients must access the entity's state through accessor or business methods.
- **All annotations either at getter or field level:** The entity manager will access the fields either directly or via getters and setters. The method used depends on whether the annotations are don on the fields or on the getters. They should, however, not be mixed - i.e. they should be either all at the field level or all at the getter level. Alternatively the access type can be explicitly specified on an entity by annotating the entity with either `@Access(AccessType.FIELD)` or `@Access(AccessType.Property)`.

24.1.3 What is persisted?

The persistent state of an entity is defined by its fields. The fields are accessed by the entity manager either

- via *accessors* (`getXXX()` methods) following the JavaBeans specification, or
- via *direct access to fields*.

The schema used for a particular object is inferred, based on whether the *primary key* (the `@javax.persistence.Id` annotation) has been indicated on a field, or an accessor method.

For example, state management would be performed through the get/set service of the following class. The entity manager will call the services in order to extract the state to be stored in the database, or to populate an instance with information from the database.

24.1.3.1 Valid persistent field types

The following are valid data types for persistent fields:

- Java primitives and primitives wrappers,
- The following built-in classes:
 - `java.lang.String`,
 - `java.util.Date` (requiring the `@Temporal` annotation),
 - `java.math.BigDecimal` and `BigInteger`,

- `java.sql.Time` and `Timestamp` (requiring the `@Temporal` annotation),
- `byte[]`, `Byte[]`, `char[]` and `Character[]`,
- `java.sql.Blob` and `Clob`,
- embedded classes,
- other entities,
- collections of primitives, and
- any other serializable objects.

24.1.3.2 Collection variables

Collection variables are automatically persisted. The supported `java.util.Collection` types are `java.util.List`, `java.util.Set` and `java.util.Map`.

Generics should be used (e.g. `List<Account>`), and the various relationship annotations (such as `@OneToMany`) are required to control the mapping, such as putting bi-directional mappings in place.

Collections are mapped onto separate tables. In the case of many to many relationships a join table is created.

24.1.3.3 Transient fields

Transient fields (fields which do should not be persisted, and hence do not form part of the object's persistent state) are specified by

- in the case of property access annotating the getter or setter as `@javax.persistence.Transient`, or
- by declaring the field itself as `transient` using the Java language keyword, in the case of field access.

24.1.3.4 Field validation

Field validation may be done in the *setter* methods, which may throw an exception. An exception will cause the controlling transaction (if any) to be rolled back.

Note:

- *Consider using bean validation*
- *It is typically questionable whether use-case specific validation should be performed on the entity object at all: This should rather be enforced at services level (e.g. the session beans).*

24.1.4 Embeddable classes

Entities may have as components finer grained objects which are persisted, not as separate entities, but are expanded as a set of columns within within the tables created for the entities within which they are embedded.

For example, a location may have a name, an address and geographic coordinates which include the degrees longitude and degrees latitude. Embedding the `GeographicLocation` class within a `Location` entity would add the `degreesLongitude` and `degreesLatitude` columns to the `Location` table.

As such embedded objects have no persistent identity. Their identity is the role in the context of the owner entity.

Note: *Embedded classes are only used for composition relationships between classes, i.e. no other object may obtain a reference to an embedded object.*

24.1.4.1 Defining embeddable classes

An class which is meant to be embeddable within entity beans must be annotated as such using the `@Embeddable` annotation.

```

1 @Embeddable
2 class GeographicLocation implements Serializable
3 {
4     // getters & setters
5
6     private double degreesLongitude, degreesLatitude;
7 }

```

24.1.4.2 Specifying the access type

By default, the access type of an embeddable class is determined by the access type of the entity within which it is embedded. This can be changed by annotating the embeddable class with an `@Access` annotation whose value is either `AccessType.Field` or `AccessType.Property`.

Note: *It is generally recommended to specify the access type of the embeddable explicitly in order to prevent potential object-relational mapping errors caused by the entity manager loosing track of the state due to access through both channels. This can happen when the embedded class is contained in an entity with one access type which is, in turn, part of an entity which uses another access type.*

24.1.4.3 Embedding a class within an entity

To embed an embeddable class within an entity one has to add a field for the embedded class and annotate it or the getter as `@Embedded`

```

1 @Entity
2 class Location implements Serializable
3 {
4     ...
5
6     @Embedded
7     public GeographicCoordinates getCoordinates() {return coordinates;}
8
9     private String name;
10    ...
11    private GeographicCoordinates coordinates;
12 }

```

24.1.5 Primary keys

For every entity one must specify a primary key which may be

- a simple primary key, or
- a composite primary key.

24.1.5.1 Simple primary keys

It is generally preferable to have a simple primary key which is independent of any business semantics.

Simple primary keys are persisted into a single database column which will be assigned a primary key constraint.

24.1.5.1.1 Valid data types for simple primary keys The following are valid data types for persistent fields:

- Java primitives and primitives wrappers, and
- `java.lang.String`

Although approximate numeric types like `float` or `double` are permitted, they should generally not be used due to their inability to represent absolute values.

24.1.5.1.2 Specifying the primary key field A primary field is specified for an entity by annotating either

- an accessor method, or
- an instance field

with `@javax.persistence.Id`.

For example, the following code snippet specifies that the `accountNo` is to be used as a primary key for accounts:

```
1 @Entity
2 public class Account
3 {
4     public getAccountNo() {return accountNo;}
5     ...
6     @Id
7     private int accountNo;
8 }
```

24.1.5.1.3 Automatic key generation One will commonly request the entity manager/-database to automatically generate the value of a primary key (which will always be a unique value) by annotating the key with the `@GeneratedValue` annotation:

```
1 import java.io.Serializable;
2 import javax.persistence.*;
3
4 @Entity
5 public class Account implements Serializable
6 {
7     ...
8
9     @Id
10    @GeneratedValue(strategy=GenerationType.AUTO)
11    private long accountNumber;
12 }
13 }
```

The annotation takes parameters, which allows the developer to indicate the generator (such as a particular database table), and/or to indicate the strategy to be used (typically realised by the underlying database). When one “doesn’t care, as long as it is unique” the `AUTO` strategy usually works well.

24.1.5.2 Composite primary keys

The JPA specification supports composite keys via primary key classes. A primary key class is defined as an embeddable class whose properties form the primary key fields. It must have a default constructor as well as setters and getters for the primary key fields.

24.1.5.2.1 Interface for the primary key class

```

1 package za.co.solms.partsCatalog;
2
3 /**
4  * Interface for a part identifier.
5  */
6 public interface PartId
7 {
8     public String getCode();
9
10    public String getManufacturerId();
11
12    public interface Mutable extends PartId
13    {
14        public void setCode(String newCode);
15
16        public void setManufacturerId(String newManufacturer);
17    }
18 }

```

24.1.5.2.2 Implementation of a primary key class

```

1 package za.co.solms.partsCatalog;
2
3 import java.io.Serializable;
4 import javax.persistence.Embeddable;
5
6 @Embeddable
7 public class PartPK implements PartId.Mutable, Serializable
8 {
9     public PartPK(String code, String manufacturerId) {
10         setCode(code); setManufacturerId(manufacturerId);}
11
12     protected PartPK(){}
13
14     public void setCode(String newCode) {this.code = newCode;}
15
16     public void setManufacturerId(String newManufacturer) {
17         this.manufacturerId = newManufacturer;}
18
19     public String getCode() {return code;}
20
21     public String getManufacturerId() {return manufacturerId;}
22
23     private String code, manufacturerId;
24 }

```

```

1 package za.co.solms.partsCatalog;
2
3 import javax.persistence.EmbeddedId;
4 import javax.persistence.Entity;
5

```



```

6 @Entity public class PartBean implements Part.Mutable
7 {
8     public PartPK getPartId() {return partPk;}
9
10    public String getDescription() {return description;}
11
12    public String getName() {return name;}
13
14    public void setDescription(String newDescription) {
15        this.description = newDescription;}
16
17    public void setName(String newName) {this.name = newName;}
18
19    public void setPartId(PartId newPartId) {
20        this.partPk = new PartPK(newPartId.getCode(), newPartId.getManufacturerId());}
21
22    @EmbeddedId
23    private PartPK partPk;
24    private String name, description;
25 }

```

24.1.6 Specifying column mappings

The object-relational mapping can be customized in the `orm.xml` entity descriptor file or via in-code annotations. For example, the column name, length and precision can be specified via the `@Column` annotation:

```

1 @Entity(name="VHCL")
2 public class Vehicle
3 {
4     @Column(name="REG_NO" length="10")
5     public String getRegistrationNumber()
6     {
7         ...
8     }

```

24.1.7 Column constraints

Commonly column constraints include a specification on whether a column is required or not and whether the entries in the column need to be unique:

```

1 @Entity(name="VHCL")
2 public class Vehicle
3 {
4     @Column(name="REG_NO", length="10", nullable=false, unique=true)
5     public String getRegistrationNumber()
6     {
7         ...
8     }

```

24.1.8 Primitive collections and maps

Can annotate collections and maps of basic types as `@ElementCollection`. If the storage provider is a relational database, collections of primitives are mapped onto a separate table with a link column and value columns. Maps of primitives onto primitives are mapped onto a separate table with one link column, one key column and one value column.

The mapping can be customized using the `@CollectionTable` annotation which allows you to specify the `name` of the table. Maps can be additionally annotated with a `@MapKeyColumn(name="...")` annotation.

```

1 @Entity
2 public class CarPriceList implements Serializable
3 {
4
5     @ElementCollection(fetch = FetchType.LAZY)
6     @CollectionTable(name = "UnavailableCars")
7     public List<String> withdrawnCars;
8
9     @ElementCollection(fetch = FetchType.EAGER)
10    public Map<String, double> activeCarPrices;
11 }

```

24.2 Relationships

JPA supports persistent relationships which are mapped down to database layer. There is some limited support for the standard object-oriented relationships:

- association and aggregations,
- composition, and
- specialization.

24.2.1 Summary of UML relationships

Figure 24.1 summarizes the UML relationships. It shows that these are conceptually specializations of each other and that we have weak and strong variants of “*is a*”, “*has a*” and “*uses*”.

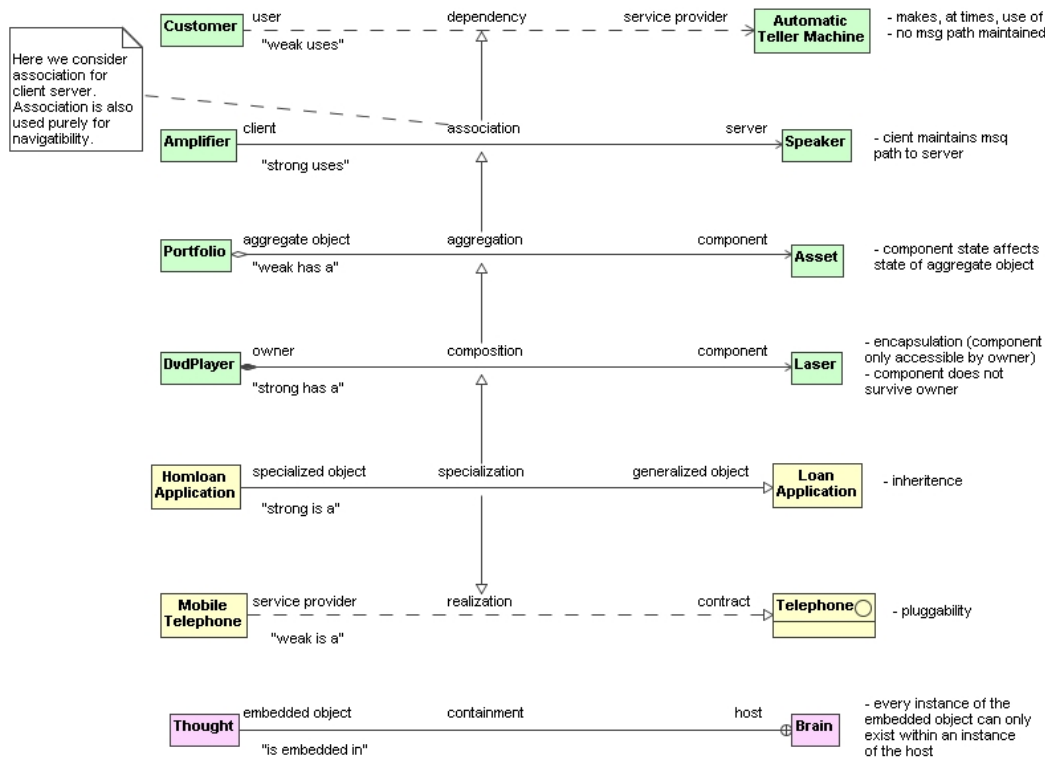
24.2.1.1 Dependency

Instances of the one class, the user, make, at times, use of instances of the other class, the service provider. The latter is often modelled as an interface in order to decouple the user from any particular implementation of a service provider. For example, clients of the bank, upon spotting an ATM, may decide to use it in order to withdraw some cash from their account, but they do not maintain a message path to any particular ATM. A dependency is called a “*weak uses*” because the user does not maintain a message path and is not in a position to, at any stage, send further service requests to the service provider.

24.2.1.2 Association

Association is used for two purposes. On the one side it is used purely for navigability. In the second case it is used for a client server relationship (or peer-to-peer in the case of binary associations). In either case, the object which has the association maintains a message path to the associated object. It is conceptually a special form of dependency where the client still, at

Figure 24.1: Summary of UML relationships



times, makes use of the service provider, but now the client maintains a message path to the service provider. For example, an amplifier has a message path to the speakers (the cables) in order to send service requests to them.

An association is called a “*strong uses*” because the client maintains the relationship and is in a position to send, at any stage, further service requests to the service provider.

24.2.1.3 Aggregation

Aggregation is a special form of association. The aggregate object still maintains a message path to the component. It still can make use of the components. For example, in the context of a portfolio calculating its value, it will request the value of each asset and sum them up. However, in aggregation a state transition in the component may imply a state transition in the aggregate object, i.e. aspects of the component state are part of the state of the aggregate object.

In our example, a change in the value of any of the assets results in a change in the value of the portfolio. Aggregation is a weak has a relationship because it does not take exclusive control of the component. The component can be accessed directly and may be part of other aggregate objects. Furthermore, the asset can survive the portfolio. For example, a particular asset may be part of a number of different portfolios. A change in its value results in the value of multiple portfolios changing. Furthermore, one may decide to remove a portfolio (a particular grouping view onto one’s assets), but the assets would still survive.

24.2.1.4 Composition

Composition is a special type of aggregation (and hence also a special type of association and a special type of a dependency). If the component state changes, the state of the owner also changes. The owner also maintains the message path and may, at any stage, issue further service requests to the component. Now we have, however, a “*strong has a*” relationship where the owner takes full responsibility for the component and encapsulates the component.

If a user of the DVD player wants to send a service request to its laser, it will have to do so via the services offered by the DVD player itself. If the laser is broken, the DVD player is broken too (it is responsible for the laser). Finally, should we decide to scrap the DVD player, the laser will be scrapped also.

24.2.1.5 Realization

Realisation is a weak is a relationship. It is used to show that a service provider implements an interface (and often a complete contract). This facilitates substitutability of one service provider with any other realising the same contract.

24.2.1.6 Specialization

Specialisation is a very strong relationship which should be used with care. It is commonly used for data or value objects. Specialisation can be conceptually seen as special form of realisation in that the sub-class is a specialised realisation of the super-class. One can say, specialisation inherits substitutability from realisation. It can also be seen as a special form of composition as every sub-class instance will create an encapsulated super-class instance through which it obtains the superclass attributes, services and relationships. The super-class instance for the sub-class cannot be accessed directly from outside the sub-class instance. It will also not survive the sub-class instance. The superclass instance is part of the state of the sub-class instance. If the state of the superclass instance changes, the state of the sub-class instance changes too.

For example, assume a home loan application inherits a loan amount from loan application. If the loan amount changes the state of the home loan application changes. The sub-class instance also maintains a message path to the super class instance (*super* in Java and *base* in C#). It is thus also a special for of association. It may, for example make use of a superclass service via *super.serviceRequest()* .

24.2.1.7 Containment

Containment is a separate relationship where instances of one class can only exist in instances of another There are examples of such relationships in nature.

24.2.1.8 Shopping for relationships

In order to determine the correct relationship between two classes one can take a requirements driven approach - similar to a shopping list for relationships. In either case one should *always choose the weakest relationship* which fulfils one’s requirements. The process of determining the

correct relationship goes along two legs. On the one side you are trying to establish the type of dependency between the two classes. On the other side you will assess the level of substitutability and inheritance required.

First we assess whether there is a dependency between the classes. If instances of one class, A, never make use of instances of another class, B, and if one also does not need to be able to navigate from an A to a B, then there is not much of a relationship between these classes. Otherwise there is at least a dependency of A on B.

Next ask yourself whether instances of A should maintain a message path to instances of B. If so, upgrade the dependency to an association. If not, leave the relationship as a dependency. If we reached this point, we have at least an association from A to B. Next you can ask yourself whether any change in the state of an instance of B results in a change of state in the instance of A which maintains an association to it. If the answer is yes, then upgrade the association to an aggregation relationship. Otherwise leave it as an association.

If we reached this point we have at least an aggregation relationship from A to B. Next, you can ask yourself whether the aggregate object needs to take full control of the component, or whether other objects should be allowed to access the component directly. If full control is required, then upgrade the relationship to a composition relationship. Otherwise leave it as an aggregation relationship.

Note: *If you decided on composition, you can do the following test to check whether you perhaps made an error. Check whether it would make sense for the component to outlast (survive) the owner. If the answer is yes, then the relationship could not have been a composition relationship.*

Next let us look at the plug-ability requirements. If the class should be pluggable (i.e. if the service provider should be substitutable), then one should introduce a contract for the service requirements. In the bare form, the contract is simply an interface and we have a realisation relationship. In order to assess whether you should upgrade the realisation relationship to a specialisation relationship, assess whether you want to inherit common properties and services.

Note: *In general we would recommend to favour interfaces and realisation above inheritance and specialisation. The latter tends to result in very rigid designs which are difficult to modify. One may choose to use specialisation only for value or data objects which do not perform significant functionality.*

24.2.2 Composition relationships between entities

In a composition relationship the component may not survive the owner. This is supported in JPA via the *cascading* relationship attribute. Cascading is supported for *create*, *merge* and *remove* operations. **Note:** *Cascading-delete enforces that the component entity bean is removed when the owner is removed.*

24.2.3 Relationship types

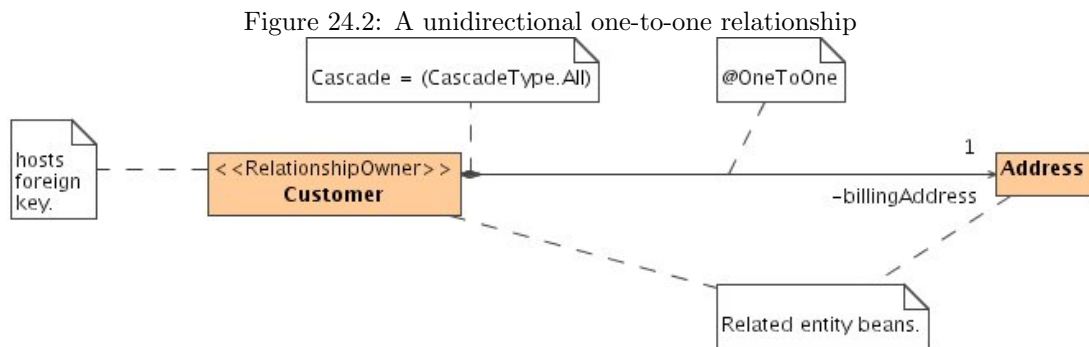
JPA supports uni-directional and bi-directional one-to-one, one-to-many, many-to-one and many-to-many relationships.

24.2.3.1 Relationships owner

For each relationship there is a *relationship owner* who maintains the pointer (e.g. foreign key) of the relationship. In the case of bi-directional relationships the related entity also provides a message path to the relationship owner.

24.2.3.2 Uni-directional single-valued relationships

Consider the uni-directional single-valued relationship shown in the following figure:

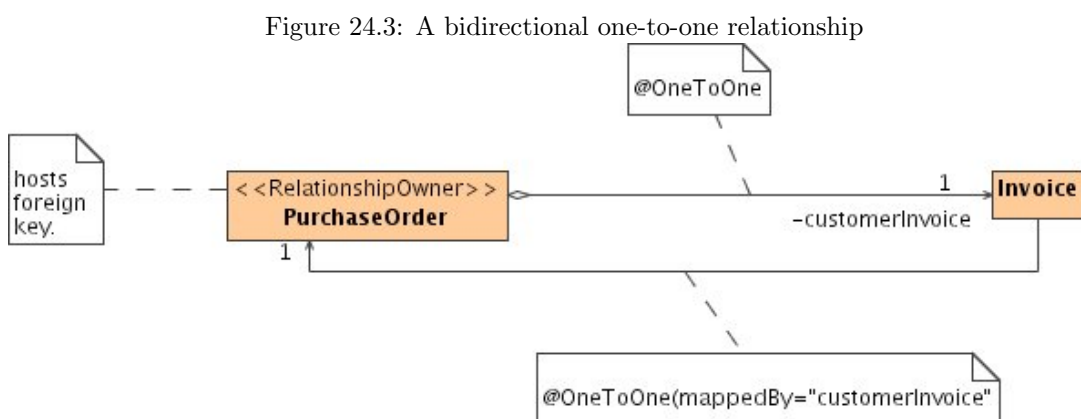


The relationship owner simply maintains the message path as well as the foreign key. The mapping onto entity beans would be as follows:

Note: *Cascading is specified to request the composition behaviour, i.e. that the component should not outlast the owner.*

24.2.3.3 Bi-directional one-to-one relationships

Consider the bi-directional one-to-one relationship shown in the following figure:



Here both entities maintain message paths to one another. At database level, there is, however, only one foreign key maintained, i.e. only one relationship owner.

```

1 @Entity public class PurchaseOrder
2 {
3     public Invoice getIssuedInvoice() {return invoice;}
4     public void setIssuedInvoice(Invoice inv) {invoice = inv;}
5     ...
6     @OneToOne
7     private Invoice issuedInvoice;
8 }
9
10 @Entity public class Invoice
11 {
12     public PurchaseOrder getPurchaseOrder() {return purchaseOrder;}
13     public void setPurchaseOrder(PurchaseOrder order) {purchaseOrder = order;}
14     ...
15     @ManyToOne(mappedBy="issuedInvoice")
16     private PurchaseOrder order;
17 }

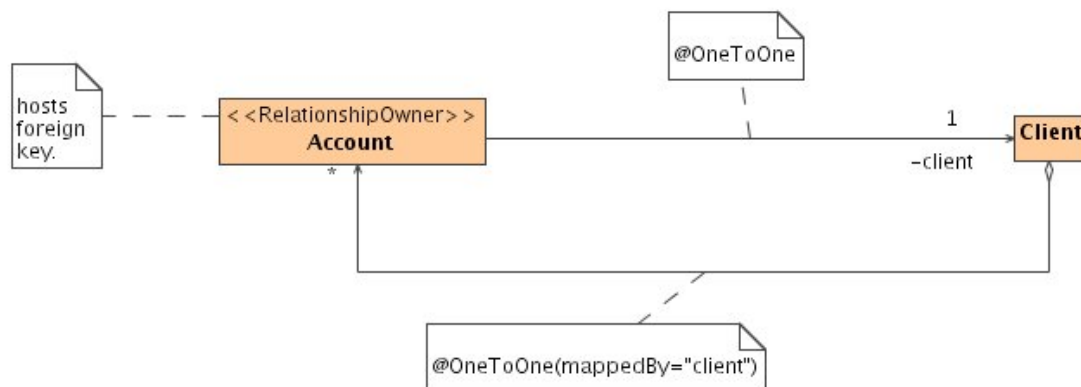
```

For the reverse relationship we specify a `mappedBy` attribute which ensures that this relationship is not implemented at storage level:

24.2.3.4 Bi-directional many-to-one relationships

To implement one-to-many or many-to-many relationships we need to use either one of the `java.util` collection types:

Figure 24.4: A bidirectional many-to-one relationship



The mapping onto entity beans would be as follows:

```

1 @Entity
2 public class Client
3 {
4     public Collection<Account> getAccounts() {return accounts;}
5     ...
6     @OneToMany(mappedBy="client")
7     private Collection<Account> accounts;
8 }
9
10 @Entity
11 public class Account

```

```

12 {
13     public Client getClient() {return client;}
14     public void setClient(Client clnt) {client = clnt;}
15
16     @ManyToOne
17     private Client client;
18 }

```

24.2.3.5 Cascading operations

Depending on your design, you may desire that operations on a parent component (e.g. a `Client`) *cascade* to its constituent components (for example, deleting a client may or may not cause all the client's accounts to be deleted). This is specified with the `cascade` parameter of any of the relationship annotations, with a set of enumerated values provided by the enumeration `javax.persistence.CascadeType`. For example, to cause all operation (including deletion) to be cascaded to the constituent component:

```

1 public class Client
2 {
3     ...
4     @OneToOne(cascade=CascadeType.ALL)
5     private Portfolio portfolio;
6 }

```

The allowable values are:

- **ALL:** Cascade all operations
- **MERGE:** Cascade merge (update) operation
- **PERSIST:** Cascade persist operation
- **REFRESH:** Cascade refresh operation
- **REMOVE:** Cascade remove operation
- **DETACH:** Cascade detach operations

From the perspective of mapping object-oriented relationships onto persistent storage, one should use not use any cascading for association and for composition one should use `CascadeType.ALL`.

24.2.4 Fetching strategies

Fetching strategies are used to optimize performance and scalability based on the expected usage of entities. They determine how much of an object graph is loaded into the cache when an entity is retrieved from persistent storage. The options are *EAGER* and *LAZY* fetching which respectively fetch or do not fetch the associated entity which has been annotated with the corresponding fetching strategy.

For example, below we request eager fetching of the client entity when retrieving an order entity, i.e. when the order is retrieved from the database, the associated client entity is also loaded into the cache:


```

1 @Entity
2 public class Order
3 {
4     ...
5     @ManyToOne(fetchType=FetchType.EAGER)
6     public Client getClient()
7     {
8         ...
9     }
10 }

```

The default fetching strategies in JPA are *EAGER* for one-to-one and many-to-one relationships and *LAZY* for one-to-many and many-to-many relationships.

24.2.5 Specialization

The concept of specialization and substitutability is not directly supported in relational database management systems. In order to support OO \leftrightarrow Relational mapping, we need to map specialization relationships onto relational databases.

JPA supports mapping of specialization relationships as well as polymorphism through to persistent storage level. To this end JPA supports a range of mapping strategies. None is perfect and JPA provides the option of requesting a mapping strategy which makes the appropriate quality attribute trade-offs for the problem at hand. In particular, one commonly trade-offs performance and scalability for maintainability and improved semantics.

24.2.5.1 Mapping onto relational databases

JPA supports 4 types of mappings of specialization relationships onto relational databases:

1. Joined subclass
2. Single table per class hierarchy
3. Table per class
4. Mapped superclass

24.2.5.2 Joined subclass

This is usually the preferred mapping. Each class in the specialization hierarchy is persisted in its own table. Subclass tables have a primary key column which acts as foreign key to the primary key column of the superclass, i.e. the object identity is preserved across all abstractions of an object.

The annotation specifying the mapping strategy is inherited, i.e. it need only be specified for the ultimate base class of the specialization hierarchy. For example

```

1 @Entity
2 @Inheritance(strategy=InheritanceType.JOINED)
3 public class Person {...}
4
5 @Entity
6 public class Employee extends Person {...}
7

```

```

8 @Entity
9 @InheritanceJoinColumn(name="EMPLOYEE_REF")
10 public class Contractor extends Employee {...}

```

24.2.5.3 Single table per class hierarchy

When choosing the *Single Table Per Class Hierarchy* strategy, the entire class is persisted in a single, typically sparsely populated table. The table has columns for the primary key, fields of all properties of all classes in the class hierarchy, and a discriminator column identifying the concrete class for that instance.

This strategy requires only a single lookup (no table joins), but a change to any classes in the hierarchy or the addition of another sub-class requires changing the table structure.

```

1 @Entity
2 @Inheritance(strategy=InheritanceType.SINGLE_TABLE
3 discriminatorType=DiscriminatorType.STRING
4 discriminatorValue="Person" /*default: fully qualified class name*/)
5
6 @DiscriminatorColumn(name="Type" // default: "TYPE")
7 public class Person {...}
8
9 @Entity @Inheritance(discriminatorValue="Employee")
10 public class Employee extends Person {...}
11
12 @Entity @Inheritance(discriminatorValue="Contractor")
13 public class Contractor extends Employee {...}

```

24.2.5.4 Single table per concrete class

In this mapping strategy each concrete subclass is separated by its own stand-alone table which contains all fields (also the inherited fields) of the class as well as a primary key column. The mapping results in a non-normalized database structure. Modifying a superclass will require modifying all tables for all concrete subclasses. On the other hand, an entity lookup is a single table lookup.

24.2.5.5 Mapped superclass

At times you don't want a separate table for an abstract superclass. Neither do we want to use a single table for class hierarchy. Instead we would like to embed superclass fields in table of concrete subclass. In such cases one would consider using a **MappedSuperclass**. The abstract base class would not be annotated as an **Entity** as no table is created for it.

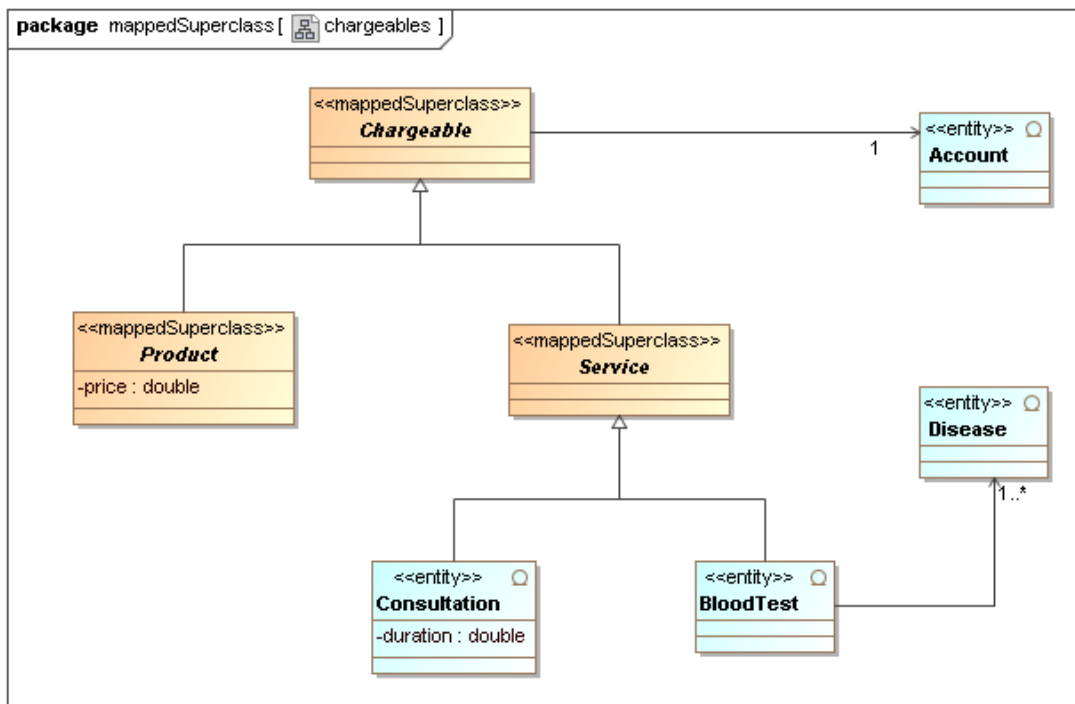
A **MappedSuperclass** is typically used if an abstract superclass has only very few fields one does not want the overheads of query on another small table. The resultant persistence mapping is not normalized. One effectively trades *maintainability* off for *performance* and *scalability* whilst retaining pluggability and polymorphism.

A **MappedSuperclass** is effectively an **Embeddable** class. The latter is used for *composition* relationships whilst the former is used for *specialization* relationships.

To request that the abstract base entity should be embedded annotated as **@MappedSuperclass**. In addition, one can still specify an inheritance strategy for class hierarchy.

24.2.5.5.1 Example Consider, for example, we could introduce the concept of a **Chargeable** as a *mapped superclass*. It holds only the object identifier and a reference to the income account. The **Product** class adds only a price and the **Service** class only introduces the concept of a service without adding anything. All three are abstract super classes which could potentially be implemented as `@MappedSuperclass`.

Figure 24.5: Using mapped super classes for Chargeables, Products and Services



```

1 import javax.persistence.GeneratedValue;
2 import javax.persistence.Id;
3 import javax.persistence.Inheritance;
4 import javax.persistence.InheritanceType;
5 import javax.persistence.MappedSuperclass;
6 import javax.persistence.OneToOne;
7
8 @MappedSuperclass @Inheritance(strategy=InheritanceType.JOINED)
9 public abstract class Chargeable
10 {
11     public Chargeable() {}
12
13     public Account getIncomeAccount() {return incomeAccount;}
14
15     public void setIncomeAccount(Account incomeAccount) {
16         this.incomeAccount = incomeAccount;}
17
18     public long getCode() {return code;}
19     public void setCode(long code) {this.code = code;}
20
21     @OneToOne
22     private Account incomeAccount;
23

```

```
24 @Id @GeneratedValue
25 private long code;
26 }
```

```
1 import javax.persistence.MappedSuperclass;
2
3 @MappedSuperclass
4 public class Service extends Chargeable
5 {
6     public Service() {}
7 }
```

```
1 import javax.persistence.MappedSuperclass;
2
3 @MappedSuperclass
4 public class Product extends Chargeable
5 {
6     public Product() {}
7
8     public double getPrice() {return price;}
9
10    public void setPrice(double price) {this.price = price;}
11
12    double price;
13 }
```

```
1 import javax.persistence.Entity;
2
3 @Entity
4 public class Consultation extends Service
5 {
6     public Consultation() {}
7
8     public double getDuration() {return duration;}
9
10    public void setDuration(double duration)
11    {
12        this.duration = duration;
13    }
14
15    private double duration;
16 }
```

```
1 import java.util.Set;
2
3 import javax.persistence.Entity;
4 import javax.persistence.OneToMany;
5
6 @Entity
7 public class BloodTest extends Service
8 {
9     public BloodTest() {}
10 }
```

```
11 public Set<Disease> getDiseaseCheckList()
12 {
13     return diseaseCheckList;
14 }
15
16 public void setDiseaseCheckList(Set<Disease> diseaseCheckList)
17 {
18     this.diseaseCheckList = diseaseCheckList;
19 }
20
21 @OneToMany
22 private Set<Disease> diseaseCheckList;
23 }
```


Chapter 25

The Java Persistence Query Language (JPQL)

The Java Persistence Query Language (JPQL) is a storage technology-neutral object-oriented query language enabling users to formulate queries across object graphs. As such the conceptual queries specified in JPQL are mapped onto the query language for the chosen persistence technology like the SQL for the relational database you've chosen or OQL for an object database.

25.1 JPQL versus SQL

The structure of an JPA query is in many ways similar to a traditional SQL query. It is generally of the form in which

- **SELECT:** specifies the type of objects or values to be selected which may be
 - an entity,
 - a value object or
 - a primitive data type
- **FROM:** specifies the domain to which the query applies and
- **WHERE:** specifies constraints which restrict the result collection.

For example, if we have an `Account` entity with a `balance` field, we can issue the following query:

```
1 SELECT a FROM Account a WHERE a.balance > 0
```

25.1.1 Result collections in JPQL

A core difference between JPQL and SQL is that the result collection in JPQL will be a collection of references to one of

- entities,
- other Java objects which are expanded within the same table (embedded classes),

- Java primitives,
- new instances of Java result objects whose fields are populated from the query,

while in SQL the result is a new conceptual table with column entries sourced potentially from different tables, i.e. it can contain elements from different tables and hence elements extracted from different entities.

Ultimately the result collection will be either a standard `java.util.Collection` or `java.util.Set`.

25.1.2 Selecting entity attributes

We can use the element access operator to select specific attributes of an entity. For example

```
1 SELECT a.balance FROM Account a
```

returns a collection of all account balances. The result collection will be of the data type of the `balance` field in the `Account` class (e.g. an instance of a `Money` class or a `Double`).

In this case the `Object(..)` phrase is dropped. The JPQL specification requires that you wrap your result with an `Object()` phrase only in that case where a stand-alone variable is returned without navigating along a path.

25.2 Statement types

The JPQL is syntactically similar to the Standard Query Language (SQL) in that it supports 3 types of statements:

1. **Select statements:** Select statements are used access to selected data in persistent storage,
2. **Update statements:** Update statements are used to modify information maintained in persistent storage.
3. **Delete statements:** Delete statements are used to remove information currently held in persistence storage.

25.2.1 Elements of JPQL query statement

The elements of a *select* statement are

- a *select clause* which determines the type of the objects or values returned (in JPQL the result set is always a collection of objects or values), where the objects are either retrieved entities which match the query or new objects whose fields were populated from the query,
- a *FROM clause* which constrains the domain from which the selection is done,
- an optional *WHERE clause* which may be used to constrain the collection of objects selected from that domain,
- an optional *GROUP BY clause* which enables one to group query results in terms of groups,
- an optional *HAVING clause* used in conjunction with the *GROUP BY* clause in order to filter over aggregated groups, and
- an optional *ORDER BY clause* enabling one to request an ordering from the returned result objects/values.

25.2.2 Elements of update and delete statements

The update and delete statements contain only the *UPDATE/DELETE* clause and an optional *WHERE* clause .

25.3 Polymorphism

JPQL statements are intrinsically polymorphic. All statement elements which apply to a target bean also apply to all its specializations. The result is often a polymorphic collection.

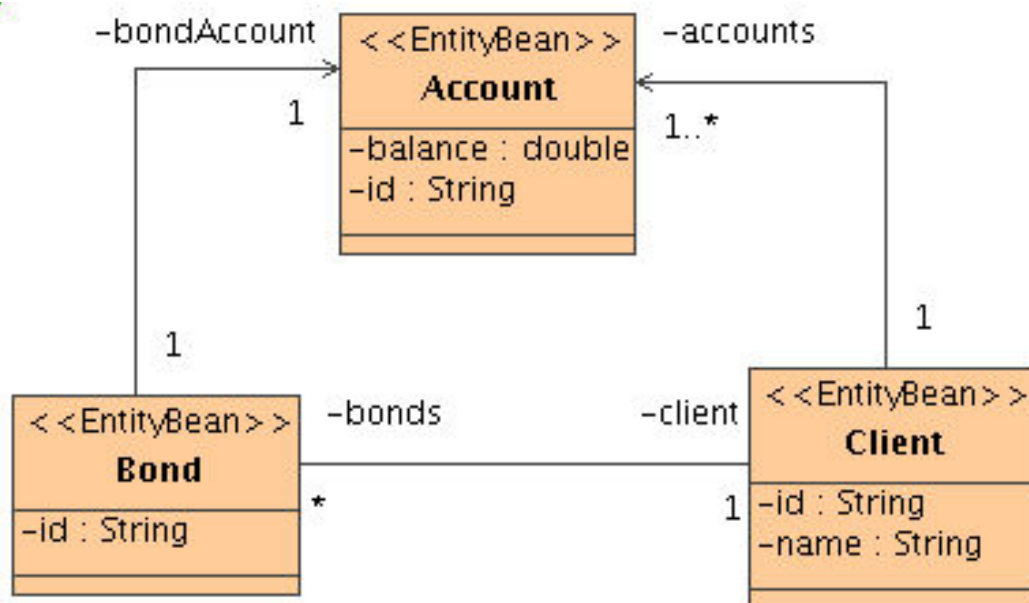
25.4 Navigating object graphs

In an object-oriented query language one queries along associations including aggregation, composition and specialization relationships as well as primitive fields.

25.4.1 Simple paths

Assume we have a *Bond* entity bean which has the structure shown in 25.1. One of the strengths of JPQL is its ability to smoothly navigate across relationships, i.e. through object graphs. Consider, as an example, the UML diagram for a bond shown in 25.1

Figure 25.1: UML class diagram for a bond



In JPQL one can traverse relationships in an object-oriented fashion using the Java element access operator. For example, we could specify the following **SELECT** statement to select all bond accounts

```
1 SELECT b.bondAccount FROM Bond b
```

returns a collection of accounts, each of which is a bond account. The `Bond` entity bean must supply an abstract accessor method to query the related bond account.

The equivalent SQL statement would look something like this

```
1 SELECT account from Account, Bond
2 WHERE Bond.bondAccount = Account.id
```

Our query may span multiple nodes like in

```
1 SELECT bond.bondAccount.balance FROM Bond bond
```

25.4.2 Single-valued versus multi-valued paths

A single valued path is a path without any branching below the highest layer (i.e. the layer connected to the result objects). `SELECT` clauses and most `WHERE` clauses require a single-valued path.

For example, all the queries discussed in the previous section use single valued paths in the `SELECT` statement and are hence valid JPQL statements. On the other hand, querying all the bond accounts of all the clients via

```
1 SELECT client.bonds.account FROM Client client --> INVALID
```

resembles a multi-valued path because `c` refers to a collection of clients each of which has a collection of bonds which each has an account.

As a second example, consider the UML diagram for a course schedule shown in figure 25.2. In a relational database this object graph could be represented as 4 tables, one for each entity.

If we wanted to extract all course names which are currently scheduled i.e. for which there exists a presentation), we could do this via the following SQL query:

```
1 SELECT Course.name from Course, Presentation WHERE Presentation.course = Course.id
```

To achieve the same in JPQL we can specify the following query:

```
1 SELECT p.course.name FROM Presentation p
```

This is a single-valued path and hence the query is valid. On the other hand, the query

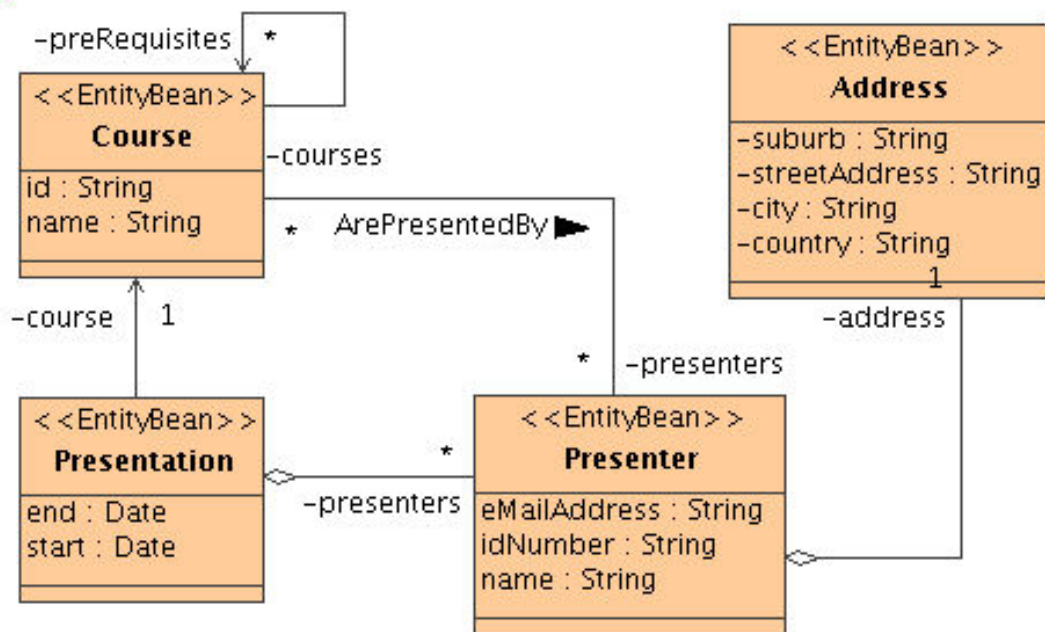
```
1 SELECT p.presenters.course.name FROM Presentation p --> INVALID
```

is incorrect because we have, once again, a multi-valued path.

25.5 Specifying the source of a query

The `FROM` clause specifies and constrains the domain of the query by specifying the domain as a particular entity type.

Figure 25.2: UML class diagram for presentations of courses for a course schedule



25.5.1 Selecting from multiple domains

The FROM clause supports selecting from multiple domains delimited by commas.

For example, should you wish to find all election results which had a greater attendance than South Africa's 1994 election you could specify the following query:

```

1 SELECT DISTINCT election
2 FROM Election election, Election election94
3 WHERE election.turnout > election94.turnout AND
4     election.country = 'South Africa' AND
5     election94.year = '1994'
  
```

25.5.2 Joins

An JOIN clause is used to combine two or more entities which have some common property.

25.5.2.1 Inner joins

Inner joins are used to select from an inclusion set obtained by a join condition over different entities. They can be specified implicitly via the Cartesian product or explicitly.

25.5.2.1.1 Implicit inner joins Implicit inner joins join multiple paths from multiple entities implicitly. For example, the following query performs an implicit inner join to determine those companies which are both customers and service providers:

```

1 SELECT DISTINCT c
2 FROM Customer c, ServiceProvider sp
3 WHERE c.companyRegistrationNo = sp.id

```

25.5.2.1.2 Explicit inner joins The following explicit INNER JOIN returns a collection of all bond accounts of clients living in Johannesburg:

```

1 SELECT bonds.account
2 FROM Client c INNER JOIN c.bonds bonds
3 WHERE c.address.city = 'Johannesburg'

```

Here the INNER JOIN can be abbreviated to JOIN (INNER is optional). This is equivalent to

```

1 SELECT bonds.account
2 FROM Client c IN (c.bonds) bonds
3 WHERE c.address.city = 'Johannesburg'

```

25.5.2.2 Left outer joins (fetch joins)

While an inner join retrieves only those objects which satisfy the join condition, an outer join does the same thing but with the addition of returning objects from the left collection for which there were no matching objects in the right collection.

For example, assume we have a *one-to-zeroOrOne* relationship between book and publisher, i.e a book may or may not be published by a publisher. Now assume you want to retrieve all book entities and load the publishers for those books which have publishers into the cache. We thus use an outer join to retrieve the set of entities where matching values in the join condition may be absent.

```

1 SELECT b FROM Book b LEFT JOIN b.publisher p WHERE p.address.country = South Africa

```

gets all books, irrespective of whether they do or do not have publishers and also loads all those publishers of books who are in South Africa into the cache.

25.6 Collapsing multi-valued paths into Single-valued paths

SELECT clauses are restricted to single-valued paths. The same is largely true for WHERE clauses. So, how do we handle queries along multi-valued paths?

In JPQL this is done by defining collection variables via an IN clause. Consider, for example the invalid query

```

1 SELECT client.bonds.account FROM Client client --> INVALID

```

The correct form of this query in JPQL is

```

1 SELECT bonds.account FROM Client c, IN(c.bonds) bonds --> VALID

```

Here the IN-clause defines a collection variable, **bonds**, which, for each client, resembles the client's bonds.

In a similar way we can fix the following invalid JPQL statement

```
1 SELECT p.presenters.course.name FROM Presentation p --&gt; INVALID
```

by defining a collection variable, `ps`, via an IN clause

```
1 SELECT ps.course.name FROM Presentation p, IN(p.presenters) ps --&gt; VALID
```

25.7 Constraining a result set via a WHERE clause

Analogous to SQL, EJB-QL uses a WHERE clause to restrict the elements returned in the result collection. For example, we can select only those courses to which one or more presenters have been allocated via

```
1 SELECT Object(c)
2 FROM Course c
3 WHERE c.presenters NOT EMPTY
```

25.7.1 Comparison operators

JPQL supports a relatively extensive set of comparison operators which can be used in where clauses:

- `==`, `<`, `>`, `<=`, `>=`, `<>`
- `BETWEEN`, `LIKE`, `IN`, `IS NULL`, `EMPTY`, `MEMBER OF` which can all be inverted by combining them with a
- logical operators `AND`, `NOT`, `OR`.
- ...

For example

```
1 SELECT a FROM album a WHERE a.year NOT BETWEEN 1980 AND 2005
2
3 SELECT s FROM soccer_club s where s.home.city IN ('London', 'Madrid', 'Rio de Janeiro')
4
5 select c FROM customer c WHERE c.email LIKE '%ac.za'
```

25.7.2 Calculation and logical operators

JPQL support arithmetic operators (`+` `-` `*` `/`), calculation operators (`MAX`, `MIN`, `SUM`, `MOD`, `AVG`, `COUNT`, `SQRT`) and a range of string operators (`LENGTH`, `LOCATE`, `SUBSTRING`, `UPPER`, `LOWER`, `CONCAT`).

25.7.3 Using collection variables in WHERE clauses

We often have to define collection variables for multi-valued path constraints in WHERE clauses. For example

```
1 SELECT Object(c)
2 FROM Course c
3 WHERE c.prerequisites.name = 'Programming in Java'
```

is invalid because of the match against a multi-valued path, while

```
1 SELECT Object(c)
2   FROM Course c, IN(c.prerequisites) p
3   WHERE p.name = 'Programming in Java'
```

25.8 Constructing result objects

The result of a JPQL query is always single object or collection of objects. Instead of returning persisted objects (e.g. entities, embedded objects, ...), the query can construct a collection of new objects which have been populated from information obtained from entities.

For example, the following JPQL query

```
1 SELECT NEW za.co.academics.UniversityInfo (u.name, u.address, c.name c.registeredStudents)
2   FROM University u JOIN u.course c WHERE c.registeredStudents > 100
```

creates a list of result objects populated from university and course entities.

25.9 Nested queries

JPQL supports nested queries, i.e. queries which have sub-queries embedded within the conditional expression of a **WHERE** or **HAVING** clause.

For example, to select the best student on a course, one could use

```
1 SELECT s FROM student s where s.courseResults.average
2   = (SELECT MAX(s.courseResults.average) from student s)
```

25.10 Ordering

One can use the **ORDER BY** clause followed by either **ASC** or **DESC** to request that the result set should be ordered in ascending or descending order of some field.

For example, to return a list of soccer stadiums which can seat at least 10000 spectators in the order of the number of spectators they can accommodate, one can use

```
1 SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY numSEATS DESC
```

To refine the sort order, one can use multiple comma-separated sort criteria which will be applied in the order in which they are defined:

```
1 SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY s.numSEATS DESC, s.age ASC
```

25.11 Grouping

The **GROUP BY** construct enables the aggregation of values according to a set of properties. The **HAVING** construct enables conditions to be specified that further restrict the query result.

For example,

```
1 SELECT j.publisher, count(j.circulation)
2 FROM journal j
3 GROUP BY j.publisher
4 HAVING COUNT(j.circulation) > 100000
```

selects all journals with a circulation of more than 100000 **Note:** *The expression which appears in the **GROUP BY** clause must appear in the **SELECT** clause.*

25.12 Query parameters

One may specify query inputs either as positional or as named parameters. The query input can only be used in the **WHERE** clause or **HAVING** clause of a query.

25.12.1 Positional parameters

Positional parameters are specified with a question mark (?) prefix followed by an integer designating the position of the parameter. Input parameters are automatically numbered, starting from 1. The same parameter can be used multiple times within a the same query.

```
1 SELECT sf FROM soccerFixture sf WHERE (sf.date >= :date1) AND (sf.date <= :date2)
```

25.12.2 Named parameters

Named parameters are case sensitive and their identifier is prefixed by the ":" symbol.

For example

```
1 SELECT sf FROM soccerFixture sf WHERE (sf.date >= :date1) AND (sf.date <= :date2)
```


Chapter 26

Constructing and executing queries

The entity manager provides an API for the persistence context and object-relational mapper. It is used to construct and execute queries:

```
1 List<Product> products = entityManager.createQuery
2     ("SELECT p FROM Product p WHERE p.description like :descr")
3     .setParameter("descr", description)
4     .setMaxResults(30)
5     .setFirstResult(pageNo*30)
6     .getResultList();
```

26.1 Named queries

Named queries are statically defined queries with predefined, unchangeable query strings. They are typically pre-compiled.

Named queries are defined via the `@NamedQuery` annotation. This could be in entities or in sessions beans.

```
1 @NamedQuery(name="bonds.getAllAbove"
2     query="select b from Bond b where b.balance >= :amount")
```

Named queries are instantiated and executed via

```
1 Query query = entityManager.createNamedQuery("bonds.getAllAbove");
2
3 query.setParameter(0, new Double(500000));
4
5 List<Bond> list = (List<Bond>)query.getResultList() ;
```


Chapter 27

JPA converters

At times the data type in the database differs from the data type used in Java entity objects. For example, one might use a boolean in the Java class and have a zero or one in the database column for that field. Another example is to convert a Java list to a comma-delimited string. Hence, upon accessing the persistent storage, we need to convert between the data type used in the database and the data type used in the Java class – i.e. we need *custom converters*.

27.1 Defining custom converters?

```
1 @Converter
2 public class BooleanToIntConverter implements AttributeConverter<Boolean, String>
3 {
4     public int convertToDatabaseColumn(Boolean value)
5     {
6         if (value)
7             return 1;
8         else
9             return 0;
10    }
11
12    public Boolean convertToEntityAttribute(int value)
13    {
14        return (value == 1);
15    }
16 }
```

27.2 Applying custom converters

```
1 @Entity
2 public class Order
3 {
4     ...
5     @Convert(converter=BooleanToIntConverter.class)
6     private Boolean priorityShipping;
7     ...
8 }
```

27.3 Default converters

At times a conversion should apply by default to all fields of a type. In that case one annotates the converter with `@Converter(autoApply=true)`:

```
1 @Entity
2 public class Order
3 {
4     ...
5     // No custom converter specified
6     private Boolean priorityShipping;
7 }
8
9 @Converter(autoApply=true)
10 public class BooleanToIntConverter implements AttributeConverter<Boolean, String>
11 {
12     public int convertToDatabaseColumn(Boolean value) {...}
13
14     public Boolean convertToEntityAttribute(int value) {...}
15 }
```

Chapter 28

Calling stored procedures

Similar to the specification of a JPA named query, we can also define a named query for a stored procedure.

```
1 @Entity
2 @NamedStoredProcedureQuery(name = ``calcAverageTemperature`, procedureName = ``CalcAvgTemp`)
3 public class WeatherReading
4 {
5     ..
6 }
```

We can then ask the entity manager to create us an instance of the named procedure query, register the stored procedure parameter types and set the corresponding parameter values. Finally we execute the query and extract the result:

```
1 StoredProcedureQuery query = EntityManager.createNamedStoredProcedureQuery("calcAverageTemperature");
2 query.registerStoredProcedureParameter(1, String.class, ParameterMode.IN);
3 query.setParameter(1, locationId);
4 query.registerStoredProcedureParameter(2, Timestamp.class, ParameterMode.IN);
5 query.setParameter(2, date1, TemporalType.TIMESTAMP);
6 query.registerStoredProcedureParameter(3, Timestamp.class, ParameterMode.IN);
7 query.setParameter(3, date2, TemporalType.TIMESTAMP);
8 ...
9 query.execute();
10 Double averageTemp = query.getSingleResult();
```


Chapter 29

The criteria API

29.1 Overview

The *JPA Criteria API* provides an object-based API for defining queries across object graphs as an alternative to String-based queries whose syntax is only checked at deploy time. One thus creates query objects

Due to the nature of constructing the queries in code through objects, they provide a simpler, more natural API for dynamic query construction within object-oriented code.

JPA queries were introduced with JPA 2.0

29.1.1 Benefits of using the criteria API for specifying queries

- **Simpler dynamic query construction:** The Criteria API provides a simpler way to dynamically and incrementally assemble a complex query allowing for reuse of query elements across queries. This is particularly true for complex queries.
- **Compile-time checking of queries:** JPQL queries are typically checked only at run-time. Criteria-based queries are validated at compile-time. Consider for example the following query:

```
1 String jpqlQuery = "select weatherReading from WeatherReading where weatherReading.temperature >  
40";  
2 Query query = em.createQuery(jpqlQuery);  
3 List<WeatherReading> result = query.getResultList();
```

The syntax error in the above query is not glaringly obvious and would only be spotted at either deploy-time or run-time (hopefully in the context of unit testing). The correct query string is, of course,

```
1 String jpqlQuery = "select weatherReading from WeatherReading weatherReading where weatherReading.  
temperature > 40";
```

On the other hand, when using the Criteria API, syntactically incorrect queries will result in compiler errors which are typically already highlighted by the IDE during coding.

- **Type-safe queries:** String-based JPQL queries are intrinsically not type-safe and one generally suppresses type-safety warnings via the corresponding annotation. For example,

```

1 @SuppressWarnings("unchecked")
2 public List<WeatherReading> getAllWeatherReadings()
3 {
4     Query query = entityManager.createNamedQuery("findAllWeatherReadings");
5     return query.getResultList();
6 }

```

Criteria-based queries can, on the other hand, be used in a type safe way.

29.1.2 Query tree

When using the criteria API to assemble a query, one assembles a query tree with a root node representing the starting (*from*) point for the query. The nodes of a query tree represent the semantic query elements such as

- WHERE clauses,
- GROUP BY or ORDER BY clauses,
- sub-queries,
- ...

29.2 Generating the JPA metamodel

Since JPA criteria queries are assembled from instances of classes which define the persistence metamodel, one needs to generate the metamodel classes from the entity definitions. This can be done by using a Java APT annotation pre-processor which generates the canonical metamodel classes. In the case of *EclipseLink* this is provided by `org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor`.

29.2.1 Maven build declarations to generate a canonical metamodel

We need to generate the metamodel classes. This should be done in the *generate-sources* life cycle phase so that the generated classes are available in the *compile* phase.

In the maven build we need to include

- A dependency on some JPA implementation (the JavaEE API does, of course, not provide them),
- A dependency on the object-relational mapper used (e.g. *EclipseLink*),
- Configuring the Maven compiler plugin to compile for Java 7 or later (supporting annotations processing),
- Configuring the Annotations processor plugin to execute the annotations processor used to generate the canonical model generation, e.g. *Eclipse*'s `CanonicalModelProcessor`.
- The required Maven repositories and plugin repositories.

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     ...
6     <repositories>

```



```

7      <repository>
8          <id>maven-repo</id>
9          <name>Maven repository</name>
10         <url>http://repo1.maven.org/maven2/</url>
11     </repository>
12
13     <repository>
14         <id>java.net</id>
15         <url>http://download.java.net/maven/</url>
16     </repository>
17
18     <repository>
19         <id>EclipseLink Repo</id>
20         <url>http://www.eclipse.org/downloads/download.php?r=1&nf=1&file=/rt/eclipselink/
21             maven.repo</url>
22         <snapshots>
23             <enabled>true</enabled>
24         </snapshots>
25     </repository>
26 </repositories>
27
28 <pluginRepositories>
29     <pluginRepository>
30         <id>maven-annotation-plugin</id>
31         <url>http://maven-annotation-plugin.googlecode.com/svn/trunk/mavenrepo</url>
32     </pluginRepository>
33     <pluginRepository>
34         <id>maven2-repository.dev.java.net</id>
35         <name>Java.net Repository for Maven</name>
36         <url>http://download.java.net/maven/2/</url>
37         <layout>default</layout>
38     </pluginRepository>
39 </pluginRepositories>
40
41 <dependencies>
42     <dependency>
43         <groupId>org.eclipse.persistence</groupId>
44         <artifactId>javax.persistence</artifactId>
45         <version>2.0.0</version>
46     </dependency>
47
48     <dependency>
49         <groupId>org.eclipse.persistence</groupId>
50         <artifactId>eclipselink</artifactId>
51         <version>${eclipselink.version}</version>
52     </dependency>
53
54     <dependency>
55         <groupId>javax</groupId>
56         <artifactId>javaee-api</artifactId>
57         <version>7.0</version>
58         <scope>provided</scope>
59     </dependency>
60 </dependencies>
61
62 <build>
63     <plugins>
64         <plugin>
65             <groupId>org.apache.maven.plugins</groupId>
66             <artifactId>maven-ejb-plugin</artifactId>
67             <version>2.3</version>
68             <configuration>
69                 <ejbVersion>4.0</ejbVersion>
70             </configuration>
71         </plugin>
72
73         <plugin>
74             <groupId>org.bsc.maven</groupId>
75             <artifactId>maven-processor-plugin</artifactId>
76             <executions>
77                 <execution>
78                     <id>process</id>
79                     <goals>

```

```

79         <goal>process</goal>
80     </goals>
81     <phase>generate-sources</phase>
82     <configuration>
83         <outputDirectory>${project.build.directory}/generated-sources/apt</
            outputDirectory>
84         <compilerArguments>-AeclipseLink.persistence.xml=src/main/resources/META-INF/
            persistence.xml</compilerArguments>
85         <processors>
86             <processor>org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor
            </processor>
87         </processors>
88     </configuration>
89 </execution>
90 </executions>
91 </plugin>
92
93 <plugin>
94     <groupId>org.apache.maven.plugins</groupId>
95     <artifactId>maven-compiler-plugin</artifactId>
96     <version>2.3</version>
97     <configuration>
98         <source>8</source>
99         <target>8</target>
100    </configuration>
101 </plugin>
102
103 </plugins>
104 </build>
105 </project>

```

29.2.2 Generated metamodel classes

The purpose of the metamodel classes is to provide a infrastructure through which queries can be assembled from an object tree. They provide singular and collection attributes used to navigate the object graph.

```

1  package za.co.solms.weather;
2
3  import javax.annotation.Generated;
4  import javax.persistence.metamodel.SingularAttribute;
5  import javax.persistence.metamodel.StaticMetamodel;
6  import za.co.solms.location.Location;
7
8  @StaticMetamodel(WeatherReading.class)
9  public class WeatherReading_
10 {
11     public static volatile SingularAttribute<WeatherReading, Integer> id;
12     public static volatile SingularAttribute<WeatherReading, Double> humidity;
13     public static volatile SingularAttribute<WeatherReading, Date> dateTime;
14     public static volatile SingularAttribute<WeatherReading, Location> location;
15     public static volatile SingularAttribute<WeatherReading, Ambiance> ambiance;
16     public static volatile SingularAttribute<WeatherReading, Double> temperature;
17 }

```

29.3 Simple example

In this section we show a simple criteria-based query in order to introduce some of the core concepts. The simple query will resolve all weather readings for a particular location.

```

1  public List<WeatherReading> getWeatherReadingsForLocation(Location location)
2  {
3      CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

```

```

4 CriteriaQuery<WeatherReading> criteriaQuery
5   = criteriaBuilder.createQuery(WeatherReading.class);
6
7 Root<WeatherReading> weatherReading = criteriaQuery.from(WeatherReading.class);
8
9 Predicate predicate
10  = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
11 criteriaQuery.where(predicate);
12
13 TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
14 return query.getResultList();
15 }

```

In the above listing we

- ask the entity manager for a criteria builder and use it to create a criteria query which yields weather readings,
- specify the root (FROM domain) of the query,
- create a predicate which affirms weather readings which have the required location and add a WHERE node to the query using that predicate, and
- create a *typed* JPQL query from the criteria query and execute it, returning the type-safe result list.

29.4 Query operators

The query builder provides a range of query operators including

- *arithmetic operators* like `sum`, `diff`, `prod`, `quot`, `min`, `max`, `avg`, `abs`, and `sqrt`,
- *relational operators* like `gt`, `ge`, `lt`, `le`, `equal`, `like` and `notLike`,
- *logical operators* like `and`, `or`, `xor`, `not`
- *set and collection operators* like `count`, `countDistinct`, `isEmpty`, `between`, `isMember`, `isNotMember`, `exists(subQuery)`, `any(subQuery)`, `all(subQuery)`
- *sorting operators* like `asc`, `desc`
- *date/time operators* like `currentDate`, `currentTime`, and `currentTimeStamp`,
- *string operators* like `upper`, `lower`, `concat`, `substring`, and `trim`,
- *data conversion operators* like `toDouble`, `toInteger`, `toLong`, `toString`,
- and some general `testOperators` like `isNull`, `isNotNull`, `isTrue`, and `isFalse`.

29.5 Composite predicates

The criteria builder can be used to create composite logical expressions from multiple predicates. For example, if one wants to find all weather readings for a particular location for which the temperature was above 40 °C, it can be done as follows:

```

1 Predicate atLocation = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
2 Predicate tempGe40 = criteriaBuilder.ge(weatherReading.get(WeatherReading_.temperature), 40);
3 criteriaQuery.where(criteriaBuilder.and(atLocation,tempGe40));
4 TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
5
6 return query.getResultList();

```

29.6 Ordering

To request ordering, we add a `orderBy` node to the query tree, supplying it an `ascending` or `descending` operator obtained from the criteria builder:

```

1 CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
2 CriteriaQuery<WeatherReading> criteriaQuery
3     = criteriaBuilder.createQuery(WeatherReading.class);
4
5 Root<WeatherReading> weatherReading = criteriaQuery.from(WeatherReading.class);
6
7 Predicate atLocation
8     = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
9 criteriaQuery.where(atLocation);
10
11 criteriaQuery.orderBy(criteriaBuilder.desc(weatherReading.get(WeatherReading_.temperature)));
12
13 TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
14 return query.getResultList();

```

29.7 Joins

If queries are based on multiple entities one may need to use joins. For this purpose JPA introduces the `Join` and `SetJoin` classes, both of which are generic, taking the joining `from` and the binding type as template parameters.

For example, if we want to find all students who have one or more enrollments which are not canceled, we could use the following join:

```

1 CriteriaQuery<Student> q = cb.createQuery(Student.class);
2 Root<Student> c = q.from(Student.class);
3 SetJoin<Student, Enrollment> o = c.join(Student_.enrollments);
4
5 Predicate p = cb.equal(o.get(Enrollment_.status), Status.Canceled).negate();
6 c.where(p);

```

Part V

Enterprise Java Beans

Enterprise Java Beans (EJB) is a managed server-side component architecture supporting modular construction and deployment of enterprise application. In its core there is an EJB container which is mean to be a business logic container providing a process execution environment for (business) processes.

Chapter 30

Session beans

Session beans are the central components for the EJB container. The application server maintains bean pools for deployed session beans which act as thread pools.

Session beans receive and propagate session context. They are either stateless (no state maintained across service requests) or stateful (state maintained across service requests). The latter incur activation/passivation overheads.

Stateless session beans should be used for business logic whilst stateful session beans should be used only in exceptional cases like that of developing certain infrastructural components.

Components are decoupled through business interfaces. For remote access, there is a remote interface and for local access there is a local interface.

30.1 Remote interfaces

Remote clients (clients running in a different Java Runtime Environment) access enterprise beans through remote interfaces. Examples of remote clients include client applications (e.g. Java-FX or Swing clients), JSF managed beans, servlets and beans running in other runtime environments (e.g. on another node of a cluster).

30.1.1 Defining remote interfaces

There are two ways of indicating that a particular interface should be used in the role of remote interface. The first is to classify the interface itself with the `@Remote` EJB annotation. For example

```
1 import javax.ejb.*;
2
3 @Remote
4 public interface OrderProcessorRemote
5 {
6     public Invoice processOrder(Order order)
7         throws InsufficientFundsException;
8 }
```

The Bean realising the interface may now simply state that it implements it:

```
1 @Stateless
2 public class OrderProcessorBean implements OrderProcessorRemote
3 { ... }
```

The second is to have a plain interface

```

1 public interface OrderProcessor
2 {
3     public Invoice processOrder(Order order)
4         throws InsufficientFundsException;
5 }

```

and assign that interface as `@Remote` interface to the bean implementation class (stateless session bean:)

```

1 import javax.ejb.*;
2
3 @Stateless
4 @Remote({OrderProcessor.class})
5 public class OrderProcessorBean implements OrderProcessor
6 { ... }

```

The latter approach is preferable, as the interface is now a pure contract, which could even be realised using technology other than EJB, without affecting existing clients.

30.1.2 Access path when using a remote interface

Service requests provided via the local interface still need to be intercepted by the server-generated `EJBObject`, in order to still apply enterprise services. However, the service request messages need no longer be marshaled onto the RMI/IIOP protocol:

1. Client receives a handle to a generated *RMI/CORBA client stub* (potentially via dependency injection or JNDI lookup) and makes the request against the client stub.
 - potentially via dependency injection.
2. The *client stub* serializes request onto RMI/IIOP/TCP/IP.
3. A generated `EJBRemoteObject` acts as server side proxy which receives and de-serializes the request.
4. The `EJBRemoteObject` applies enterprise services like security and transaction (e.g. checking whether the user has the security roles required to access the requested service, starting a transaction and enlisting resources within that transaction, ...)
5. An instance of the bean implementation class is retrieved from the bean pool and associated with the session (the session bean is activated).
6. The request is forwarded to the bean implementation class instance which applies the business logic.
7. The `EJBRemoteObject` receives the response and applies further enterprise services (e.g. commit a transaction on successful completion or roll it back otherwise).
8. `EJBRemoteObject` applies further enterprise services upon completion (e.g. commit a transaction on successful completion or roll it back otherwise).
9. Finally `EJBRemoteObject` serializes the response (including exceptions) onto RMI/IIOP/TCP/IP.
10. The client stub de-serializes the response and either provide the return value to the client or raises an exception.

30.2 Local interfaces

A local interface provides access to an enterprise bean from within the same run-time environment in which the bean itself is deployed. This enables local clients like other enterprise beans and servlets deployed within a servlet container running in the run-time environment of the application server to obtain efficient access to the bean instance, bypassing the marshaling of the request onto an IIOP message.

30.2.1 Defining local interfaces

As with remote interfaces, there are two ways of indicating that a particular interface should be used in the role of local interface. The first is to classify the interface itself with the `@Local` EJB annotation. For example

```
1 import javax.ejb.*;
2 @Local
3 public interface ShippingRequestProcessor
4 {
5     public TrackingNumber ship(Order order);
6 }
```

The second is to leave the interface unclassified, and classify the role the interface plays to the bean within the bean itself, using the `Local` annotation:

```
1 import javax.ejb.*;
2
3 @Stateless
4 @Local({ShippingRequestProcessor.class})
5 public class ShippingBean implements ShippingRequestProcessor
6 {
7     public TrackingNumber ship(Order order)
8     {
9         //...
10    }
11 }
```

30.2.2 Access path when using a local interface

Service requests provided via the local interface still need to be intercepted by the server-generated `EJBObject`, in order to still apply enterprise services. However, the service request messages need no longer be marshaled onto the RMI/IIOP protocol:

1. Client receives a handle to a generated *EJBLocalObject* via dependency injection or JNDI lookup and makes the request against it.
2. The *EJBLocalObject* applies enterprise services like security and transaction (e.g. checking whether the user has the security roles required to access the requested service, starting a transaction and enlisting resources within that transaction, ...)
3. An instance of the bean implementation class is retrieved from the bean pool and associated with the session (the session bean is activated).
4. The request is forwarded to the bean implementation class instance which applies the business logic.
5. The *EJBLocalObject* receives the response and applies further enterprise services (e.g. commit a transaction on successful completion or roll it back otherwise).

6. `EJBLocalObject` applies further enterprise services upon completion (e.g. commit a transaction on successful completion or roll it back otherwise).
7. Finally `EJBLocalObject` either provide the return value to the client or raises an exception.

Note: *Bypassing the message serialization will typically result in a significant performance benefit within the application server - for example accessing the beans from a local web application.*

30.3 Switching between local and remote interfaces

EJB does not support automatic switching between local and remote interfaces. The reason for this is that it would be unsafe, as the different parameter handling semantics may result in changes in logic.

30.3.1 Parameter handling for plain Java objects

Java only supports input parameters, i.e. they are copied from the client to the server upon service request and never back. Method parameters may be either primitives (like `int`, `boolean` or `double`) or object references. In either way they are copied to the client. In the case of an object reference, the client using that reference (without changing the value of the reference itself) will access the actual object and potentially change its state. It is through this mechanism that Java simulates output and input/output parameters.

30.3.2 Parameter handling for remote Java objects (RMI)

When accessing remote Java objects, primitive parameters are also treated as input arguments. For object parameters, there are, however, two scenarios.

1. If the argument is itself a remotely accessible object (i.e. a RMI server which implements `java.rmi.UnicastRemoteObject`, then the client will receive a remote reference through which the object itself can be manipulated.
2. Otherwise the class for that object must implement `java.io.Serializable` and the object will be serialized onto the RMI stream upon service request. The object will not be sent back upon service completion.

From the above we see that objects which are RMI servers are treated the same across local and remote interfaces, but other objects are not. Changing a local to a remote interface and vice versa is only safe if the client only requires read access. **Note:** *An effective way of enforcing this is by making the argument an interface which provides clients read-only access to the underlying object.*

30.4 Automatically generated interfaces

30.4.1 Automatic interface naming

Beans for which interfaces are to be generated automatically must be named with the one of the following suffixes:

```
1 <MyBeanName>Impl
2 <MyBeanName>Implementation
3 <MyBeanName>Bean
4 <MyBeanName>EJB
```

The application server will generate an interface with the same name as the bean implementation class, but with the suffix dropped. Thus, the automatically generated business interface for `AccountBean` would be named `Account`.

30.4.2 Which services are published in the automatically generated interface?

The application server will, by default, publish all public methods (except those used by the application server for dependency injection.)

This default can, however, be overridden by annotating those methods which should appear in the interface with the `@BusinessMethod` annotation. As soon as any method is annotated using `@BusinessMethod`, all other non-annotated public methods will no longer be included in the application server generated business interface.

30.4.3 Automatically generated local interfaces

The application server will generate automatically a local interface for any session bean class which is not annotated as `@Remote`.

30.4.4 Automatically generated remote interfaces

To request an automatically generated remote interface, one must annotate the bean implementation class with the `@Remote` annotation.

30.4.5 Should one automatically generate interfaces?

Though a useful feature, if one follows a typical design process such as URDAD, the *contracts* are usually the first coding deliverable, which means that the Java interfaces should already be derived even before design decisions are made such as the types of components to be used (e.g. EJB).

In this spirit, it is usually a better idea to always write (or generate from UML) your contracts as the result of the *design process*, and not by “reverse engineering” them from an implementation. You can then use your IDE to generate a shell for a bean implementation class from your interface.

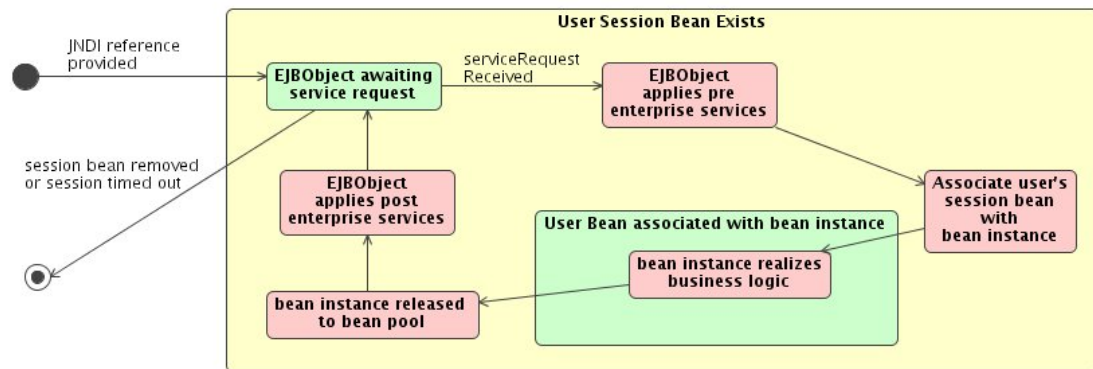
30.5 Stateless session beans

Stateless session beans provide users a services interface without maintaining state across service requests, i.e. it is an aggregation of stateless services.

30.5.1 Life cycle of a stateless session bean

For stateless session beans, no state is maintained across service requests. This results in very simple bean activation and passivation, i.e. a simple, efficient process for assigning a physical object to a virtual user object and releasing the physical object back to the bean pool.

Figure 30.1: Life cycle of a stateless session bean



30.6 Stateful session beans

Stateful session beans maintain state across service requests. The state services only for the duration of the session and is only accessible from within the user session.

In order to facilitate the representation of potentially many stateful session bean objects by a limited number of physical bean instances, EJB makes use of the flyweight pattern. Object identity and state is externalised; the EJB-Object maintains an external identity for the conceptual user session bean which may be realised by different physical bean instances over time and the persistent storage maintains the beans state externally.

Object serialisation is used to grab a memento (the encapsulated state of the bean instance) without breaking encapsulation as the serialised state can only be used to reconstruct the bean instance state.

30.6.1 Life cycle of a stateful session bean

Stateful session beans maintain state for the duration of the user session across bean passivation and re-activation. That requires that the state of the bean's object identity and state needs to be persisted upon bean passivation (i.e. when the bean is released to the bean pool) and reconstructed upon re-activation.

For every service request the application server activates any bean from the method ready pool by binding the flyweight object to the user session bean object. After having applied the enterprise services, the application server delegates the realisation of that request to the flyweight.

The following image illustrates the life cycle of a stateful session bean including the passivation and re-activation of the physical bean instance.

30.7 Asynchronous session bean services

In order to support processing of asynchronous service requests you could always make use of message driven beans. The client thread returns as soon as the message has been successfully delivered to the queue or topic.

As of EJB 3.1, there is support for session beans processing asynchronous requests without the need of transmitting a message via a queue or topic.

A service is declared an *asynchronous service* by annotating it with an `@asynchronous` annotation. For example, the `OrderProcessorBean` shown below provides a normal `processOrder` service which is processed synchronously (within the calling thread), but it makes use of an asynchronous `shipOrder` service. The `processOrder` service will typically return with the `processOrderResult` prior to the `shipOrder` service having been completed:

```

1 @Stateless
2 public class OrderProcessorBean
3 {
4     @Asynchronous
5     private void shipOrder(OrderShipmentRequest orderShipmentRequest)
6     { ... }
7
8     public ProcessOrderResult processOrder(ProcessOrderRequest processOrderRequest)
9     { ...
10         shipOrder(shipOrderRequest);
11         return processOrderResult;
12     }
13 }

```

30.7.3 Deferred asynchronous requests

The `@asynchronous` annotation can also be used for processing of deferred synchronous requests, i.e. for services which are requested asynchronously, but for which, at a later stage the return value needs to be obtained. In Java this is done by having the asynchronous service return a `Future` from which the result is obtained later stage.

Consider, for example, a process which needs to obtain prices for multiple service providers (e.g. airlines, hotels, ...) and then assemble an optimal itinerary for a client (optimized say first on price, then travel time and so on). In this case the process of booking an optimal itinerary would source multiple quotes through deferred synchronous requests made to different service providers, block until sufficient quotes have been obtained and then assemble the optimal itinerary. In such an example, the `provideServiceProviderQuote` service would be annotated as `@asynchronous` returning a `Future` (i.e. be a deferred synchronous service):

```

1 @Stateless
2 public class BookingServicesBean
3 {
4     @asynchronous
5     public Future<QuoteResult> provideQuote(QuoteRequest quoteRequest)
6     { ... }
7
8     public BookItineraryResult bookItinerary(BookItineraryRequest request)
9     { ...
10         Future<QuoteResult> futureQuoteResult = this.provideQuote(quoteRequest);
11         ...
12         // do some other things (eg request further quotes)
13         ...
14         // Now block process until result is available
15         QuoteResult result = futureQuoteResult.get();
16         ...
17     }
18 }

```

30.7.4 Asynchronous beans

Asynchronous beans are beans for which all services are processed either asynchronously or in a deferred synchronous way, i.e. by annotating a session bean as `@Asynchronous` one specifies that all its services are asynchronous (the ones which return a future are deferred synchronous).

For example, we could annotate the `BookingServicesBean` as `@asynchronous`, thereby specifying that both, the `provideQuote` and `bookItinerary` services be asynchronous (or, in this case, deferred synchronous services):

```

1 @Asynchronous
2 @Stateless
3 public class BookingServicesBean
4 {
5     public Future<QuoteResult> provideQuote(QuoteRequest quoteRequest) {...}
6
7     public BookItineraryResult bookItinerary(BookItineraryRequest request)
8     { ...
9         Future<QuoteResult> futureQuoteResult = provideQuote(quoteRequest);
10        ...
11        // do some other things (eg request further quotes)
12        ...
13        // Now block process until result is available
14        QuoteResult result = futureQuoteResult.get();
15        ...
16    }
17 }
```

All resources including connection resources should be managed by the application server. Hence one should not open directly a connection to a mail server, but request a mail resource from the application server instead.

30.8 Registering a connection pool to an email server

This is typically done through the management console of your application server. On Glassfish the resultant mail service descriptor is added to the `config/domain.xml` domain descriptor:

```

1 <mail-resource host="solms.co.za" store-protocol="pop3" store-protocol-class="com.sun.mail.imap.
   POP3Store"
2     jndi-name="mail/solms" from="appServer@solms.co.za" user="appServer" >
3     <property name="mail.smtp.auth" value="true" />
4     <property name="mail.smtp.password" value="appServerPassword" />
5     <property name="mail.smtp.user" value="sbss" />
6     <property name="mail.smtp.port" value="425" />
7 </mail-resource>
```

30.9 Sending an email from an enterprise bean

Sending an email from a session or message-driven bean is no different than sending one from any other Java code except that the email server session is not to be created within the code, but rather to be provided (via dependency injection) by the application server:

```

1 public void emailAuthenticationCredentials(String emailAddress)
2     throws NoPersonWithThatEmailAddressException, MessagingException
3 {
4     List<Person> persons = this.getPersonsWithEmailAddress(emailAddress);
5     if (persons.size() == 0) throw new NoPersonWithThatEmailAddressException();
6
7     Transport transport = mailSession.getTransport();
8
9     for (Person person:persons) {
10         User user = person.getUser();
11
12         MimeMessage message = new MimeMessage(mailSession);
13         message.setSubject("Password notification");
```

```
14     message.setText("Your username and password for solms.co.za are \n" +
15                     "username: " + user.getUsername() + "\npassword:" + user.getPassword());
16     message.setFrom(new InternetAddress("info@solms.co.za"));
17     message.addRecipient(Message.RecipientType.TO,
18                          new InternetAddress(emailAddress));
19
20     transport.connect();
21     transport.sendMessage(message, message.getRecipients(Message.RecipientType.TO));
22     transport.close();}
23 }
24 @EJB
25 private UserServices userServices;
26
27 @Resource(name = "mail/solms")
28 private Session mailSession;
29 }
```

Chapter 31

Interceptors

Interception is a powerful feature that, for any managed component in a Java EE container, provides an extension point for future pluggable extensions around the services these components provide. This results in improved maintainability, as the classes themselves can be kept simple (the extra logic is not absorbed within the class itself).

The EJB specification caters for two features through interception

- *dependency injection* where the container injects dependencies on resources it manages, and
- *interceptor methods and classes* used to intercept a bean's business services

31.1 When should one use interceptors?

Interceptor classes are useful when one wants to add additional responsibilities to a service, which you do not want to include in the responsibility domain of the class(es) whose services you are intercepting.

Interceptors thus add additional optional work flow steps around some core service or across a range of services. Interception provides a mechanism to weave these additional work flow steps into your code, applying it across the application domain of the interceptor. In that context they are similar to simple aspects and can often be used instead of aspects.

31.2 Defining interceptors

Interceptors may be defined as interceptor methods within a bean class or as separate interceptors which can be applied across beans. In either case the interceptor is provided an interception context which it can inspect to obtain information about the service request which is being intercepted. One can also use dependency injection in interceptors to, for example, inject a message queue or a stateless session bean.

31.2.1 The interception context

The interceptor invocation runs within the *transaction and security context* of the business service it intercepts, and it may access those objects just like any session bean (i.e. by having them injected). The interceptor method, however, receives an instance of `javax.interceptor.InvocationContext`, which it may use to:

- get a handle to target object, i.e. the object which would have received the service request message had it not been intercepted, via `getTarget():Object`,
- get a handle to the method it currently intercepts via the `getMethod():Method` service,
- get a handle to the parameters of the service it intercepts via `getParameters():Object[]`,
- change the parameters which are passed on to the actual business service via `setParameters(Object[] params)`,
- get a handle to the context data (a map which interceptors may use to place any information to be shared between interceptors, and other components) via `getContextData():Map<String,Object>`, and
- pass control to the next interceptor (if there is one) or alternatively to the business service itself via `proceed():Object`.

31.2.2 Interceptor methods

Interceptor methods can be defined directly in the bean class itself. This provides switchable additional functionality around the core bean service. They must have the following signature:

```

1 import javax.ejb.*;
2 import javax.interceptor.*;
3
4 @Stateless
5 @Remote({SomeInterface.class})
6 public class SomeBean implements SomeInterface
7 {
8     ...
9     /** This is an interceptor method */
10    @AroundInvoke
11    public Object intercept( InvocationContext ctx )
12    {
13        // Here the interception logic
14        ...
15        //
16        // delegate control to next object in chain of interceptors:
17        return ctx.proceed();
18    }
19 }
```

An interceptor method, irrespective of whether it is defined within a bean class or in a separate interceptor class, will thus obtain an invocation context as parameter and must return `invocationContext.proceed()`.

Calling `proceed` on the invocation context passes control to the next interceptor (if there is one) or alternatively to the business service being intercepted.

An interceptor method can choose to refuse forwarding to intercepted method by simply not returning `invocationContext.proceed()`, but directly returning the return value for the service it is intercepting (without the service which is intercepted actually being called) or by throwing an exception.

31.2.3 Interception classes

31.2.3.1 Introduction

Commonly an interceptor is a separate responsibility which should be specified in a separate class, an interceptor class. An interceptor class is simply a class with a default constructor which

has one or more interceptor methods annotated with `@AroundInvoke`. These methods receive an `InvocationContext` and return an `Object`.

```

1 import javax.interceptor.*;
2 public class BankingFeeInterceptor
3 {
4     @AroundInvoke
5     public void applyTransactionFee(InvocationContext invocationContext)
6     {
7         invocationContext.getTarget().debit(transactionFee);
8         return invocationContext.proceed();
9     }
10 }
```

Applying the interceptor class to a method results in all interceptor methods of the interceptor being applied to the intercepted method:

```

1 import javax.interceptor.Interceptors;
2
3 public class Account
4 {
5     @Interceptors({za.co.solms.finance.BankingFeeInterceptor})
6     public TransactionConfirmation debitAccount(double amount)
7     { ... }
8 }
```

31.2.3.2 Interceptor life-span and state

An interceptor has the same life span as the context whose service is intercepted. An interceptor for a session bean has hence the life span of the session. Such an interceptor is passivated when the session bean is passivated and activated when the session bean is activated. The normal callback services may be applied to an interceptor, i.e. one may annotate methods as `@PostConstruct`, `@PreDestroy`, `@PostActivate` or `@PrePassivate`.

On the other hand, an interceptor applied to a message driven bean exists while the service request is being processed. The state of an interceptor can thus survive multiple invocations.

31.2.3.3 Interceptor exceptions

Interceptors may throw

- any checked exception thrown by the business service it intercepts, and
- any runtime exceptions in order to indicate an error the inability to realize the contractual obligations.

Any exception thrown within a transactional context which is not caught and handled prior to reaching the transaction boundary will result in *transaction roll-back* .

31.2.3.4 Interceptors specialization

An interceptor class may subclass another interceptor class, inheriting further interceptor methods. These can be overridden and further interceptor methods may be added by the specialized interceptor.

When the interceptor class is applied, all inherited interceptor methods are applied first before any specialized interceptor methods are applied.

31.3 Interception order

If multiple interceptors are applicable for a particular service request, then they are applied in the following order

1. Default interceptors are applied first in order of their specification in the deployment descriptor.
2. Class level interceptors are applied prior to method level interceptors.
3. Interceptor classes are applied prior to interception methods. They are applied in the order they are requested in either the bean annotation or the deployment descriptor.
4. If the interception context (e.g. the bean to which the interception is applied) has a superclass then the superclass interceptions take preference to the subclass interceptions.
5. The `@AroundInvoke` method of a superclass of an interceptor class is applied prior to that of the subclass. **Note:** *There may be only a single `@AroundInvoke` method in any class, though an interceptor class. However, applying an interceptor class can still result in multiple interceptions for a single service request as the interceptor methods of the superclass of the interceptor class are also applied. These are applied in order of the class hierarchy with superclass interceptor methods taking preference above subclass interceptor methods.*

31.4 Default interceptors

At times one may want to weave in additional work flow steps across all business services of all session and message-driven beans within a module or application, i.e. across all business services published within a particular domain. This could be useful, for example, for auditability (e.g. logging). This is where default interceptors become useful.

Default interceptors cannot be specified via annotations. They must be declared in the `ejb-jar.xml` deployment descriptor for the domain, and this is done by specifying `*` (all) as the `ejb-name` which is targeted.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <ejb-jar xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-
   jar.3.0.xsd" version="3.0">
3
4   <interceptors>
5     <interceptor>
6       <interceptor-class>
7         za.co.solms.auditing.logging.RequestLogger
8       </interceptor-class>
9     </interceptor>
10  </interceptors>
11
12  <assembly-descriptor>
13    <interceptor-binding>
14      <ejb-name>*</ejb-name>
15      <interceptor-class>
16        za.co.solms.auditing.logging.RequestLogger
17      </interceptor-class>
18    </interceptor-binding>
19  </assembly-descriptor>
20 </ejb-jar>

```

31.5 Exercises

The intelligence agency of the western world's foremost economical power is very paranoid about news agencies publishing bad publicity regarding their president.

- Discuss three potential uses of EJB filters which have not been mentioned in the course notes thus far.
- Write a stateless session bean which offers a service of accepting news articles from journalists working remotely in the field. A news article should at least have a headline, a body of text, and a date. The bean does not have to do anything with the article for now, but a real implementation may store it persistently (using JPA) or place it on a message queue for approval.
- Write an interceptor which is applied as a method-level interceptor to the news submission service: If a submitted news article contains any mention of the country's president, it should not prevent the article from being submitted, but it should indicate that it has logged this incident (and probably that the journalist in question will receive a visit from an unmarked, black government vehicle and be taken away for questioning).
- Write a default interceptor (applied to all beans in the module) which, for any news article being sent as a parameter (or return value) to/from any service, will perform bad language filtering: Any occurrence of a bad word (from a list of bad words) should silently be replaced by "***" once it reaches its destination.

Chapter 32

The Java Messaging Service (JMS)

32.1 Introduction

The Java Messaging Service (JMS) is an open API which enables you to interact with any implementation of the JMS in a vendor-neutral way. Code thus remains portable across different Message-Oriented Middleware (MOM) implementations like IBM WebSphere MQ, Oracle Tuxedo/Q, Progress SonicMQ, Microsoft MSMQ and others. These different vendors all supply JMS drivers which provide the coupling to their specific implementation. All Java EE application servers (such as JBoss or Glassfish) are packaged with their own JMS-compliant messaging service, but they can of course be configured to use any other compliant service.

32.2 Styles of Messages: Point-Point vs Publish-Subscribe Domains

MOMs have traditionally supported two types of messaging mechanisms, point-to-point messaging and publish-subscribe messaging.

32.2.1 The Point-to-Point Messaging Domain

Point-to-point messaging is used when you send messages which should be consumed only by a single consumer. Messages are sent to specific message queues and consumers extract messages from these queues. **Note:** *Multiple producers may send messages to the same queue and that multiple consumers may register with the same queue. Each message will, however, only be processed by a single consumer.* Point-to-point messaging is in some ways similar to making a telephone call. Multiple persons may have access to the same telephone but only a single person will answer a particular call (process a particular message).

32.2.1.1 Characteristics of point-to-point messaging

- Each message has only one consumer.

- Receivers (message consumers) can fetch messages irrespective of whether the receiver was running when the message was sent or not.
- Receivers acknowledge successful processing of messages.

32.2.2 The Publish-Subscribe Messaging Domain

Publish subscribe messaging is used when

- you want to decouple clients from service providers or
- when the message can be processed by zero or more consumers.

In the case of publish-subscribe messaging, publishers publish messages under a topic and each message is received by all subscribers to that topic.

32.2.2.1 Characteristics of publish-subscribe messaging

- Each message may have multiple consumers which are registered with a particular topic.
- If no consumer is registered with that topic, the message is discarded.
- Normal subscribers (message consumers) only receive messages which were sent after while they were registered with a topic and they must remain registered until they consume the message.
- In JMS subscribers may create durable subscriptions with topics which enables subscribers to receive messages which were sent while they were not active.

32.3 Message Types

The JMS specification supports the following 5 messages types, which are represented by the corresponding Java interface in the package `javax.jms`:

- **BytesMessage:** This is the most primitive message which can contain, in principal, anything. However, publishers and message consumers must agree on some data format/protocol for this to be useful. Bytes messages offer services similar to that obtained when combining byte input/output streams with data input/output streams, i.e. one can read or write bytes, arrays of bytes or primitives.
- **TextMessage:** Text messages are text-string based messages offering `setText()` and `getText()` services.
- **StreamMessage:** Is used to transmit a sequence of primitive data types (similar to a record containing primitive fields). The primitive data types may correspond to the column types of database table or to a file structure. It supplies an interface which is very similar to the combination of a `DataInputStream` and a `DataOutputStream` offering services like `readDouble()` and `writeInt(value)`.
- **ObjectMessage:** An object message is used to send serialized Java objects. An `ObjectMessage` offers the services `getObject()` and `setObject(Object o)`.

- **MapMessage:** Map messages enable publishers to send a map of key-value pairs with the keys restricted to strings, but the values may be strings, primitives or objects. Map messages supply set services like `setInt(name, value)` or `setObject(name, obj)` and query services like `getObject(name)`.

32.4 Using JMS queues and topics

32.4.1 General Algorithm for Connecting to a Queue or Topic

When connecting to a message queue or topic, the following steps need to be performed:

1. Look up, via JNDI, a `javax.jms.ConnectionFactory`. Specifically, either a `QueueConnectionFactory` or `TopicConnectionFactory` will be looked up based on the type of messaging scenario (point-to-point or publish-subscribe).
2. Create a message sending/receiving connection via `ConnectionFactory.createConnection()`
3. Look up, via JNDI, the `javax.jms.Queue` or `javax.jms.Topic` which will be used to send/receive messages to/from.
4. Open a `javax.jms.Session` to the connection, via `Connection.createSession(...)`
5. Via the session, create a message producer (to send messages) or a message consumer (to receive messages).

For example:

```

1 // Look up a connection factory to send messages on a queue
2 QueueConnectionFactory factory = (QueueConnectionFactory) new InitialContext().lookup("
   MyQueueConnectionFactory");
3
4 // Look up our message queue
5 Queue queue = (Queue) new InitialContext().lookup("MyQueue");
6
7 // To send messages, open a message sending session (in this case, with authentication)
8 QueueConnection connection = factory.createQueueConnection("username", "password");
9 QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
10
11 // Create a sender which will send messages on our queue
12 QueueSender sender = session.createSender( queue );
13
14 // Send a message
15 TextMessage msg = session.createTextMessage("Hello!");
16 sender.send(msg);

```

32.4.2 Using JMS queues

Queues are usually used for asynchronous service requests. They enable a *point-to-point* scenario.

32.4.2.1 Connecting to A Message Queue

The code for connecting to a message queue is shown below:

```

1 Context context = new InitialContext();
2
3 QueueConnectionFactory queueFactory = (QueueConnectionFactory) context.lookup("ConnectionFactory");
4

```

```

5 QueueSession queueSession = queueFactory.createQueueConnection().createQueueSession(false /*not transacted*/,
    Session.AUTO_ACKNOWLEDGE);
6
7 Queue queue = (Queue)context.lookup("queue/HiThere");

```

32.4.2.2 Developing Queue Senders

Message senders will use a block of code similar to that shown above to connect to a message queue. A `QueueSender` is then created via the `QueueSession`:

```

1 QueueSender sender = queueSession.createSender(queue);

```

Senders send messages to the queue by first creating and populating a message, and sending it to the appropriate queue. Below we create, populate and send a text message:

```

1 TextMessage message = queueSession.createTextMessage();
2 message.setText(messageField.getText());
3 sender.send(queue, message);

```

32.4.2.3 Creating Queue Receivers

Message receivers connect to a JMS implementation in the same way message publishers do. They then create a `QueueReceiver` via the `QueueSession` and attach a `MessageListener` to it. Message listeners are notified upon receipt of a new message:

```

1 QueueReceiver messageRecipient = queueSession.createReceiver(messageQueue);
2
3 messageRecipient.setMessageListener(new MessageListener()
4 {
5     public void onMessage(Message msg)
6     {
7         // process message
8     }
9 });

```

32.4.2.4 An Example Application using Point-To-Point Messaging

We list below an application with a client, the `MessageSender` sending messages via a message queue to a `MessageRecipient`. The latter pops the received messages up on the desktop.

When you run the application, note that you can launch the sender and send a few messages before launching a recipient. The recipient will receive the messages as it comes alive.

Alternatively, if we have multiple receivers active, any one of these will receive a particular message.

Note also that the code below is portable across different JMS-compliant MOM vendors.

```

1 package TestJMS;
2
3 import javax.jms.*;
4 import javax.naming.*;
5 import javax.swing.*;
6 import java.awt.event.*;
7
8 public class MessageSender extends JFrame
9 {
10     public static void main(String[] args)

```

```

11 {
12     new MessageSender().show();
13 }
14
15 public MessageSender()
16 {
17     createGUI();
18
19     addSubmitListener();
20
21     try
22     {
23         createMessageSender();
24     }
25     catch (Exception e)
26     {
27         e.printStackTrace();
28         System.exit(0);
29     }
30 }
31
32 private void createGUI()
33 {
34     setTitle("MessageSender");
35     JPanel messagePanel = new JPanel();
36     messagePanel.setLayout(new java.awt.GridLayout(2,1));
37     messagePanel.add(messageField);
38     messagePanel.add(sendButton);
39     getContentPane().add(messagePanel);
40     setDefaultCloseOperation(EXIT_ON_CLOSE);
41     pack();
42 }
43
44 private void addSubmitListener()
45 {
46     sendButton.addActionListener( new ActionListener()
47     {
48         public void actionPerformed(ActionEvent event)
49         {
50             try
51             {
52                 TextMessage message
53                     = queueSession.createTextMessage();
54                 message.setText(messageField.getText());
55                 messageSender.send(messageQueue, message);
56             }
57             catch (javax.jms.JMSException e)
58             {
59                 JOptionPane.showMessageDialog(MessageSender.this,
60                     e.getMessage(), "JMS Exception",
61                     JOptionPane.ERROR_MESSAGE);
62             }
63         }
64     });
65 }
66
67 private void createMessageSender() throws NamingException,
68                                     JMSException
69 {
70     Context context = new InitialContext();
71
72     QueueConnectionFactory queueFactory
73         = (QueueConnectionFactory)context.lookup
74             ("ConnectionFactory");
75
76     System.out.println("Connected to Queue connection factory.");
77
78     queueSession
79         = queueFactory.createQueueConnection().createQueueSession
80             (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);
81
82     messageQueue = (Queue)context.lookup("queue/HiThere");
83

```

```

84     messageSender
85         = queueSession.createSender(messageQueue);
86     }
87
88     private JButton sendButton = new JButton("send message");
89     private JTextField messageField = new JTextField("", 40);
90     private QueueSender messageSender;
91     private QueueSession queueSession;
92     private Queue messageQueue;
93 }

```

```

1  package TestJMS;
2
3  import javax.jms.*;
4  import javax.naming.*;
5  import javax.swing.*;
6  import java.awt.event.*;
7
8  public class MessageRecipient extends JFrame
9  {
10     public static void main(String[] args)
11     {
12         new MessageRecipient().show();
13     }
14
15     public MessageRecipient()
16     {
17         createGUI();
18
19         try
20         {
21             createMessageRecipient();
22         }
23         catch (Exception e)
24         {
25             e.printStackTrace();
26             System.exit(0);
27         }
28     }
29
30     private void createGUI()
31     {
32         setTitle("Gold Price Subscriber");
33         setDefaultCloseOperation(EXIT_ON_CLOSE);
34         setSize(new java.awt.Dimension(150,150));
35     }
36
37     private void createMessageRecipient()
38         throws NamingException, JMSException
39     {
40         Context context = new InitialContext();
41
42         QueueConnectionFactory queueFactory
43             = (QueueConnectionFactory)context.lookup
44                 ("ConnectionFactory");
45
46         System.out.println("Connected to queue connection factory.");
47
48         QueueConnection queueConnection
49             = queueFactory.createQueueConnection();
50
51         QueueSession queueSession
52             = queueConnection.createQueueSession
53                 (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);
54
55         Queue messageQueue = (Queue)context.lookup("queue/HiThere");
56
57         QueueReceiver messageRecipient

```

```

58     = queueSession.createReceiver(messageQueue);
59
60     messageRecipient.setMessageListener(new MessageListener()
61     {
62         public void onMessage(Message msg)
63         {
64             JOptionPane.showMessageDialog
65                 (MessageRecipient.this, msg, "Received message",
66                 JOptionPane.INFORMATION_MESSAGE);
67         }
68     });
69
70     queueConnection.start();
71 }
72 }

```

32.4.3 Using JMS Topics

Topics are usually used for distributing events or information. The enable a *publish-subscribe* scenario.

32.4.3.1 Connecting to a topic

The code for connecting to a topic is very similar to that used to connect to a queue:

```

1 Context context = new InitialContext();
2
3 TopicConnectionFactory topicFactory = (TopicConnectionFactory) context.lookup("ConnectionFactory");
4
5 TopicConnection topicConnection = topicFactory.createTopicConnection();
6
7 TopicSession topicSession = topicConnection.createTopicSession(false /*not transacted*/, Session.
    AUTO_ACKNOWLEDGE);
8
9 Topic topic = (Topic)context.lookup("topic/TopicName");

```

32.4.3.2 Developing Publishers for a Topic

Publishers connect to a topic, create and populate messages and publish them with the topic:

```

1 TopicPublisher publisher = topicSession.createPublisher(topic);
2
3 ...
4
5 TextMessage message = topicSession.createTextMessage();
6 message.setText("The text of the message.");
7 publisher.publish(topic, message);

```

32.4.3.3 Developing Subscribers for a Topic

Developing topic subscribers is, once again, very similar to developing queue receivers. They first connect to a topic in the same way publishers do and then they create a subscriber and set a message listener for the subscriber which processes the messages:

```

1 TopicSubscriber subscriber = topicSession.createSubscriber(topic);
2 subscriber.setMessageListener(new MessageListener()
3 {
4     public void onMessage(Message msg)
5     {
6         //process the message
7     }
8 });
9
10 topicConnection.start();

```

32.4.3.4 An Example Application using Publish-Subscribe Messaging

Working with topics is very similar to working with queues. The code is very close, yet the behaviour is different in some important ways:

- Multiple topic subscribers may receive the same message.
- Subscribers only receive messages which were sent while they were active.

Below we list a `GoldPricePublisher` which publishes gold price quotes to a `GoldPrice` topic and a `GoldPriceSubscriber` which receives messages published with the `GoldPrice` topic.

```

1 package TestJMS;
2
3 import javax.jms.*;
4 import javax.naming.*;
5 import javax.swing.*;
6 import java.awt.event.*;
7
8 public class GoldPricePublisher extends JFrame
9 {
10     public static void main(String[] args)
11     {
12         new GoldPricePublisher().show();
13     }
14
15     public GoldPricePublisher()
16     {
17         createGUI();
18
19         addSubmitListener();
20
21         try
22         {
23             createGoldPricePublisher();
24         }
25         catch (Exception e)
26         {
27             e.printStackTrace();
28             System.exit(0);
29         }
30     }
31
32     private void createGUI()
33     {
34         setTitle("Gold Price Publisher");
35         JPanel submitPanel = new JPanel();
36         submitPanel.add(submitButton);
37         submitPanel.add(goldPriceField);
38         getContentPane().add(submitPanel);
39         setDefaultCloseOperation(EXIT_ON_CLOSE);
40         pack();

```



```

41 }
42
43 private void addSubmitListener()
44 {
45     submitButton.addActionListener( new ActionListener()
46     {
47         public void actionPerformed(ActionEvent event)
48         {
49             try
50             {
51                 TextMessage message
52                     = topicSession.createTextMessage();
53                 message.setText(goldPriceField.getText());
54                 goldPricePublisher.publish(goldPriceTopic, message);
55             }
56             catch (javax.jms.JMSException e)
57             {
58                 JOptionPane.showMessageDialog(GoldPricePublisher.this,
59                     e.getMessage(), "JMS Exception",
60                     JOptionPane.ERROR_MESSAGE);
61             }
62         }
63     });
64 }
65
66 private void createGoldPricePublisher() throws NamingException,
67                                         JMSException
68 {
69     Context context = new InitialContext();
70
71     TopicConnectionFactory topicFactory
72         = (TopicConnectionFactory)context.lookup
73             ("ConnectionFactory");
74
75     System.out.println("Connected to topic connection factory.");
76
77     topicSession
78         = topicFactory.createTopicConnection().createTopicSession
79             (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);
80
81     goldPriceTopic = (Topic)context.lookup("topic/GoldPrice");
82
83     goldPricePublisher
84         = topicSession.createPublisher(goldPriceTopic);
85 }
86
87 private JButton submitButton = new JButton("submit");
88 private JTextField goldPriceField = new JTextField("", 8);
89 private TopicPublisher goldPricePublisher;
90 private TopicSession topicSession;
91 private Topic goldPriceTopic;
92 }

```

```

1 package TestJMS;
2
3 import javax.jms.*;
4 import javax.naming.*;
5 import javax.swing.*;
6 import java.awt.event.*;
7
8 public class GoldPriceSubscriber extends JFrame
9 {
10     public static void main(String[] args)
11     {
12         new GoldPriceSubscriber().show();
13     }
14
15     public GoldPriceSubscriber()

```

```

16  {
17      createGUI();
18
19      try
20      {
21          createGoldPriceSubscriber();
22      }
23      catch (Exception e)
24      {
25          e.printStackTrace();
26          System.exit(0);
27      }
28  }
29
30  private void createGUI()
31  {
32      setTitle("Gold Price Subscriber");
33      setDefaultCloseOperation(EXIT_ON_CLOSE);
34      setSize(new java.awt.Dimension(150,150));
35  }
36
37  private void createGoldPriceSubscriber()
38      throws NamingException, JMSException
39  {
40      Context context = new InitialContext();
41
42      TopicConnectionFactory topicFactory
43          = (TopicConnectionFactory)context.lookup
44              ("ConnectionFactory");
45
46      System.out.println("Connected to topic connection factory.");
47
48      TopicConnection topicConnection
49          = topicFactory.createTopicConnection();
50
51      TopicSession topicSession
52          = topicConnection.createTopicSession
53              (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);
54
55      Topic goldPriceTopic
56          = (Topic)context.lookup("topic/GoldPrice");
57
58      TopicSubscriber goldPriceSubscriber
59          = topicSession.createSubscriber(goldPriceTopic);
60
61      goldPriceSubscriber.setMessageListener(
62          new MessageListener()
63          {
64              public void onMessage(Message msg)
65              {
66                  JOptionPane.showMessageDialog
67                      (GoldPriceSubscriber.this, msg,
68                       "Received message",
69                       JOptionPane.INFORMATION_MESSAGE);
70              }
71          });
72
73      topicConnection.start();
74  }
75 }

```

32.5 The Structure of a JMS Message

Each message has the following elements:

- **Header:** The header contains pre-defined JMS properties providing generic description of messages. The header is discussed in more detail below.

- **User-Defined Properties:** These are name-value pairs for user-defined properties. Message consumers can select messages based on user-defined message properties.
- **Body:** The body contains the content of the message which depends on the type of message.

32.5.1 JMS Message Headers

JMS message headers contain JMS-defined properties. These message properties change as the message is in transit, i.e. the message publisher sets (implicitly or explicitly some of these) and some are set/modified by the message consumer.

Below is a list of the JMS pre-defined properties contained in the message header:

- **JMSDestination:** The queue or topic for which the message is meant. This is provided by the publisher upon sending the message.
- **JMSTimeStamp:** The time-instant at which the message was sent. The time-stamp is generated by the `send()` or `publish()` method.
- **JMSReplyTo:** A reply-to destination (queue or topic) set by the message publisher.
- **JMSDelivered:** A flag signaling whether a message has been delivered or not set by the JMS provider.
- **JMSCorrelationId:** The correlation id is used to link a reply with a request.
- **JMSPriority:** The priority assigned to the message.
- **JMSExpiration:** The expiry date/time of the message generated by `send()` or `publish()`.
- **JMSMessageID:** A unique identifier for the message generated by `send()` or `publish()`.
- **JMSDeliveryMode:** The delivery modes supported by the JMS API are PERSISTENT and NON_PERSISTENT
- **JMSType:** The type of the message – i.e. text, byte, stream, map or object message.
- **JMSRedelivered:** This property is set by the JMS provider and signals whether a message has been delivered more than once to a message consumer (due to not receiving an acknowledgement).

32.6 Durable Subscribers

If there is a single queue receiver, it receive all messages from the queue irrespective of whether the message was sent while they were active or not.

Normal topic subscribers, however, only receive messages which are sent while they are active. Durable subscribers subscribe to a topic and receive all messages sent from then onwards, even those which were sent while they were not active.

A durable subscriber to a topic are created in the following way:

```
1 TopicSubscriber subscriber = session.createDurableSubscriber(theTopic, "someSubscriberId");
```

32.7 Messages Participating in Transactions

The first argument supplied when we create a queue session specifies whether the message is a transacted message or not. Transacted messages create a transacted messaging session which can be committed or rolled back by calling `commit()` or `rollback()` on the session object.

32.8 Selected message retrieval

JMS supports selecting messages on either pre-defined or user-defined message properties. For example, to select only messages with a priority of 6 or higher one can write the following code:

```
1 String msgSelector = "JMSPriority > 5";  
2 QueueReceiver receiver = session.createReceiver(queue, selector);
```

The receiver will only receive messages with a priority exceeding 5.

Chapter 33

Singleton beans

A singleton is a class which has, at most, a single access. Singletons were not traditionally supported in EJB, but as of EJB 3.1 there is explicit support for singletons.

A enterprise bean which has been declared a singleton bean will be instantiated only once by the application server and the application server will make that instance generally available (e.g. through dependency injection or JNDI lookup).

The core benefit of singletons is that one is able to share state across an entire application and across users. The singleton is generally accessible and can be injected into other enterprise beans or into other managed components. **Note:** *Singletons should not be over-used. They are seldom the best solution for business information or services and generally more useful for infrastructural components and access to technical resources.*

33.1 Declaring singletons

To declare a singleton bean you annotate a normal POJO with the `@Singleton` annotation:

```
1 import javax.ejb.Singleton;
2
3 @Singleton
4 public class MyPropertiesBean
5 {
6     public String getProperty(String propertyName)
7     {
8         ...
9     }
10
11     private Properties properties;
12 }
```

33.2 Using singleton beans

Singleton beans can be accessed like session beans, i.e. they can be either looked up via JNDI or injected by the application server. Normally one would prefer injection:

```
1 @Stateless
2 public class MyServiceBean
3 {
4     public void someService()
```

```
5  {
6      myPropertiesBean.getProperty("msgs.helloWorld");
7      ...
8  }
9
10 @Inject
11 private MyPropertiesBean myPropertiesBean;
12 }
```

33.3 Application startup and shutdown callbacks

Because singletons usually provide access to application-wide resources, they are typically initialized during application startup. They can also be used to perform some other application startup initialization.

By default the EJB container will decide when to initialize the singleton instance. This could happen the first time the singleton is accessed. However, one can request that the singleton should be instantiated upon application startup using the `@Startup` annotation.

The standard bean life cycle interceptors `@PostConstruct` and `@PreDestroy` can be used to specify tasks which should be executed upon bean initialization and bean destruction. The bean destruction is done when the application is undeployed. Using the `@Startup` annotation one requests the bean instantiation to be done on application startup and in that case the tasks annotated with `PreConstruct` will also be executed upon application startup.

This is done as follows:

```
1  @Singleton
2  @Startup
3  public class PropertiesBean
4  {
5      @PostConstruct
6      private void startup() { ... }
7
8      @PreDestroy
9      private void shutdown() { ... }
10     ...
11 }
```

Chapter 34

The Java-EE timer service

Most session beans offer services that are either invoked by clients (in the case of stateless and stateful session beans), or by the delivery of JMS messages (e.g. message-driven beans). However, certain workflow steps may need to be performed based on scheduled, timed events (i.e. without direct user interaction). This could be at a particular date-time or repetitively.

Historically, this was accomplished by using an external timer service (such as `cron`) to invoke a program that connects to and invokes a service from an EJB. This mechanism, however, is laborious, and difficult to integrate with a Java EE solution.

As of EJB 2.1, JavaEE supported a timer service. However, the timer service was not very mature compared to scheduling frameworks like Unix `cron`, Quartz, ... As of EJB 3.1 the EJB timer service is a lot more feature rich and flexible. It is largely inspired by `cron`.

Scheduled timers are *reliable* and *robust* - they must survive application server crashes or machine restarts.

34.1 What can the Java-EE timer service be used for?

The timer service support calendar based scheduling to schedule tasks to occur either once or repetitively

- at specific particular date-times,
- in specific intervals, or
- at durations after the previous task has been completed.

The EJB timer service primarily exists to support long-running business processes. Since session beans conceptually only last for the duration of a session, the timer service extends the usefulness of EJBs in situations such as the following:

- **Time-based workflow steps:** For example, imposing a fine when a rented item (library book, video, car) is not returned on time, or prompting a client to renew their support contract a month before it expires.
- **Monitoring / Polling:** Periodically making sure that certain services, resources or devices are available and responding.
- **Periodical reminders:** Sending a monthly reminder to users of a system to remind them to update their password for security purposes.

Timers may be used in both **Stateless Session Beans** as well as *Message-Driven Beans* .

34.2 Architecture of timer service

The timer service is a container service which enables one to register session bean services to be called on timer events, i.e. one can

1. either automatically (via annotations) or programmatically registers a stateless session bean service with a timer
2. after which the container will call the registered services on the corresponding timer events.

Timer services are durable since the schedule is persisted in a database.

34.3 Specifying the schedule

The Java-EE timer service uses a schedule specification approach which is analogous to the Unix **cron** syntax for scheduling jobs. One specifies the second, minute, hour, dayOfMonth, month, dayOfWeek and year and can specify relative to which time zone the time instant specification is done.

34.3.1 Specifying schedule expressions

For each of schedule instant parameters one may specify

- a single value, e.g. like `month="7"`,
- a wild card to specify any, e.g. `dayOfWeek="*"`
- a comma-delimited list, e.g. `dayOfMonth="15,Last"`
- an increment specification specifying an initial value and an increment thereafter which is separated via a forward slash, e.g.
 - `minute="20/5"`
 - * after 20 minutes and every 5 minutes thereafter.
 - `hour="*/3"`
 - * start any hour, but then perform the task every 3 hours thereafter.

34.4 Automatic timer creation

If a session bean has services which are annotated with the **@Schedule** annotation, timer services for those services will automatically be created. The annotated services will thus be automatically scheduled as per annotation.

For example, if you want to have a weekly sales report emailed every Friday afternoon at 16h30 could annotate the respective service as follows:


```

1 @Stateless
2 public class ReportingBean
3 {
4     @Schedule(minute="30", hour="16", dayOfWeek="Fri")
5     public void emailWeeklySalesReport()
6     { ... }
7 }

```

You could also specify multiple scheduling requests as follows:

```

1 @Stateless
2 public class ReportingBean
3 {
4     @Schedules({
5         @Schedule(minute="30", hour="16", dayOfWeek="Fri")
6         @Schedule(minute="30", hour="16", dayOfMonth="Last")
7     })
8     public void emailWeeklySalesReport()
9     { ... }
10 }

```

which requests the report to be emailed every Friday as well as on the last day of the month.

34.5 Programmatic timer creation

At times you want to schedule tasks in response to certain business events. To this end the EJB specification supports the programmatic timer creation. This is done by asking a timer service provided by the EJB container to create an appropriate timer for a given schedule expression. Upon timeout event of the created timer, the application server will call the `@Timeout` annotated service of that stateless session bean from which the timer was created.

For example, in order to ensure that an invoice is paid on its due date, one could schedule, for each received invoice, an appropriate timer programmatically.

```

1 @Stateless
2 public class InvoiceProcessorBean
3 {
4     public void processInvoice(Invoice invoice)
5     {
6         entityManager.persist(invoice);
7
8         Calendar dueDate = invoice.getDueDate();
9         int dayOfMonth = dueDate.get(Calendar.DAY_OF_MONTH);
10        int month = dueDate.get(Calendar.MONTH);
11        int year = dueDate.get(Calendar.YEAR);
12
13        ScheduleExpression payDateExpr = new ScheduleExpression().dayOfMonth(dayOfMonth).month(month).year(year)
14        ;
15
16        // Now ask timer service to create persistent timer (true parameter)
17        timerService.createCalendarTimer(payDateExpr, new TimerConfig(invoice, true);
18    }
19
20    @Timeout
21    public void payInvoice(Timer timer)
22    {
23        Invoice invoice = (Invoice) timer.getInfo();
24        paymentProcessor.payInvoice(invoice);
25    }
26
27    @EJB
28    PaymentProcessor paymentProcessor;
29
30    @PersistenceContext
31    private EntityManager entityManager;
32
33 }

```

```
32  @Resource  
33  private TimerService timerService;  
34  }
```

Chapter 35

Transactions

35.1 The Java Transaction API (JTA)

35.1.1 Overview of JTA

35.1.1.1 JTS and JTA

Java Transaction Service (JTS) is Java's implementation of CORBA's Object Transaction Service (OTS). OTS specifies support for enlisting multiple resources and two-phase commit. JTS hence supports distributed transactions across multiple X/Open XA resources.

The Java Transaction API (JTA) is a standardized API for interfacing with a JTS.

35.1.1.2 Transaction managers

In the case of programmatic transaction demarcation (i.e. bean managed transactions), one interacts directly with a transaction manager. **TransactionManager** offers the following services:

- **void begin():** Create a new transaction and associate it with the current thread.
- **void commit():** Complete the transaction associated with the current thread, finalising all changes made on the different resources enlisted in the transaction.
- **int getStatus():** Obtain the status of the transaction associated with the current thread.
- **Transaction getTransaction():** Get the transaction object that represents the transaction context of the calling thread.
- **void resume(Transaction tobj):** Resume the transaction context association of the calling thread with the transaction represented by the supplied Transaction object.
- **void rollback():** Roll back the transaction associated with the current thread.
- **void setRollbackOnly():** Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.
- **void setTransactionTimeout(int seconds):** Modify the value of the timeout value that is associated with the transactions started by the current thread with the begin method.
- **Transaction suspend():** Suspend the transaction currently associated with the calling thread and return a Transaction object that represents the transaction context being suspended.

35.1.1.3 JTA transactions

35.1.1.3.1 JTA transaction API One can also interface with the transaction itself via

- **boolean enlistResource(XAResource xaRes):** Enlist the resource specified with the current transaction context of the calling thread.
- **boolean delistResource(XAResource xaRes, int flag):** Delist the resource specified from the current transaction associated with the calling thread.
- **int getStatus():** Obtain the status of the transaction associated with the current thread.
- **void registerSynchronization(Synchronization sync):** Register a synchronization object for the transaction currently associated with the calling thread.
- **void commit():** Complete the transaction represented by this `Transaction` object.
- **void rollback():** Rollback the transaction represented by this `Transaction` object.
- **void setRollbackOnly():** Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

35.1.1.3.2 The status of a transaction The status of a transaction can be one of the following:

- **Status.STATUS_ACTIVE:** A transaction is associated with the target object and it is in the active state.
- **Status.STATUS_COMMITTED:** A transaction is associated with the target object and it has been committed.
- **Status.STATUS_COMMITTING:** A transaction is associated with the target object and it is in the process of committing.
- **Status.STATUS_MARKED_ROLLBACK:** A transaction is associated with the target object and it has been marked for rollback, perhaps as a result of a `setRollbackOnly` operation.
- **Status.STATUS_NO_TRANSACTION:** No transaction is currently associated with the target object.
- **Status.STATUS_PREPARED:** A transaction is associated with the target object and it has been prepared.
- **Status.STATUS_PREPARING:** A transaction is associated with the target object and it is in the process of preparing.
- **Status.STATUS_ROLLEDBACK:** A transaction is associated with the target object and the outcome has been determined as rollback.
- **Status.STATUS_ROLLING_BACK:** A transaction is associated with the target object and it is in the process of rolling back.
- **Status.STATUS_UNKNOWN:** A transaction is associated with the target object but its current status cannot be determined.

Multiple resources (e.g. databases, message queues, email servers, ...) may be enlisted within a transaction. The state of the system may become inconsistent due to having committed the transaction on some resources, but failing to commit the same transaction on other resources (perhaps to connectivity problems).

In *two-phase commit* each resource is asked to prepare to commit. In the second round the commit is confirmed with each resource. If a resource goes/is down during commit preparation, the transaction is rolled back. If a resource goes down/is on transaction commitment, the resource will read the ready-to-commit message either when it comes up again or on time-out and will auto-commit.

35.1.2 Examples of using JTA directly

35.1.2.1 Implementing an XID class

Before using JTA, you must first implement an X/Open transaction identifier (Xid) class for identifying transactions (this would normally be done by the transaction manager). The Xid contains three elements: formatID, gtrid (global transaction ID), and bqual (branch qualifier ID).

The formatID is usually zero, meaning that you are using the OSI CCR (Open Systems Interconnection Commitment, Concurrency, and Recovery standard) for naming. If you are using another format, the formatID should be greater than zero. A value of -1 means that the Xid is null.

The gtrid and bqual can each contain up to 64 bytes of binary code to identify the global transaction and the branch transaction, respectively. The only requirement is that the gtrid and bqual taken together must be globally unique. Again, this can be achieved by using the naming rules specified in the OSI CCR.

The following example illustrates implementation of an Xid:

```

1 import javax.transaction.xa.*;
2
3 public class MyXid implements Xid
4 {
5     protected int formatId;
6     protected byte gtrid[];
7     protected byte bqual[];
8
9     public MyXid(int formatId, byte gtrid[], byte bqual[])
10    {
11        this.formatId = formatId;
12        this.gtrid = gtrid;
13        this.bqual = bqual;
14    }
15
16    public int getFormatId() {return formatId;}
17    public byte[] getBranchQualifier() {return bqual;}
18    public byte[] getGlobalTransactionId() {return gtrid;}
19 }

```

35.1.2.2 Creating an XADataSource

You need to create an XA-DataSource for the resource (e.g. database) you are using:

```

1 public DataSource getDataSource() throws SQLException
2 {
3     SQLServerDataSource xaDS = new
4         com.merant.datadirect.jdbcx.sqlserver.SQLServerDataSource();

```

```

5   xaDS.setDataSourceName("SQLServer");
6   xaDS.setServerName("server");
7   xaDS.setPortNumber(1433);
8   xaDS.setSelectMethod("cursor");
9   return xaDS;
10 }

```

35.1.2.3 Two-phase commit

```

1  XADataSource xaDS = getDataSource();
2  XAConnection xaCon = xaDS.getXAConnection("jdbc_user", "jdbc_password");
3  XAResource xaRes = xaCon.getXAResource();
4  Connection con = xaCon.getConnection();
5  Statement stmt = con.createStatement();
6
7  Xid xid = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});
8
9  try {
10     xaRes.start(xid, XAResource.TMNOFLAGS);
11     stmt.executeUpdate("insert into test_table values (100)");
12     xaRes.end(xid, XAResource.TMSUCCESS);
13
14     int ret = xaRes.prepare(xid);
15     if (ret == XAResource.XA_OK)
16         xaRes.commit(xid, false);
17 }
18 catch (XAException e) { e.printStackTrace(); }
19 finally { stmt.close(); con.close(); xaCon.close(); }

```

35.1.2.4 Rollking back a transaction

```

1  xaRes.start(xid, XAResource.TMNOFLAGS);
2  stmt.executeUpdate("insert into test_table values (100)");
3  xaRes.end(xid, XAResource.TMSUCCESS);
4
5  ret = xaRes.prepare(xid);
6  if (ret == XAResource.XA_OK)
7      xaRes.rollback(xid);

```

35.1.2.5 Suspending and resuming a transaction

This example shows how a distributed transaction branch suspends, lets the same connection do a local transaction, and then resumes the branch later. The two-phase commit actions of distributed transaction do not affect the local transaction.

```

1  xid = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});
2
3  xaRes.start(xid, XAResource.TMNOFLAGS);
4  stmt.executeUpdate("insert into test_table values (100)");
5  xaRes.end(xid, XAResource.TMSUSPEND);
6
7  // This update is done outside of transaction scope, so it
8  // is not affected by the XA rollback.
9  stmt.executeUpdate("insert into test_table2 values (111)");
10
11 xaRes.start(xid, XAResource.TMRESUME);
12 stmt.executeUpdate("insert into test_table values (200)");
13 xaRes.end(xid, XAResource.TMSUCCESS);
14
15 ret = xaRes.prepare(xid);
16 if (ret == XAResource.XA_OK)
17     xaRes.rollback(xid);

```

35.1.2.6 Sharing an XAResource across transactions

This example illustrates how one XA resource can be shared among different transactions.

Two transaction branches are created, but they do not belong to the same distributed transaction. JTA allows the XA resource to do a two-phase commit on the first branch even though the resource is still associated with the second branch.

```

1  xid1 = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});
2  xid2 = new MyXid(100, new byte[]{0x11}, new byte[]{0x22});
3
4  xaRes.start(xid1, XAResource.TMNOFLAGS);
5  stmt.executeUpdate("insert into test.table1 values (100)");
6  xaRes.end(xid1, XAResource.TMSUCCESS);
7  xaRes.start(xid2, XAResource.TMNOFLAGS);
8
9  // Should allow XA resource to do two-phase commit on
10 // transaction 1 while associated to transaction 2
11 ret = xaRes.prepare(xid1);
12 if (ret == XAResource.XA_OK)
13     xaRes.commit(xid1, false);
14
15 stmt.executeUpdate("insert into test.table2 values (200)");
16 xaRes.end(xid2, XAResource.TMSUCCESS);
17
18 ret = xaRes.prepare(xid2);
19 if (ret == XAResource.XA_OK)
20     xaRes.rollback(xid2);

```

35.1.2.7 Recovering from a failed transaction

```

1  MyXid[] xids;
2
3  xids = xaRes.recover(XAResource.TMSTARTRSCAN | XAResource.TMENDRSCAN);
4  for (int i=0; xids!=null && i<xids.length; i++) {
5      try {
6          xaRes.rollback(xids[i]);
7      }
8      catch (XAException ex) {
9          try {
10             xaRes.forget(xids[i]);
11         }
12         catch (XAException ex1) {
13             System.out.println("rollback/forget failed: " + ex1.errorCode);
14         }
15     }
16 }

```

35.2 Declarative transaction demarcation

Specifying transaction boundaries in code hard-codes transaction boundaries, limits reuse and pollutes the business logic with transaction specifications.

Instead it is generally recommended to declare your transaction requirements more abstractly. This facilitates reuse of a service in different processes which may require requiring different transaction boundaries.

Declarative transaction demarcation is done by specifying the transactional requirements through a transaction attribute, either via annotations or in a deployment descriptor.

35.2.1 Transaction attributes

Services offered by enterprise beans must have a transaction attribute which specifies the type of transaction support required by the service (or all services of the enterprise bean). The transactions attributes supported by the EJB specification are:

- `@NotSupported`
- `@Required`
- `@Supports`
- `@Mandatory`
- `@RequiresNew`
- `@Never`
- `@BeanManaged`

The transaction attribute is specified via annotations or in in the `ejb-jar.xml` deployment descriptor. In either case, a transaction attribute can be applied to either a specific method of a session bean or to all methods of a bean. Method-specific transaction requirement specifications override class wide transaction requirement specifications.

35.2.1.1 NotSupported

A bean defined with the `NotSupported` transactional attribute is not allowed to partake in any transaction. Invoking a method on a bean with this attribute has the consequence that the EJB container suspends any transaction until the method has been completed.

35.2.1.2 Required

This attribute guarantees that all bean services are performed always within a transaction context. If the calling client or bean is within the scope of a transaction the requested service will be included within that transaction scope. Otherwise a new transaction is created for the service request.

In the case where a new transaction is created for a service request the transaction will be committed upon successful completion and will be rolled back if an exception is thrown and not handled within the context of the service.

35.2.1.3 Supports

This attribute specifies that the bean will be included in the transaction scope if it is called from a transaction scope, but it will not create a new transaction scope if it is not called from a transaction scope.

35.2.1.4 Mandatory

This attribute specifies that the bean services must always be called from within a transaction scope. A

will be thrown if it is called from a client who is not operating within the context of a transaction.

35.2.1.5 RequiresNew

This attribute specifies that if a bean service is called from within a transaction scope, that a new transaction is created. The current transaction is suspended until the new transaction has been committed.

If a bean service is called outside any transaction scope it simply creates a new transaction scope for that service request.

35.2.1.6 Never

This attribute specifies that the bean services may not be called from any client operating within the scope of a transaction. A `java.rmi.RemoteException` is thrown if a bean service is called from a transaction scope.

35.2.1.7 BeanManaged

From EJB onwards only session beans are allowed to manage their own transactions. In this case the bean developer controls transactions via `begin()`, `abort()`, `commit()` and `rollback()` messages sent to the Java Transaction API (JTA).

35.2.2 Selecting a transaction attribute

If you need transactional control

- use **Required**
 - Still reusable in higher level services which require wider transaction scope.
- If what you do can be undone, but do not need transaction
 - use **Supports**
- If you cannot undo your work
 - use **Never**

35.2.3 Annotating Session Beans with transaction attributes

Transaction attributes are specified by annotating bean methods or the entire class with a `@TransactionAttribute` annotation:

```
1 @Stateless
2 public class OrderProcessorBean
3 {
4     @TransactionAttribute {TransactionAttributeType.REQUIRED}
5     public ProcessOrderResult processOrder(ProcessOrderRequest request)
6     {
7         ...
8     }
9 }
```

35.2.4 Transaction boundaries on method boundaries

Using container managed transactions requires that transaction boundaries all on method boundaries. This is usually not a problem and if that is not the case, it usually means that responsibilities are not sufficiently localized and that the design could benefit from refactoring. If one still requires finer grained transaction boundaries, one needs to go to container managed transactions.

35.3 Potential problems caused by concurrent access

35.3.1 Introduction

Not using transactions or relaxing transaction isolation levels may result in a number of issues.

35.3.2 Update loss

Not using any resource locking may result in update losses and hence in data corruption.

For example, a debit thread may read the account balance and calculate the new balance. Prior to updating the balance, a credit thread reads the same balance and calculates and modifies the balance. The first thread then continues and modifies balance. This would result in the credit being lost and in the balance of the account being corrupted.

35.3.3 Dirty Reads

Dirty reads is the activity of reading data which may never be valid. This is data which is deleted due to transaction roll-back and is never committed.

For example,

1. Transfer service updates balance of source account
2. In other transaction, debit service is refused because insufficient balance, providing balance.
3. Transfer service continues and tries to credit destination account, fails and is rolled back.

Provided balance was never correct. *Dirty Reads* may thus lead to data inconsistencies.

35.3.4 Non-repeatable reads

Non-repeatable reads occurs when reading the same data multiple times within a transaction, yet receiving different results. All results are valid since they are all committed data.

35.3.5 Phantom reads

A phantom read data which was not yet present (did not exist) when your transaction started, but has been added while your transaction is busy processing.

35.4 Transaction isolation levels

Not using transactions may result in update loss and data corruption. In principle, when using transactions, any resource which is accessed by code executing under the control of one transaction should be accessible by any any code executing under the control of another transaction prior to the first transaction being either committed or rolled back. This approach prevents resources from being corrupted due to concurrent access.

However, such stringent transaction isolation may result in performance and scalability problems. It is thus common to relax transaction isolation somewhat in order to improve scalability and performance.

35.4.1 ReadUncommitted

ReadUncommitted is the weakest transaction isolation level. At this level it is possible to read data which is not committed and which may never be committed, i.e. data which was never “true”. This phenomenon is called *Dirty Reads* and may result in data inconsistencies.

For example, a **transfer** service may update the balance of the source account. Prior to that transaction being committed, another transaction may get the balance and refuse the transaction due to insufficient funds (reporting the balance). The transfer service may not be able to perform the credit leg of the transaction and may be rolled back. The provided balance may never be reflected on a statement.

This isolation level provides the highest level of performance and scalability. It is, however, generally only used for applications which read data which is not modified.

35.4.2 ReadCommitted

This isolation level prevents your transaction from reading uncommitted data. However, both phantom reads and uncommitted reads can occur.

This is the default isolation level for most databases

35.4.3 Repeatable reads

Repeatable Reads ensures that when reading the state of a resource (e.g. database record) within the a transaction, one always obtains the same result. *Phantom Reads* may still occur, i.e. one may read data which did not exist at start of transaction.

35.4.4 Serializability

This is the most stringent isolation level which enforces that your transaction requires exclusive read and write privileges to data. Transactions can neither read nor write the same data your transaction accesses until your transaction has committed.

35.4.5 Setting the transaction isolation level

The EJB, JPA and JTA specifications don't specify a standard API for managing the isolation level of a resource, i.e. the degree to which the access to the resource by one transaction is isolated from other transactions. The setting of the isolation level will thus be resource manager specific. The isolation level is usually set for the JDBC connection pool.

Chapter 36

Security

36.1 Declarative authorization

When using declarative authorization, the role requirements for a service (or all services of a bean) are annotated, i.e. one specifies which security roles an authenticated principles requires in order to be able to access certain services.

36.1.1 Specifying authorization requirements

36.1.1.1 Specifying the required security roles for a service

To specify that a particular service should be available only to authenticated principals who have one of a number of security roles assigned to them, one annotated the respective service with a list of roles:

```
1 @Stateless
2 public class FinancialReportsGenerator
3 {
4     @RolesAllowed({"accountant", "shareholder", "financialManager"})
5     public IncomeStatement generateIncomeStatement(IncomeStatementRequest request)
6     { ... }
7 }
```

36.1.1.2 Default authorization requirements

By annotating a bean with a `@RolesAllowed` annotation, one specifies the default role requirements for services offered by that bean.

```
1 @Stateless
2 @RolesAllowed({"accountant", "financialManager", "shareholder"})
3 public class FinancialReportsGenerator
4 {
5     public IncomeStatement generateIncomeStatement(IncomeStatementRequest request)
6     { ... }
7
8     public BalanceSheet generateBalanceSheet(BalanceSheetRequest request)
9     { ... }
10
11     @RolesAllowed("financialManager")
12     public void signOffIncomeStatement(IncomestatementSignRequest request)
13     { ... }
14 }
```

36.1.2 Run-as

In the context of assembling higher level services from lower level services one encounters, at times, a situation where a user has the required security roles for the higher level service, but not for one of the lower level services called from the higher level services. In such cases one can temporarily assign the required security role for the lower level service to users who make use of the higher level service by annotating the bean offering the higher level service with a `@RunAs` annotation.

For example,

```

1 @Stateless
2 @RolesAllowed({"client", "salesRep"})
3 @RunAs("stockManagement")
4 public class OrderProcessorBean
5 {
6     public OrderResult processOrder(Order order)
7     { ...
8         // The following service requires the stockManagement role:
9         inventory.releaseStock(stockReleaseRequest);
10        ... }
11    ...
12 }
```

ensures that the `releaseStock` service is accessible from the `processOrder` service even though users of the `processOrder` service may not have the required security role for the `releaseStock` service by temporarily allocating the `stockManagement` role to the context of the services of the `OrderProcessorBean`.

All referenced roles are automatically declared for the application server. The `@DeclareRoles` annotation enables one to declare some additional roles which are not used within the bean itself, but which should be made available to the application server. The `@DeclareRoles` annotation takes a list of roles, just like the `@RolesAllowed` annotation.

36.2 Programmatic security

For many cases declarative security is sufficient. However, for more complex security rules such as only permitting a particular role to authorize purchases up to some amount, one may need to use programmatic authorization.

To this end the EJB session context provides two services

- `boolean sessionContext.isCallerInRole(String roleName)`: which returns true if the user of the service has been assigned the specified security role, and
- `java.security.Principal sessionContext.getCallerPrincipal()`: which returns the caller principal which enables one to
 - retrieve the name of the caller principal, and
 - verify whether the principal is equal to another principal.

For example

```

1 @Stateless
2 public class ProcurementBean
3 {
4     public AuthorizeProcurementResult authorizeProcurement(AuthorizeProcurementRequest request)
5     {
6         if ((request.getTotal() > departmentalLimit)
```

```
7         && (!sessionContext.isCallerInRole("organizationLevelAuthorizer"))
8             throw new SecurityException("RequireOrganizationLevelAuthorization
9
10        ...
11    }
12
13    @Resource
14    private SessionContext sessionContext;
15 }
```

Note: *The session context is injected by the application server.*

```
1 if (sessionContext.getCallerprincipal().getName().equals("ThaboKhumalo"))
2     ...
```


Part VI

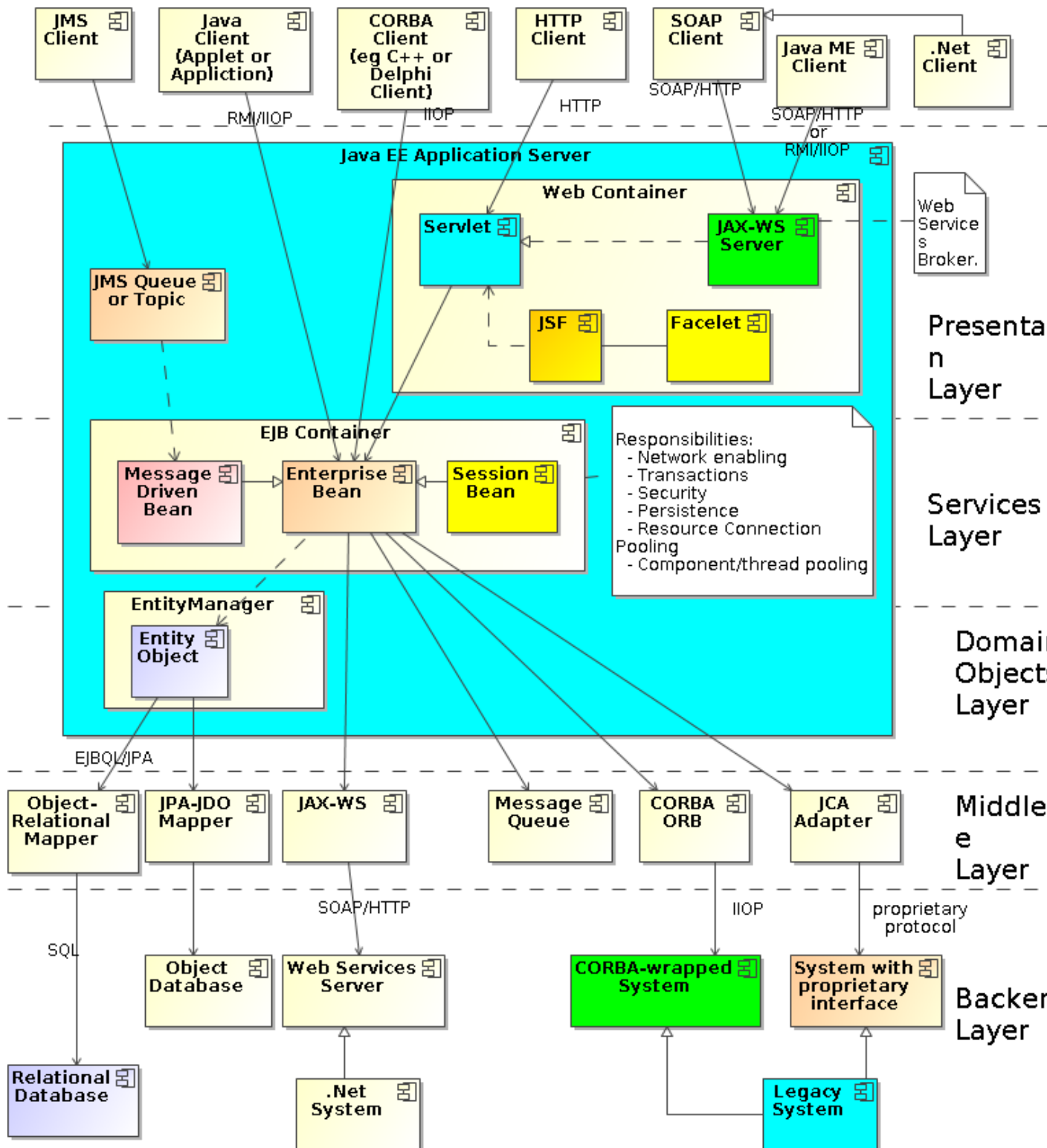
JSF

Chapter 37

Overview of JSF and Facelets

JSF/Facelets are the new Java web technology which is intended to fully replace JSPs/Servlets for web front end development, i.e. for developing human web adapters. Nevertheless servlets will continue to provide the base HTTP request processing components, though they will largely be confined to a low-level component used within the infrastructure.

37.1 Java-EE Architecture



37.2 What do JSF/Facelets provide?

- **Components for view state management and business logic binding:** JSF provides managed or backing beans which
 - maintain state variables for the view state, i.e. the state of the presentation layer process
 - provide binding to the business logic/services layer with dependency injection of Java EE session beans and web services.
- **Component model:** The JSF component model is similar to a thick client component model like AWT/Swing. It enables one to use and develop libraries of reusable presentation layer components which support *event listening* (e.g. change event listening) and the construction of composite components.
- **Converters:** JSF supports pluggable custom and user defined converters. Converters provide bi-directional conversion between the strings required in the user interface components and the actual data types used in the objects through which the state of the view is managed (e.g. the conversion between a string entered into a text field and the underlying data object).
- **Validators:** JSF supports pluggable custom and user defined validators. Validators validate that the user input satisfies certain constraints (i.e. that a date is in the past or that a entered value is within some range).
- **Messaging:** JSF supports a messaging infrastructure which enables one to submit page wide or component-specific messages to a message queue. These messages are ultimately rendered on a page.
- **View templates:** JSF supports the specification of page templates and even recursive templates with different components being inserted into the different regions of a template.
- **Bookmarkable pages:** In the context of dynamic page construction, bookmarking has always been a challenge. JSF provides explicit support for having bookmarkable pages which are still generated through the JSF engine.
- **Pluggable render kits:** JSF supports pluggable render kits to allow for rendering across devices as well as customized rendering.
- **Ajax support:** JSF-2 provides support for lightweight dynamic components via Ajax.

Chapter 38

Understanding JSF

38.1 Introduction

In this section we focus on

- introducing the *core concepts of JSF*,
- providing an understanding of the *JSF component tree*,
- explaining the *JSF request life cycle*,
- showing the *different navigation mechanisms* within a JSF application, and
- explaining *JSF event processing*.

38.2 Core JSF concepts

This section contains a glossary of core JSF concepts. It introduces some of the core terminology required to understand JSF.

38.2.1 Facelets

Facelets are XHTML documents which either encode an entire page/view (or content of template region) or a reusable component of a page. They are populated with XHTML and facelet tags.

38.2.2 JSF UI components (controls)

JSF controls is a reusable UI component. JSF provides a set of standard controls including buttons, text fields, drop down lists, multiple-selection lists, text areas, check boxes, tables, graphics and so on. JSF also supports container components like panel grids as well as the construction of custom components. There are large JSF component libraries which provide a vast selection of powerful components which are typically AJAX enabled.

38.2.3 JSF component tree

The JSF component tree is built up by the JSF engine from the page specification. The root of the tree will always be `UIViewRoot`. Typically it will contain a HTML form as first child node which, in turn, contains the various JSF components contained on the page.

In JSF processes like rendering, validation, conversion, data extraction, from components in order to populate bounded data fields, . . . , the component tree is traversed starting at the root element.

38.2.4 JSF renderers

Components can render themselves in JSF. However, using separate renderers results in improved responsibility localization, and flexibility.

In JSF one can register different component renderers for the same component. These would typically be used to render on different devices.

When defining a custom component one typically has to also define one or more renderers which are used to both, render the custom component and to extract information (e.g. user input) from the custom component.

38.2.5 Converters

Converters are required to convert model data types to a data type which can be rendered by the browser (e.g. text) and demarshall user input in order to populate the model objects. Converters are thus bi-directional.

38.2.6 JSF validators

For many components and scenarios the user input needs to be constrained in some way. It could be as simple as that a particular field may not be empty or that the input is constrained to 6 digits. Validators validate that the assigned constraints are met before the model is updated, thus preventing that the model ends up in an inconsistent state.

38.2.7 Backing beans and binding components (managed beans)

The backing beans are simple Java classes which comply to the Java Beans specification and are annotated to be managed beans. The role of JSF backing beans is to

- maintain the state of the presentation layer,
- bind to the business logic/services layer by making, for example, requests to stateless EJB session beans or to web services,
- to manage presentation layer processes.

We will distinguish between

- *backing beans* which solely maintain state and
- *binding components* which control user work flow and bind to the services/business logic layer.

Note: The dependency to services layer components, infrastructural components and presentation layer components are usually fulfilled by the container via *dependency injection*.

38.2.8 The Unified Expression Language

The Unified Expression Language is used to specify the binding between the UIComponent and the binding bean (also known as managed or backing bean). It provides an intuitive way for users to access the binding bean and via the binding bean's setters and getters the model objects.

38.2.9 JSF events

One of the strengths of JSF is that one can register event listeners with components. These could be triggered if the value of a component changes or if an action event is issued for a component. Besides the standard events issued by the standard JSF components, JSF also allows for custom components to define their custom events.

38.2.10 Navigation in JSF

Navigation can be specified either as

- navigation rules specified in the `facesConfig.xml`,
- or via the return values of action methods of the binding bean which manage the state of the presentation layer, or
- direct action bindings on UI components.

This makes managing both, flow specific and global navigation decisions very simple and flexible.

38.2.11 Messaging

JSF provides a mechanism to pipe messages generated during the request processing life cycle into a message queue which is ultimately rendered during the rendering phase. Messages can be process-specific and apply to an entire view or component-specific (e.g. a message notifying an invalid input on a particular component).

38.3 Minimalist example

The aim of the minimalist example is to show the common elements of JSF and how they fit into one another.

This minimalist example captures some person details and upon the save event populates a model object and renders the entered data on another page.

The simple example demonstrates

- how to build a simple POM through which the application can be built and executed,
- the definition of facelets for the user interface view,
- the definition of a binding bean which maintains the presentation layer state and could potentially call services on the business logic/services layer.

38.3.1 The Project Object Model (POM)

We create a default `pom.xml` using

```
1 mvn archetype:generate
```

and edit it specifying

- project metadata,
- project specific dependencies, build, deploy and distribution customizations,
- specify a generic POM which we use across many examples as parent POM.

The generic POM

- adds a dependency on the JSF API,
- in order to be able to execute within an embedded Jetty session, a dependency on a JSF implementation,
- and configure the `Jetty` maven plugin to use our application artifact.

This will enable us to not only build our application, but also to execute it within an embedded `Jetty` web container.

38.3.1.1 `pom.xml`

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4
5   <groupId>za.co.solms.training.jsf</groupId>
6   <artifactId>basicTextForm</artifactId>
7   <packaging>war</packaging>
8   <version>1.0-SNAPSHOT</version>
9
10  <name>basicTextForm</name>
11  <url>http://www.solms.co.za</url>
12
13  <parent>
14    <groupId>za.co.solms.training.jsf</groupId>
15    <artifactId>examples</artifactId>
16    <version>1.0-SNAPSHOT</version>
17    <relativePath>../pom.xml</relativePath>
18  </parent>
19 </project>
```

38.3.1.2 The parent `pom.xml`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
  v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4   <groupId>za.co.solms.training.jsf</groupId>
5   <artifactId>examples</artifactId>
6   <version>1.0-SNAPSHOT</version>
7   <packaging>pom</packaging>
8
```

```

9    <repositories>
10      <repository>
11        <id>java.net</id>
12        <name>Java.Net repository</name>
13        <url>http://download.java.net/maven/2/</url>
14      </repository>
15      <repository>
16        <id>maven repo</id>
17        <url>http://repo1.maven.org/maven2/</url>
18      </repository>
19    </repositories>
20
21    <dependencies>
22      <dependency>
23        <groupId>com.sun.faces</groupId>
24        <artifactId>jsf-api</artifactId>
25        <version>2.2.2</version>
26        <scope>compile</scope>
27      </dependency>
28
29      <dependency>
30        <groupId>com.sun.faces</groupId>
31        <artifactId>jsf-impl</artifactId>
32        <version>2.2.2</version>
33        <scope>compile</scope>
34      </dependency>
35
36      <dependency>
37        <groupId>junit</groupId>
38        <artifactId>junit</artifactId>
39        <version>4.11</version>
40        <scope>test</scope>
41      </dependency>
42    </dependencies>
43
44    <build>
45      <finalName>/${project.artifactId}</finalName>
46      <defaultGoal>install</defaultGoal>
47      <plugins>
48        <plugin>
49          <groupId>org.eclipse.jetty</groupId>
50          <artifactId>jetty-maven-plugin</artifactId>
51          <version>9.2.0.M0</version>
52          <configuration>
53            <webAppConfig>
54              <contextPath>/${project.artifactId}</contextPath>
55            </webAppConfig>
56          </configuration>
57        </plugin>
58      </plugins>
59
60      <pluginManagement>
61        <plugins>
62          <plugin> <!-- Configure Java compiler -->
63            <groupId>org.apache.maven.plugins</groupId>
64            <artifactId>maven-compiler-plugin</artifactId>
65            <version>3.1</version>
66            <configuration>
67              <source>1.6</source>
68              <target>1.6</target>
69              <encoding>UTF-8</encoding>
70            </configuration>
71          </plugin>
72        </plugins>
73      </pluginManagement>
74    </build>
75
76    <properties>
77      <!-- Eliminates the 'build is platform dependent!' warning. -->
78      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
79    </properties>
80
81  </project>

```

38.3.2 The configuration files

Our minimalistic example needs minimal configuration. It mainly needs to provide the linkage to the facelets framework.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
   app-2.5.xsd" version="2.5">
3
4   <description>JavaServer Faces 2.0 - Basic Text Form</description>
5
6   <!-- Faces Servlet -->
7   <servlet>
8     <servlet-name>Faces Servlet</servlet-name>
9     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10    <load-on-startup>1</load-on-startup>
11  </servlet>
12
13  <!-- Faces Servlet Mapping -->
14  <servlet-mapping>
15    <servlet-name>Faces Servlet</servlet-name>
16    <url-pattern>*.jsf</url-pattern>
17  </servlet-mapping>
18
19  <!-- Welcome files -->
20  <welcome-file-list>
21    <welcome-file>index.html</welcome-file>
22  </welcome-file-list>
23
24 </web-app>

```

```

1 <?xml version="1.0"?>
2 <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
   facesconfig-2.0.xsd"
5   version="2.0">
6
7   <!-- No special configuration needed -->
8
9 </faces-config>

```

38.3.3 Project organization

The way the files are organized is shown in the following Figure 38.1.

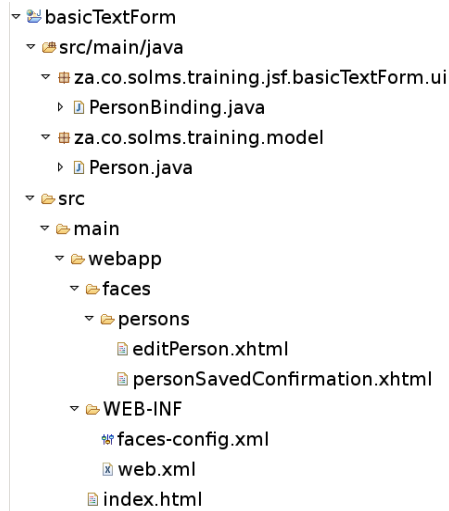
- The JSF configuration files are stored in the `src/main/webapp/WEB-INF` directory.
- An `index.html` which links to the initial facelet to be rendered is in the `src/main/webapp` directory.
- The facelets themselves are stored somewhere in the `src/main/webapp` directory, typically within a categorized tree under a `faces` sub-directory of the `src/main/webapp` directory.

38.3.4 The model

The responsibilities of the *presentation layer* include

- controlling user processes (not business processes for system services),
- capturing service request objects,

Figure 38.1: JSF project organization



- making service requests to service layer,
- receiving & rendering responses.

Domain objects are part of the *model* which is developed with the *business logic layer*. In our minimalistic example we have a simple person class as domain object which will be populated from user input. Hence the `Person` class would come from the module for the business logic layer. The presentation layer POM would have a dependency on the business logic layer POM in order to use the classes defined in the business logic layer.

38.3.4.1 Person.java

```

1 package za.co.solms.persons.model;
2
3 public class Person {
4     public Person(String firstName, String surname, Title title) {
5         setFirstName(firstName);
6         setSurname(surname);
7         setTitle(title);
8     }
9
10    public String getFirstName() {return firstName;}
11
12    public void setFirstName(String firstName) {this.firstName = firstName; }
13
14    public String getSurname() {return surname;}
15
16    public void setSurname(String surname) {this.surname = surname; }
17
18    public Title getTitle() {return title;}
19
20    public void setTitle(Title title) {this.title = title;}
21
22    public enum Title {Ms, Miss, Mrs, Mr, Mstr, Dr, unknown};
23
24    private String firstName, surname;
25    private Title title;
26 }

```

38.3.5 The managed bean (backing and binding bean)

The `CreatePersonBinding.java` managed bean in this minimalist example

- is the backing bean for the `create-person` view (maintaining the data for the view),
- binds to the services layer, calling the `savePerson` service, and
- controls the flow across views for the `createPerson` process.

38.3.5.1 CreatePersonBinding.java

```

1 package za.co.solms.persons.ui.web;
2
3 import javax.faces.bean.ManagedBean;
4 import javax.faces.bean.SessionScoped;
5 import za.co.solms.persons.model.Person;
6 import za.co.solms.persons.model.Person.Title;
7
8 @ManagedBean
9 @SessionScoped
10 public class CreatePersonBinding {
11     public CreatePersonBinding(){}
12
13     public Person getPerson() {return person;}
14     public void setPerson(Person person) {this.person = person;}
15
16     public Title[] getTitles() {return Person.Title.values();}
17
18     public String savePerson() {
19         //try {
20         //    servicesLayer.savePerson(person);
21         //    return "personSavedConfirmation";
22         //}
23         //catch (SomeException e) { return "someOtherView"; }
24     }
25     private Person person = new Person("", "", Person.Title.unknown);
26 }

```

38.3.6 The view

The contains `index.html` which forwards a request for the home facelet, and two facelet pages, one for the form capturing the person details and a second for the page confirming the capturing of the person details.

38.3.6.1 index.html

The `index.html` file contains a meta tag to request the browser to replace the content with what is provided through an alternate JSF URL. The JSF engine interprets the JSF request and parses the XHTML facelet through the appropriate rendering kit.

```

1 <html>
2 <head>
3     <meta http-equiv="refresh" content="0; URL=faces/persons/editPerson.jsf" >
4 </head>
5 </html>

```

38.3.6.2 editPerson.xhtml

The edit person facet imports the JSF core and JSF HTML tag name spaces. Note that in the document all html components are now replaced with facetlet-html components. Pure content markup like paragraphs, titles, header, body, ..., are done in XHTML, but the actual components are from the JSF html tag library.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/
  DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.
  sun.com/jsf/html">
4 <head>
5 <title>Basic Text Form: Create Person</title>
6 </head>
7 <body>
8 <h1><h:outputText id="header1" value="Basic Text Form" /></h1>
9 <h2><h:outputText id="header2" value="Edit Person" /></h2>
10
11 <h:form id="form">
12 <h:panelGrid id="grid" columns="2">
13 <h:outputLabel id="firstNameLabel" value="First Name:" for="firstNameField" />
14 <h:inputText id="firstNameField" value="#{createPersonBinding.person.firstName}" />
15
16 <h:outputLabel id="surnameLabel" value="Surname:" for="surnameField" />
17 <h:inputText id="surnameField" value="#{createPersonBinding.person.surname}" />
18
19 <h:outputLabel id="titleLabel" value="Title:" for="titleLabel" />
20 <h:selectOneMenu id="titleLabel" value="#{createPersonBinding.person.title}" />
21 <f:selectItems value="#{createPersonBinding.person.title}" />
22 </h:selectOneMenu>
23
24 <h:commandButton id="save" action="#{createPersonBinding.savePerson}" value="save" />
25 </h:panelGrid>
26 </h:form>
27 </body>
28 </html>

```

The linkage between the view and the backing bean is done using unified expression language expressions.

38.3.6.2.2 The component tree

38.3.7 Building, deploying and running the web app

To build and package the application in a web archive (**war** file) we use

```
1 mvn package
```

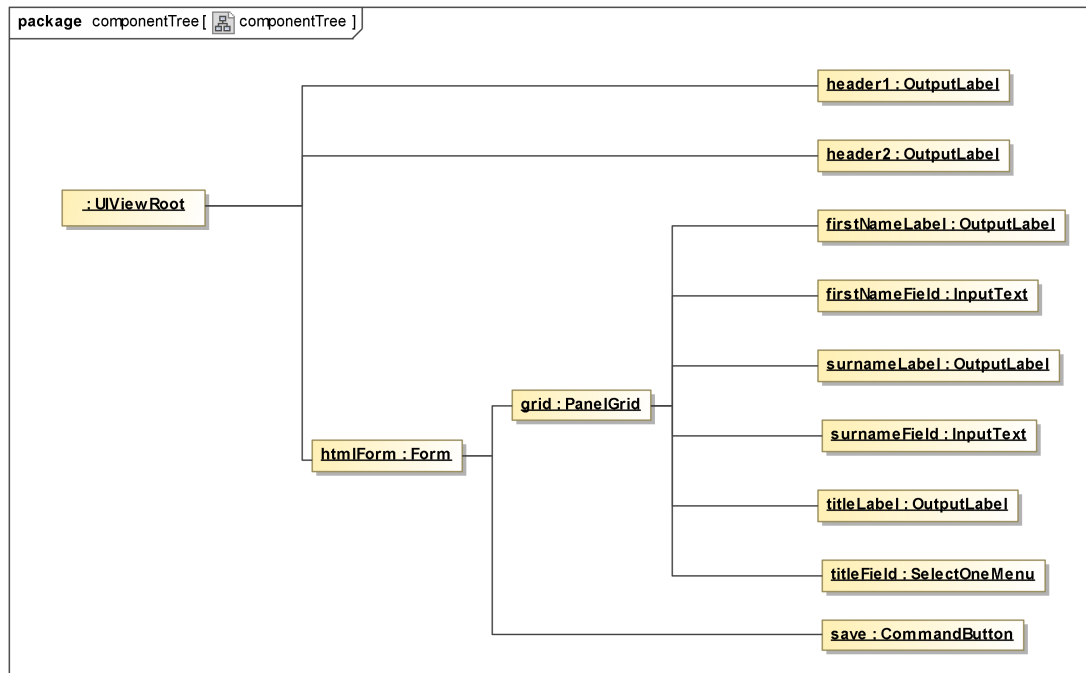
If we use

```
1 mvn install
```

the generated war and its associated pom will also be copied into the local maven repository so that it can be used by other projects which have dependencies on our project.

This web archive which can be found in the **target** directory can be deployed in any Java-EE 6 compliant web container or application server. This is typically done by simply copying the **war** file into the respective deploy directory.

Figure 38.2: The component tree for this view



However, our `pom` contains a *jetty-maven-plugin* build configuration which enables us to execute the `run-exploded` goal (function) of the embedded *Jetty* web container to which executes the web container and deploys the unpacked war in memory.

```
1 mvn jetty:run-exploded
```

38.3.8 Executing our minimalistic web app

To execute our minimal web application we open a web browser with the URL

```
1 http://localhost:8080/basicTextForm
```

If you deployed the web archive onto a stand-alone web container, you will have to replace local host by the URL of the machine hosting the web container and set the port to the port assigned to the web container (usually port 80 for production machines).

38.3.9 Exercise

Write a little web application which takes a string as input and, upon pressing a *convert* button, converts the string to upper case and shows it in a respective view.

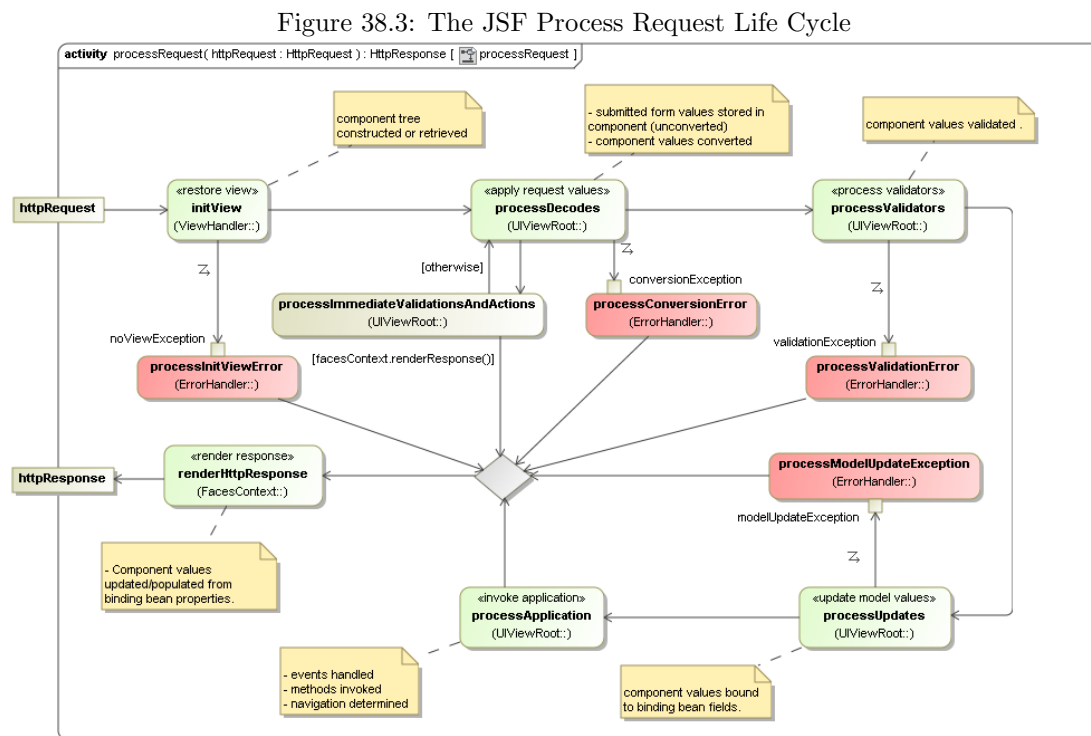
38.4 The JSF Request Life-Cycle

38.4.1 The JSF Request Life-Cycle

It is important to understand the JSF request life cycle including the various phases and the transitions between these phases in order to understand when data extraction, population, conversion and validation are done and in order to understand how AJAX works in JSF.

38.4.1.1 Overview

The JSF request life cycle has 6 phases.



38.4.1.2 Request Life Cycle Phases

This section discusses the activities which are done in the various request life cycle phases.

38.4.1.2.1 The Restore View Phase In this phase the JSF component tree is either constructed or amended, i.e. update the existing component tree if there is one and if not, construct a new the JSF component tree from the Facelet page specification.

JSF supports *partial state saving* which can provide a significant performance benefit over JSP which has to re-render the entire page for each request. This is required for AJAX.

38.4.1.2.2 The Apply Request Values Phase In this phase the each component is requested to update its state from the information contained in the request. This is done by calling `processDecodes(facesContext)` on the root of the component tree, `UIViewRoot` which forwards that request up the tree. **Note:** *Components can access the request header, request parameters, and cookies to update its state. Once a component state has been updated from the request it is said to have "local value".*

38.4.1.2.3 The Process Validations Phase Starting from the root of the component tree, the user data for each component is validated and converted. For example, a user might have entered the text 2014-05-08. This phase

- Validates the input against the associated validators. These are typically text pattern validators which could be either one of the standard validators like length validators or a user defined validator.
- Converts the input to the actual data type (e.g. int, double, date, ...) using the assigned converter. The converter could be one of the standard JSF supplied converters or a custom converter.

38.4.1.2.3.1 Validation and conversion failures Should either the validation or the conversion fail

- appropriate error messages are generated,
- the remaining phase up to the rendering phase are omitted, and
- the page is re-rendered with the the error messages.

38.4.1.2.4 The Update Model Values Phase In this phase the getters and setters on the model are called. This can be directly on the backing bean like `personBinding.setSurname(...)` or accessing the actual model via the backing bean via `personBinding.person.setSurname(...)`.

38.4.1.2.5 The Invoke Application Phase In this phase the action events are executed. These action events typically

- call business services in the services layer and
- bring up the next view for user work flow.

In addition to the action attribute on action components like buttons and menus, one can also register further action listeners who will be called prior to the core action event of the component which triggered the request process.

38.4.1.2.6 The Render Response Phase In this phase

- the component tree is assembled starting with `UIViewRoot` and
- is rendered by calling `encodeAll()` on the root element which makes use of the renderers associated with the individual components in order to convert the model data to the rendered text.

38.4.1.3 Immediate event handling

The immediate event handling property of JSF is used to handle events that normally don't necessitate validating the entire form. This is typically the case when certain components are dynamically added or removed from the form.

By setting the `immediate` attribute of a component to `true` (by default it is `false`), the conversion, validation as well as the execution of any actions and action listeners for that component is brought forward to the *apply-request-values* phase.

For example, assume you are capturing person details and, depending on whether the person is busy studying, you want to capture the details of the qualification he or she is studying for. You could have a "busy studying" check box and only if that check box is ticked, show fields through which you capture the details of the qualification they are studying for.

Because the check box is used only to populate the form with additional fields (or to remove them), there is no need for the entire form to be in a valid state when processing this event. We thus need not validate the form in its entirety and can simply go ahead and perform the conversion, validation and any action events for that component only. In this case, you would use immediate event handling.

Normally the other components are taken through the standard life cycle. However, if an action listener calls `facesContext.renderResponse()`, then the rest of the life cycle phases up to the *render phase* are skipped. **Note:** *The rest of the components are thus not yet converted or validated and need thus not yet have valid values.*

38.5 Navigation

Navigation can be controlled in three ways in JSF

1. Implicit navigation via direct linkage from one view to another view specified in an *action attribute* of a `UIComponent`.
2. Explicit navigation as a return value of an *action method* in the binding (managed) bean which acts as presentation layer controller for the view or for a process.
3. Separately configured navigation via potentially global navigation rules specified in the `faces-config.xml` file.

38.5.1 Navigation rules

The traditional way of specifying navigation in JSF is to add navigation rules to the `faces-config.xml`. This is done within `<navigation-rule>` elements.

38.5.1.1 Navigation cases for page

One can specify, for a particular view, the various navigation cases as follows:

```

1 <navigation-rule>
2   <from-view-id>/faces/persons/editPerson.xhtml</from-view-id>
3   <navigation-case>
4     <from-outcome>personSaved</from-outcome>
5     <to-view-id>/faces/persons/personSavedConfirmation.xhtml</to-view-id>
6     <redirect/>
7   </navigation-case>
8   <navigation-case>
9     <from-outcome>personExists</from-outcome>

```

```

10 <to-view-id>/faces/persons/personExistsNotification.xhtml</to-view-id>
11 </navigation-case>
12 </navigation-rule>

```

This states that if, for the `editPerson` view, the outcome of an action event is "*person-Saved*", then the `personSavedConfirmation.xhtml` page should be shown and if the outcome is "*personExists*", then the `personExistsNotification.xhtml` should be shown.

38.5.1.2 Global navigation rules

One can also specify global navigation rules by using a wild card for the page. For example, if one wants to specify that a *logout* outcome should always result in a navigation to the home page (no matter from which page the logout was requested), then this could be done as follows:

```

1 <navigation-rule>
2 <from-view-id>*</from-view-id>
3 <navigation-case>
4 <from-outcome>logout</from-outcome>
5 <to-view-id>/home.xhtml</to-view-id>
6 <redirect/>
7 </navigation-case>
8 </navigation-rule>

```

38.5.1.3 Using EL expressions in navigation rules

Instead of specifying an outcome, one can also specify a navigation for an action bound via an EL expression to a managed bean action method. This can be view-specific or generic.

For example, the following specifies that for any page which has a component bound to the `logout` action, the navigation should be to the `logout` page:

```

1 <navigation-rule>
2 <navigation-case>
3 <from-action>#{userBinding.logout}</from-action>
4 <to-view-id>/logout.xhtml</to-view-id>
5 </navigation-case>
6 </navigation-rule>

```

38.5.1.4 Handle generic outcome

Different actions across views may lead to a similar outcome. For example, many actions may require a user to be logged in and may hence have as outcome that login is required. One can specify a general navigation case that, for any view, the *loginRequired* outcome leads to the `user/login.xhtml` view:

```

1 <navigation-case>
2 <description>
3   Handle a generic error outcome that might be returned
4   by any application Action.
5 </description>
6 <from-outcome>loginRequired</from-outcome>
7 <to-view-id>/user/login.xhtml</to-view-id>
8 </navigation-case>

```

38.5.1.5 Pros and cons of navigation-rules based navigation

| Pros | Cons |
|-------------------------------------|--|
| Can specify global navigation rules | linking rules to views (not processes) |
| Navigation rules managed at 1 place | limits reuse |
| | More work to specify rules |

38.5.2 Backing bean based navigation

Typically action events of components are bound to action methods of the associated binding component. These action services would typically call some service on the business process/services layer and then either

- receive the result and show it in another page to the user, or
- in the context of the services layer providing the service and requiring further user input, be requested to capture that user input. **Note:** *The latter is not directly possible and will require us to insert a infrastructural component which will map such a request onto a return message.*

Backing bean based navigation provides a very maintainable navigation specification which is particularly appropriate when the backing bean manages user processes.

38.5.2.1 Binding the component's action to an action method

The binding to an action method is done

```
1 <h:commandButton id="submitOrder" action="#{createOrderBinding.processOrder}" value="Submit order"/>
```

38.5.2.2 The action method

The binding or managed bean would have to have a `processOrder` service which gets no parameters and returns a string specifying the navigation decision.

```
1 @ManagedBean
2 @ViewScoped
3 public class OrderBean
4 {
5     public String processOrder()
6     {
7         this.invoice = orderService.processOrder();
8     }
9     return "showInvoice";
10 }
11 }
```

38.5.3 Implicit navigation

This is the least-preferred of the navigation specification options. In implicit navigation views are directly linked. The disadvantages include the following:

- A view which is directly linked to another view is less reusable across processes.

- The view is no longer only a view but is polluted with control logic. This pollutes the MVC pattern and the responsibility localization.
- Specifying some navigation rules (and one can typically only define some of them in this way because others will be determined by the response from the services layer) results in one having to maintain the flow logic in multiple places.

As an example, consider the following facelet snippet which contains a *help* button whose action attribute requests navigation to a help view contained in the same folder as the current view:

```
1 <h:commandButton id="helpButton" value="Help" action="./help.xhtml"/>
```

38.6 Event processing

One of the strengths of JSF is that one is able to register event listeners to UI components and handle these events in event listeners. Events go into an event queue and the different registered event listeners are given the event for event processing. JSF supports user and system triggered events. The user triggered events include *action events* and *change events*. Amongst the system generated events, *phase events* prove to be very useful for a JSF developer.

38.6.1 Value change events

38.6.1.1 Value change events

JSF supports the registration of value-change listeners who process value change events fired from UI components.

They are particularly useful when one wants to change the content of a form depending on values entered in certain components of that form.

Value change listeners will receive a `ValueChangeEvent` through which they can get access to

- the new and old value of the component whose value change they are observing (via the `getNewValue()` and `getOldValue()` services), and
- the component which issued the event via the `getComponent()` method.

38.6.1.1.1 Example As an example, consider a minimalist *capture person details* web application which also captures employer details in the case where the person is employed.

38.6.1.1.1.1 The facelet To do this we add a *"employed"* checkbox to our view for which we request `immediate` event handling:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional/EN" "http://www.w3.org/TR/xhtml1/
  DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.
  sun.com/jsf/html">
4 <head>
5 <title>Basic Text Form: Create Person</title>
6 </head>
7 <body>
```

Figure 38.4: Showing additional fields on value change events

Basic Test Form

Edit Person

First Name:

Surname:

Title: unknown ▾

Employed: ☒

Employer Name:

Telephone No:

save

```

8      <h1><h:outputText id="header1" value="Form with employer fields only shown when employed."/></h1>
9      <h2><h:outputText id="header2" value="Edit Person"/></h2>
10
11     <h:form id="form">
12         <h:panelGrid id="personGrid" columns="2">
13             <h:outputLabel id="firstNameLabel" value="First Name:" for="firstNameField"/>
14             <h:inputText id="firstNameField" value="#{createPersonBinding.person.firstName}"/>
15
16             <h:outputLabel id="surnameLabel" value="Surname:" for="surnameField"/>
17             <h:inputText id="surnameField" value="#{createPersonBinding.person.surname}"/>
18
19             <h:outputLabel id="titleLabel" value="Title:" for="titleField"/>
20             <h:selectOneMenu id="titleField" value="#{createPersonBinding.person.title}">
21                 <f:selectItems value="#{createPersonBinding.titles}"/>
22             </h:selectOneMenu>
23
24             <h:outputLabel id="employedLabel" value="Employed:" for="employedField"/>
25             <h:selectBooleanCheckbox id="employedField"
26                 value="#{createPersonBinding.employed}"
27                 valueChangeListener="#{createPersonBinding.employmentChanged}"
28                 immediate="true" onclick="this.form.submit()"/>
29
30         </h:panelGrid>
31
32         <h:panelGrid id="employerGrid" rendered="#{createPersonBinding.employed}" columns="2">
33             <h:outputLabel id="employerNameLabel" value="Employer Name:" for="employerNameField"/>
34             <h:inputText id="employerNameField" value="#{createPersonBinding.employer.name}"/>
35
36             <h:outputLabel id="employerTelNoLabel" value="Telephone No:" for="employerTelNoField"/>
37             <h:inputText id="employerTelNoField" value="#{createPersonBinding.employer.telephoneNo}"/>
38         </h:panelGrid>
39
40         <h:commandButton id="save" action="#{createPersonBinding.savePerson}" value="save"/>
41     </h:form>
42 </body>
43 </html>

```

Note that we registered the `employmentChanged` method as value change listener of the checkbox. Note also that the employer grid is requested to be only conditionally rendered.

38.6.1.1.1.2 The managed bean In the backing bean (managed bean) we add the corresponding value change listener which sets the `employed` field (since the immediate validation/actions precede the updating of the state variables for the component) and request the faces

context to render the response which will be the updated page.

```

1 package za.co.solms.persons.ui.web;
2
3 import javax.faces.bean.ManagedBean;
4 import javax.faces.bean.SessionScoped;
5 import javax.faces.context.FacesContext;
6 import javax.faces.event.ValueChangeEvent;
7
8 import za.co.solms.persons.model.Company;
9 import za.co.solms.persons.model.Person;
10 import za.co.solms.persons.model.Person.Title;
11
12 @ManagedBean
13 @SessionScoped
14 public class CreatePersonBinding
15 {
16     public CreatePersonBinding(){}
17
18     public Person getPerson()
19     {
20         return person;
21     }
22
23     public void setPerson(Person person)
24     {
25         this.person = person;
26     }
27
28     public boolean isEmployed()
29     {
30         return employed;
31     }
32
33     public void setEmployed(boolean employed)
34     {
35         this.employed = employed;
36     }
37
38     public Company getEmployer()
39     {
40         return employer;
41     }
42
43     public void setEmployer(Company employer)
44     {
45         this.employer = employer;
46     }
47
48     public void employmentChanged(ValueChangeEvent e)
49     {
50         Boolean newEmployedValue = (Boolean) e.getNewValue();
51         if (newEmployedValue == true)
52             employed = true;
53         else
54             employed = false;
55         FacesContext.getCurrentInstance().renderResponse();
56     }
57
58     public Title[] getTitles() {return Person.Title.values();}
59
60     public String savePerson()
61     {
62         if (employed)
63             person.setEmployer(employer);
64         //try
65         //{
66         //    servicesLayer.savePerson(person);
67         //    return "personSavedConfirmation";
68         //}
69         //catch (SomeException e)
70         //{

```



```

71     // return "someOtherView";
72     //}
73 }
74
75 private boolean employed = false;
76 private Person person = new Person("", "", Person.Title.unknown);
77 private Company employer = new Company("", "");
78 }

```

Now, when clicking the *employed* check box, the additional fields for capturing the employer details are shown without the remainder of the form having to be in a valid state.

38.6.1.1.2 Exercises Write a little web application which captures the client name and email address and optionally credit card details or the number of the client's account depending on whether the client chooses to buy the goods on account or via a credit card.

38.6.2 Action events

Control components like `CommandButtons` and `CommandLinks` issue action commands which typically trigger some business service. In addition to the command action itself one can register multiple action listeners which will be actioned prior to the actual action command. Whilst the action command itself is meant to trigger the user requested business service, the action listeners are meant for actions which should be confined to the presentation layer itself. This enables one to keep the presentation layer logic cleanly separated from the business logic. One could, for example, use an action listener to construct the request object from the data filled into the form and the actual command action to request the business service from the services layer. **Note:** *Action listeners are also extensively used for processing Ajax events.*

The use of action listeners is identical to that of value change listeners. One registers the action listener with a component, and implements the specified action listener which receives an `ActionEvent`.

38.6.3 Phase events

JSF also supports the specification of phase event listeners which are called prior and after each request processing phase.

Unlike value change and action listeners which are commonly used to specify more complex presentation layer logic, phase listeners are generally used either

- for debugging purposes or to
- extend the functionality of the JSF framework itself.

Phase listeners are plain Java classes (POJOs) which implement the `PhaseListener` interface which requires a phase listener to implement the `beforePhase(PhaseEvent e)` and `afterPhase(PhaseEvent e)` event handlers as well as the `getPhaseId()` query service which, via flags, returns the set of phases which the phase listener is to intercept. `PhaseId` is effectively an enumeration class which can be any of the life cycle phase of `ANY_PHASE`.

Phase listeners are independent of individual UI components and are registered in the `facesConfig.xml` file within a `lifecycle` tag.

38.6.3.1 Phase events

In this section we show two phase listeners, one, the `PhaseInvocationLogger` which listens to all phase events and logs the entry and exit of each phase and a second, a `RequestLogger` which traverses the parameter map of the HTTP request and logs all parameter values.

38.6.3.1.1 `PhaseInvocationLogger` The phase invocation logger intercepts all phases and logs the start and completion of each phase:

```

1 package za.co.solms.ui.web.phaseListeners;
2
3 import javax.faces.event.PhaseEvent;
4 import javax.faces.event.PhaseId;
5 import javax.faces.event.PhaseListener;
6
7 import org.apache.commons.logging.Log;
8 import org.apache.commons.logging.LogFactory;
9
10 public class PhaseInvocationLogger implements PhaseListener
11 {
12     @Override
13     public void beforePhase(PhaseEvent event)
14     {
15         log.debug("Starting phase: " + event.getPhaseId());
16     }
17
18     @Override
19     public void afterPhase(PhaseEvent event)
20     {
21         log.debug("Completed phase: " + event.getPhaseId());
22     }
23
24     @Override
25     public PhaseId getPhaseId()
26     {
27         return PhaseId.ANY_PHASE;
28     }
29
30     private static Log log = LogFactory.getLog(PhaseInvocationLogger.class);
31 }

```

38.6.3.1.2 `RequestParameterLogger` The phase invocation logger intercepts all phases and logs the start and completion of each phase:

```

1 package za.co.solms.ui.web.phaseListeners;
2
3 import java.util.Map;
4 import java.util.logging.Logger;
5
6 import javax.faces.context.ExternalContext;
7 import javax.faces.context.FacesContext;
8 import javax.faces.event.PhaseEvent;
9 import javax.faces.event.PhaseId;
10 import javax.faces.event.PhaseListener;
11
12 import org.apache.commons.logging.Log;
13 import org.apache.commons.logging.LogFactory;
14
15 public class RequestParameterLogger implements PhaseListener
16 {
17
18     @Override
19     public void afterPhase(PhaseEvent event){}
20
21     @Override

```

```

22 public void beforePhase(PhaseEvent event)
23 {
24     ExternalContext externalFacesContext = FacesContext.getCurrentInstance().getExternalContext();
25     String parametersString = buildParametersString(externalFacesContext.getRequestParameterMap());
26     log.debug(parametersString);
27 }
28
29 private static String buildParametersString(Map<String, String> parametersMap)
30 {
31     StringBuffer result = new StringBuffer();
32     for (String parameterName: parametersMap.keySet())
33         result.append(buildParameterString(parameterName, parametersMap.get(parameterName))).append(eol);
34     return result.toString();
35 }
36
37 private static String buildParameterString(String parameterName, String value)
38 {
39     return parameterName + " => " + value;
40 }
41
42 @Override
43 public PhaseId getPhaseId()
44 {
45     return PhaseId.APPLY_REQUEST_VALUES;
46 }
47
48 private static final String eol = System.getProperty("line.separator");
49
50 private static Log log = LogFactory.getLog(PhaseInvocationLogger.class);
51 }

```

38.6.3.1.3 Registering phase listeners Phase listeners need to be registered. This is usually done in the `faces-config.xml` configuration file:

```

1 <?xml version="1.0"?>
2 <faces-config xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
   facesconfig-2.0.xsd" version="2.0">
3
4     <lifecycle>
5         <phase-listener>
6             za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger
7         </phase-listener>
8         <phase-listener>
9             za.co.solms.ui.web.phaseListeners.RequestParameterLogger
10        </phase-listener>
11    </lifecycle>
12
13 </faces-config>

```

38.6.3.1.4 Program output The output of the application for a particular HTTP request is something like

```

1 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger beforePhase
2 Starting phase: RESTORE_VIEW(1)
3 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger afterPhase
4 Completed phase: RESTORE_VIEW(1)
5 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger beforePhase
6 Starting phase: APPLY_REQUEST_VALUES(2)
7 za.co.solms.ui.web.phaseListeners.RequestParameterLogger beforePhase
8 form:save => save
9 form:surnameField => s
10 form_SUBMIT => 1
11 form:employedField => true
12 form:employerTelNoField => f
13 form:firstNameField => a

```

```

14 form:employerNameField => d
15 form:titleField => Miss
16 javax.faces.ViewState => V1qxYgmb9p6JAuSCloeZwsnfr72wFMoeCydpNTMG5jMF2Q7
17 Bker/Bp6Bxtp/X1ViuODAAp2wr6+/K6UMRx9crrNRpnSbIWIKQd+58zVCq6c6Jr08573BA==
18 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger afterPhase
19 Completed phase: APPLY_REQUEST_VALUES(2)
20 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger beforePhase
21 Starting phase: PROCESS_VALIDATIONS(3)
22 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger afterPhase
23 Completed phase: PROCESS_VALIDATIONS(3)
24 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger beforePhase
25 Starting phase: UPDATE_MODEL_VALUES(4)
26 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger afterPhase
27 Completed phase: UPDATE_MODEL_VALUES(4)
28 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger beforePhase
29 Starting phase: INVOKE_APPLICATION(5)
30 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger afterPhase
31 Completed phase: INVOKE_APPLICATION(5)
32 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger beforePhase
33 Starting phase: RENDER_RESPONSE(6)
34 za.co.solms.ui.web.phaseListeners.PhaseInvocationLogger afterPhase
35 Completed phase: RENDER_RESPONSE(6)

```

38.7 Messaging

Messaging is commonly used to

- notify users of *invalid input*,
- confirm that a *task* was either successfully *completed* or *failed*.

38.7.1 Messages

Messages are instances of a `FacesMessage` class which defines the following severity levels:

- `SEVERITY_INFO`
- `SEVERITY_WARN`
- `SEVERITY_ERROR`
- `SEVERITY_FATAL`

38.7.2 Inserting messages into a message queue

Messages are inserted in a message queue via the

```
1 facesContext.addMessage(String clientId, FacesMessage msg)
```

service. The `clientId` refers to the id of the `UIComponent` which is not the id specified in the facelet, but which needs to be obtained by traversing the component tree (see example discussed in 38.7.4).

38.7.3 The message element

During rendering messages are shown either in a page-wide messages field

```
1 <h:messages/>
```

showing all messages for a page, or a component specific message field

```
1 <h:message for="#{componentId}" />
```

showing all messages for a particular components.

It is regarded as a best practice to have a `<h:messages/>` field on every page. As of JSF 2, such a tag is automatically inserted should one omit to specify one.

38.7.4 Example

In our simple example we generate an error message if the title of the person is *Mstr* and the person is employed - we are staunchly against child labour!!

38.7.4.1 Generating error messages

If the person has title *Mstr* and is *employed* we

1. Find the component which has `titleField` as component identifier using our `getComponentWithComponentId()` service. **Note:** The `getComponentWithComponentId()` could be made into a general utility function.
2. Ask the component for its `clientId` for the context.
3. Add the message to the faces context, specifying the `clientId` if the component was found.
4. Return an empty string or null so that the current page is re-rendered (with the messages).

```
1 package za.co.solms.persons.ui.web;
2
3 import java.util.Iterator;
4 import java.util.logging.Logger;
5
6 import javax.faces.application.FacesMessage;
7 import javax.faces.bean.ManagedBean;
8 import javax.faces.bean.SessionScoped;
9 import javax.faces.component.UIComponent;
10 import javax.faces.context.FacesContext;
11 import javax.faces.event.ValueChangeEvent;
12
13 import za.co.solms.persons.model.Company;
14 import za.co.solms.persons.model.Person;
15 import za.co.solms.persons.model.Person.Title;
16
17 @ManagedBean
18 @SessionScoped
19 public class CreatePersonBinding
20 {
21     public CreatePersonBinding() {}
22
23     public Person getPerson()
24     {
25         return person;
26     }
27
28     public void setPerson(Person person)
29     {
30         this.person = person;
31     }
32
33     public boolean isEmployed()
34     {
35         return employed;
```

```

36 }
37
38 public void setEmployed(boolean employed)
39 {
40     this.employed = employed;
41 }
42
43 public Company getEmployer()
44 {
45     return employer;
46 }
47
48 public void setEmployer(Company employer)
49 {
50     this.employer = employer;
51 }
52
53 public void employmentChanged(ValueChangeEvent e)
54 {
55     Boolean newEmployedValue = (Boolean)e.getNewValue();
56     if (newEmployedValue == true)
57         employed = true;
58     else
59         employed = false;
60     FacesContext.getCurrentInstance().renderResponse();
61 }
62
63 public Title[] getTitles() {return Person.Title.values();}
64
65 public String savePerson()
66 {
67     if (person.getTitle().equals(Person.Title.Mstr) && employed)
68     {
69         FacesMessage msg = new FacesMessage("A " + Person.Title.Mstr
70             + " should not be employed.");
71         FacesContext context = FacesContext.getCurrentInstance();
72         UIComponent component = getComponentWithComponentId(context.getViewRoot(), "titleField");
73         if (component != null)
74             context.addMessage(component.getClientId(context), msg);
75         else
76             context.addMessage(null, msg);
77     }
78     return "";
79 }
80
81 if (employed)
82     person.setEmployer(employer);
83
84 return "personSavedConfirmation";
85 }
86
87 private static UIComponent getComponentWithComponentId(UIComponent parent, String id)
88 {
89     if (id.equals(parent.getId()))
90         return parent;
91
92     Iterator<UIComponent> children = parent.getFacetsAndChildren();
93     while(children.hasNext())
94     {
95         UIComponent found = getComponentWithComponentId(children.next(), id);
96         if(found != null)
97             return found;
98     }
99     return null;
100 }
101
102 private boolean employed = false;
103 private Person person = new Person("", "", Person.Title.unknown);
104 private Company employer = new Company("", "");
105
106 private static Logger log = Logger.getLogger(CreatePersonBinding.class.getName());
107 }

```

38.7.4.2 Message views

On our facelet page we added

- a general `<h:messages/>` tag which shows all messages issued for the page, and
- for each component a component specific `<h:message for="componentId"/>` tag so that messages which displays any messages issued for its corresponding component.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html
3     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5 <html xmlns="http://www.w3.org/1999/xhtml"
6     xmlns:f="http://java.sun.com/jsf/core"
7     xmlns:h="http://java.sun.com/jsf/html">
8 <head>
9 <title>Basic Text Form: Create Person</title>
10 </head>
11 <body>
12 <h1><h:outputText id="header1" value="Form with Error Messages"/></h1>
13 <h2><h:outputText id="header2" value="Edit Person"/></h2>
14
15 <h:messages/>
16
17 <h:form id="form">
18 <h:panelGrid id="personGrid" columns="3">
19 <h:outputLabel id="firstNameLabel" value="First Name:" for="firstNameField"/>
20 <h:inputText id="firstNameField" value="#{createPersonBinding.person.firstName}"/>
21 <h:message for="firstNameField"/>
22
23 <h:outputLabel id="surnameLabel" value="Surname:" for="surnameField"/>
24 <h:inputText id="surnameField" value="#{createPersonBinding.person.surname}"/>
25 <h:message for="surnameField"/>
26
27 <h:outputLabel id="titleLabel" value="Title:" for="titleField"/>
28 <h:selectOneMenu id="titleField" value="#{createPersonBinding.person.title}"/>
29 <f:selectItems value="#{createPersonBinding.titles}"/>
30 </h:selectOneMenu>
31 <h:message for="titleField"/>
32
33 <h:outputLabel id="employedLabel" value="Employed:" for="employedField"/>
34 <h:selectBooleanCheckbox id="employedField"
35     value="#{createPersonBinding.employed}"
36     valueChangeListener="#{createPersonBinding.employmentChanged}"
37     immediate="true" onclick="this.form.submit()"/>
38 <h:message for="employedField"/>
39
40 </h:panelGrid>
41
42 <h:panelGrid id="employerGrid" rendered="#{createPersonBinding.employed}" columns="2">
43 <h:outputLabel id="employerNameLabel" value="Employer Name:" for="employerNameField"/>
44 <h:inputText id="employerNameField" value="#{createPersonBinding.employer.name}"/>
45
46 <h:outputLabel id="employerTelNoLabel" value="Telephone No:" for="employerTelNoField"/>
47 <h:inputText id="employerTelNoField" value="#{createPersonBinding.employer.telephoneNo}"/>
48 </h:panelGrid>
49
50 <h:commandButton id="save" action="#{createPersonBinding.savePerson}" value="save"/>
51
52 </h:form>
53 </body>
54 </html>

```

If there are no messages in the message queue, the message fields are not shown.

On the other hand, should we fill in the form specifying an employed *Mstr*, then the page is re-rendered with the component specific message in a message field for that component and all messages accumulated in the general messages field.

Figure 38.5: Message fields not visible if no messages rendered

Form with Error Messages

Edit Person

First Name:

Surname:

Title:

Employed: ☐

Employer Name:

Telephone No:

Figure 38.6: Accumulated and component specific messages

Form with Error Messages

Edit Person

- A Mstr should not be employed.

First Name:

Surname:

Title: A Mstr should not be employed.

Employed: ☒

Employer Name:

Telephone No:

Note: Typically one will use styling to render the messages in another font and/or colour.

Chapter 39

The Unified Expression Language

39.1 Overview

The *Unified Expression Language* (EL) was developed as a merger of expression languages used in JSP and JSF development.

The core purpose of the EL is to enable views (e.g. facelets) to access backing bean properties and services. For this purpose the EL provides a very simple syntax.

For example, in order to access, from a facelet, the **name** of the **person** property maintained by the **createPerson** backing bean, one can use

```
1 ${createPersonBinding.person.name}
```

and to request the **persistPerson** service of that same backing bean we can either use

```
1 ${createPersonBinding.persistPerson}
```

or, should one prefer to highlight that a service is requested, we could alternatively use

```
1 ${createPersonBinding.persistPerson()}
```

Note: *The expression*

```
1 ${createPersonBinding}
```

resolves the backing bean itself. This is done via the page context using the

```
1 PageContext.findAttribute(String name)
```

service.

The expressions may also contain logic which may be relevant for the view. One could use expressions like

```
1 <c:if test="${captureOrderBean.order.lineItems.size == 0}">
```

Note: *This is an EL expression which requests immediate evaluation.*

39.2 Immediate and deferred expression evaluation

EL supports both *immediate* and *deferred* expression evaluation.

In the case of **immediate Evaluation** the EL expression is compiled when the page is compiled and the processing engine (JSF or JSP engine) evaluates the expression immediately when the page is rendered. Immediate evaluation is used in JSP, but is not suited for JSF which uses a multi-phase life cycle and has a more complex component model. Immediate evaluation expressions are always read-only expressions, i.e. **rvalue** expressions and not **lvalue** expressions.

Immediate evaluation is requested by prepending the EL expression with a \$:

```
1 <c:forEach var=?lineItem? items=?${order.lineItems}? >
```

In the case of **deferred evaluation** the processing engine can decide itself when it needs to evaluate the expression. The expression may even be evaluated multiple times. For example, for the initial *get* request of a page, the EL expression is evaluated solely during the *render* phase. But for the processing of subsequent *post* requests, the EL expression may be evaluated during the *apply request values*, *process validations*, and *update model* phases. Deferred evaluation is suited for multi-phase life cycles and complex component models. It is used by default in JSF.

To request deferred evaluation of an EL expression, prepend the expression with #:

```
1 <h:inputText value="#{createPersonBinding.name}"/>
2
3 <h:commandButton action="#{createPersonBinding.persistPerson()}/>
```

You can mix JSF component tags with JSTL (JSP Tag Library) tags (e.g. the core JSTL action tags). Preferably this should be done only in exceptional cases and one needs to take care as JSP EL expressions (those used in the JSTL tags) are evaluated immediately whilst the expressions used in JSF component tags are evaluated at the appropriate time in the JSF life cycle.

For this reason there is a need to be able to request immediate evaluation of EL expressions when using JSTL tags. This can be done by prepending the EL expression with a \$ sign:

```
1 <c:forEach var=?lineItem? items=?${order.lineItems}? >
```

Note: For JSPs *#someExpression* and *\$someExpression* are both evaluated in the same way, using immediate evaluation.

39.3 Value expressions

You can refer to objects and object properties using value expressions. The EL provides 2 alternative notations to refer to object properties, you can use either

```
1 #{person.name}
```

or

```
1 #{person[name]}
```

39.3.1 Navigating across object graphs

You can navigate along the object graph with further levels of de-referencing:

```
1 #{book.author.address.country}
```

or

```
1 #{book.author.address[country]}
```

39.3.2 Accessing array or list elements

You can access array or list elements using an index.

For example

```
1 #{showPersonBinding.person.qualifications[1]}
```

will bind to the first qualification in the list or array of qualifications. This can be combined with a JSTL `for` tag

```
1 <ul>
2 <c:set var="numQualifications" value="${showPersonBinding.person.qualifications.size()}" />
3 <c:forEach var="i" begin="1" end="${numQualifications}">
4     <li>
5         #{showPersonBinding.person.qualifications[${i}]}
6     </li>
7 </c:forEach>
8 </ul>
```

39.3.3 Accessing enumeration values

EL also enables you to refer to enumeration values. For example, if you have

```
1 public enum Mood {ecstatic, happy, content, irritated, furious}
```

then EL allows one to test for an enumeration value

```
1 <c:if test="${showPersonBinding.person.mood == 'happy'}">
```

39.4 Method expressions

In EL a service can be requested via

```
1 #{captureOrderBinding.processOrder}
```

or

```
1 #{captureOrderBinding.processOrder()}
```

39.4.1 Passing parameters to methods

JSF also enables one to pass parameters to methods. For example

```
1 <h:commandButton value="muteComplaint"
2   action="#{complaintCaptureBinding.complaint.text.replaceAll('[!]+','')}" />
```

would replace all exclamation marks (and consecutive sequences of explanation marks) with a single full stop.

39.5 EL operators

EL supports the standard set of operators including

- **Logical operators:** and `&&`, or `||`, not `!`,
- **Arithmetic operators:** `+` `-` `*` `/` `div` `%`
- **Relational operators:** `==` `<` `<=` `>` `>=` `!=`
- **The empty operator:** `empty`
- **Conditional operator:** `booleanExpression ? A : B`

39.6 Accessing context objects

In JSF you can use the EL to access a range of context objects through predefined variables:

- **facesContext:** provides access to the faces context.
- **requestScope:** provides access to the request map.
- **sessionScope:** provides access to the session map.
- **applicationScope:** provides access to the application map.
- **view:** provides access to the view root.
- **param:** provides access to the request parameter map as an object map.
- **paramValues:** provides access to the request parameter values map in the form of a string array.
- **header:** provides access to the request header map
- **headerValues:** provides access to the request header map values as a string array.
- **cookie:** provides access to the request cookie map.

Chapter 40

JSF components

40.1 Overview

JSF supports a component model which includes

- a standard component library with common components like text fields, drop-down lists, check boxes, menus and so on,
- the ability to develop custom components and distribute them in component tag libraries,
- the ability to listen to events generated by components, and
- the ability to support bookmarking in the context of pages which are dynamically generated via the JSF engine.

40.2 JSF standard components

JSF provides a rudimentary, but solid set of core components.

40.2.1 Standard tag libraries

The JSF standard tag libraries are

- the **core tag library**, and
- the **HTML custom tag library**.

40.2.1.1 The core tag library

This tag library has the core components and functionality which is independent of any specific rendering kit (and hence independent of HTML). Its namespace is `http://java.sun.com/jsf/core` and its elements are commonly imported into a namespace prefix `f`:

```
1 <!DOCTYPE html
2 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5 <html xmlns="http://www.w3.org/1999/xhtml"
6 xmlns:f="http://java.sun.com/jsf/core">
```

The core tag library is often directly used to access functionality which is independent of the rendering. Examples include

- adding configurable attributes,
- customizing facets of structured components,
- validation and conversion,
- event listener registration, and
- specifying Ajax functionality which should be used for a view.

40.2.1.1.1 Adding configurable attributes One can specify component attributes and add configurable attributes to components via the attribute tag:

```
1 <h:commandButton action="#{doSomethingBinding.doSomething}">
2 <f:attribute name="value" value="press me"/>
3 <f:attribute name="myAttributeName" value="myAttributeValue"/>
4 </h:commandButton>
```

40.2.1.1.2 Customizing Facets Structured components like `panelGrid` and `dataTable` allow you to specify that a particular component should be used to represent a particular facet (aspect) of the parent component. For example

```
1 <h:dataTable value="#{order.orderItems}" var="item">
2
3 <h:column>
4 <f:facet name="header">Product Name</f:facet>
5 #{item.productName}
6 </h:column>
7
8 <h:column>
9 <f:facet name="header">Quantity</f:facet>
10 #{item.quantity}
11 </h:column>
12
13 </h:dataTable>
```

```
1 <h:inputText value="#{myBackingBean.myDate}">
2 <f:convertDateTime dateStyle="short" locale="za"/>
3 <f:validator validatorIf="za.co.solms.IsWeekendValidator"/>
4 </h:inputText>
```

40.3 Observer (event listener) registration

The services for registering event listeners to JSF components are provided by the core tag library. This includes the registration of

- phase event listeners,
- action event listeners,

- property action listeners, and
- value change listeners.

For example,

```
1 <h:selectBooleanCheckbox onclick="this.form.submit()"
2 value="#{captureOrderBinding.requireShipping}" immediate="true"
3 <f:valueChangeListener="#{captureOrderBinding.shippingRequirementsChanged}"/>
```

40.3.0.1.4 Requesting Ajax functionality One can use core component tags to specify any Ajax functionality which should be used for a view:

```
1 <h:table id="coursesTable" var="course" value="#{manageCoursesBinding.courses}">
2 ...
3 <f:commandLink action="deleteCourse" value="delete course">
4 <f:ajax render=":form:coursesTable"/>
5 </f:commandLink>
6 ...
7 </h:table>
```

40.3.0.2 The HTML custom tag library

The HTML tag library provides the core user interface components like text fields, buttons and menus, drop-down lists and check boxes and so forth. This library is specific to the HTML rendering kit.

40.3.1 Generic component attributes

There is a range of attributes which is common to all HTML components. These include

- **id :String:** An identifier for the component which needs to be unique to the page. The JSF framework will generate a unique **id** for those components for which no **id** was specified.
- **value :Object:** The value of the object which is shown to the user. This can be text or a value expression.
- **rendered :boolean:** This attribute determines whether the component will be visible or not.
- **immediate :boolean:** If this attribute is set, the conversion, validation as well as the execution of any actions and action listeners for that component is brought forward to the *apply-request-values* phase. This is useful if the entire page need not be parsed.
- **converter:** This attribute specifies which converter should be used to convert the user input to the internal data type.
- **validator:** This attribute specifies the validator which should be used to validate the data provided by the user.
- **style:** The CSS style to be applied to the component.
- **styleClass:** The CSS style class (or classes) to be assigned to the component. If there are multiple, the class names are separated by spaces.

40.3.2 UIViewRoot

The root of the component tree is an instance of the class `UIViewRoot`.

`UIViewRoot` provides a lot of the core JSF functionality like

- processing encoding and decoding of the view,
- processing the validators across the tree components,
- managing the phase listeners for the view,
- generating unique identifiers for components inserted into the tree, and
- managing view events.

40.3.3 UIComponent

This is the base class for all faces components. It provides the common under-the hood functionality required by the JSF framework from any component and also enables the user to specify, amongst other things,

- whether the component should be rendered or not (using the `rendered` attribute for the corresponding component tag), e.g.

```
1 <h:commandButton value="#{msgs.placeOrder}"
2   rendered="#{captureOrderBinding.conditionsAccepted}"
3   action="#{captureOrderBinding.submitOrder()}" />
```

- a `disabled` attribute which is a boolean flag that, when set to `true`, indicates that this component should not receive focus or be included in a form submit.

40.3.3.1 Components and component tags

The use of components is usually not directly requested in Java code. Presentation layer developers normally request the use of components by inserting component tags into the facelet XHTML file.

Thus, for concrete JSF component there is a corresponding component tag. For example, the component tag for `HtmlCommandButton` is `<h:commandButton/>`.

40.3.3.2 Assigning standard (X)HTML tags to component tags

Since components are XHTML tags, the user can specify any of the standard (X)HTML tags for any component.

- **accesskey:** The `accesskey` attribute is a standard HTML attribute that sets the access key that transfers focus to this element when pressed.
- **alt:** The `alt` attribute is a standard HTML attribute that sets the alternate textual description rendered by this component.
- **id:** A unique identifier (unique for the page) for the component. This can be used to, for example, uniquely associate labels with the component, or to assign some styling to a particular component.

- **onblur:** The **onblur** attribute sets the JavaScript code to execute when this component loses focus. **Note:** *In JSF JavaScript is usually used within the AJAX framework.*
- **onchange:** The **onchange** attribute sets the JavaScript code to execute when this component loses focus and its value changes after gaining focus.
- **onclick and ondblclick:** The **onclick** and **ondblclick** attributes set the JavaScript code to execute when the mouse pointer is clicked or double-clicked over this component.
- **onfocus:** The **onfocus** attribute sets the JavaScript code to execute when the component receives focus.
- **onkeydown, onkeyup, onkeypress:** The **onkeydown**, **onkeyup** and **onkeypress** attributes set the JavaScript code to execute when a key is pressed down, up or pressed-and-released over this component.
- **onmouseover, onmouseout, and onmousemove:** The **onmouseover**, **onmouseout**, and **onmousemove** attributes set the JavaScript code to execute when the mouse pointer is moved onto, away from or within this component.
- **onmousedown and onmouseup:** The **onmousedown** and **onmouseup** attributes set the JavaScript code to execute when the mouse pointer is pressed down or released over this component. **Note:** *The JavaScript/AJAX receives the event and from the event it will be able to query which mouse button triggered the event.*
- **onselect:** The **onselect** attribute sets the JavaScript code to execute when the user selects the text contained by this component.
- **style:** The **style** attribute sets the CSS style definition to be applied to this component when it is rendered.
- **styleClass:** The **styleClass** attribute sets the CSS class to apply to this component when it is rendered.
- **tabindex:** The **tabindex** attribute is a standard HTML attribute that sets the order in which this element receives focus when the user cycles through the elements using the TAB key. The value for this attribute must be a short non-signed integer, i.e. an integer between 0 and 32767.
- **title:** The **title** attribute is a standard HTML attribute that sets the tooltip text to display for the rendered component.

40.3.4 UIForm

UIForm is the JSF equivalent of a (X)HTML form. It provides the functionality to capture the data entered into fields and submit it via HTTP POST request. A form typically has buttons through which data is submitted or through which services are requested. **Note:** *A page can have multiple forms.*

As an example, consider the following simple form:

```

1 <h:form>
2
3   <h:panelGrid columns="2">
4     <h:outputLabel id="numeratorLabel" for="numeratorField" value="numerator"/>

```

```

5 <h:inputText id="numeratorField" value="#{dividerBinding.numerator}" />
6 <h:outputLabel id="denominatorLabel" for="denominatorField" value="denominator" />
7 <h:inputText id="denominatorField" value="#{dividerBinding.denominator}" />
8 </h:panelGrid>
9
10 <h:commandButton id="divideButton" value="divide" action="#{dividerBinding.divide()}" />
11
12 </h:form>

```

40.3.5 UIPanel

UIPanel is a layout component which manages how the components are positioned within a container, the panel. There are two concrete subclasses of UIPanel, HtmlPanelGroup and HtmlPanelGrid.

40.3.5.1 HtmlPanelGroup

```

1 <h:form id="form">
2   <h:panelGrid id="grid" columns="1">
3     <h:panelGroup>
4       <h:inputText id="username" value="#{userBean.user.username}"
5         required="#{true}" />
6       <h:message for="username" />
7     </h:panelGroup>
8   </h:panelGrid>
9 </h:form>

```

40.3.5.2 HtmlPanelGrid

HTMLPanelGrids organize the inserted components in a table. One needs to specify the number of columns via the columns attribute of the corresponding <h:panelGrid> component tag.

Components are inserted into the panel grid by nesting the component tags within the tag for the panel grid. The components are inserted row for row, filling first the first row of the grid and then consecutive rows until there are no more components.

```

1 <h:panelGrid columns="2">
2
3   <h:outputLabel id="temperatureLabel" for="temperatureField" value="#{msgs.temperature}" />
4   <h:inputText id="temperatureField" value="#{weatherDetailsBinding.temperature}" />
5
6   <h:outputLabel id="humidityLabel" for="humidityField" value="#{msgs.humidity}" />
7   <h:inputText id="humidityField" value="#{weatherDetailsBinding.humidity}" />
8
9   <h:outputLabel id="ambianceLabel" for="ambianceField" value="#{msgs.ambiance}" />
10  <h:selectOneMenu id="ambianceField" value="#{weatherDetailsBinding.ambiance}">
11    <f:selectItems value="#{weatherDetailsBinding.ambianceValues}" />
12  </h:selectOneMenu>
13
14 </h:panelGrid>

```

40.3.5.2.1 Pre-defined facets Facets are child components of a parent component which play a specific role. They are identified via a name and are stored in a name-value map maintained by the parent component.

Facets can be any components, i.e. they could be output fields for text or graphics or input fields or themselves composite components.

Panel grids have the following pre-defined facets which span across the table columns:

- **header:** A header which spans across the columns of the panel grid and is typically rendered with a specific style above the table.
- **footer:** A footer which spans across the columns of the panel grid and is typically rendered with a specific style below the table.

```

1 <h:panelGrid columns="3">
2
3     <f:facet name="header">
4         <h:panelGroup>
5             <h:outputLabel for="titleText" value="Title:"/>
6             <h:outputText id="titleText" value="Standard JSF single selection components"
7         </h:panelGroup>
8     </f:facet>
9
10    <h:outputText value="HtmlSelectOneRadio"/>
11    <h:outputText value="Radio button group"/>
12    <h:graphicImage url="/images/components/selectOneRadio.png"
13        width="100" height="100" title="HtmlSelectOneRadio"/>
14
15    <h:outputText value="HtmlSelectOneMenu"/>
16    <h:outputText value="Drop down menu"/>
17    <h:graphicImage url="/images/components/selectOneMenu.png"
18        width="100" height="100" title="HtmlSelectOneMenu"/>
19
20    <h:outputText value="HtmlSelectOneListbox"/>
21    <h:outputText value="Single selection list box"/>
22    <h:graphicImage url="/images/components/selectOneListbox.png"
23        width="100" height="100" title="HtmlSelectOneListbox"/>
24
25    <f:facet name="footer">
26        <h:outputText value="CopyLeft Solms Training & Consulting"/>
27    </f:facet>
28 </h:panelGrid>

```

40.3.5.2.2 Pre-defined CSS style classes Panel grids also pre-define a set of CSS style classes which can be used in style sheets to specify the styling preferences for various aspects of panel grids. The following are the CSS classes which have been pre-defined for styling aspects of panel grids:

- **panelClass:** This styling class is used to specify default styling requirements for the panel as a whole, i.e. across all panel components.
- **headerClass:** This styling class is used to specify styling requirements panel header.
- **footerClass:** This styling class is used to specify styling requirements panel footer.
- **rowClasses:** Here one can specify a list of classes which are applied one for one to consecutive rows until the styling classes have been used up, after which the first styling class is applied again.
- **columnClasses:** Here one can specify a list of classes which are applied one for one to consecutive columns until the styling classes have been used up, after which the first styling class is applied again.

For example:

```

1 <h:panelGrid columns="3" rowClasses="evenRowClass, oddRowClass"
2     columnClasses="columnClass" headerClass="headerClass">
3

```

```

4  <f:facet name="header">
5      <h:panelGroup>
6          <h:outputLabel for="titleText" value="Title:" />
7          <h:outputText id="titleText" value="Standard JSF single selection components" />
8      </h:panelGroup>
9  </f:facet>
10
11 <h:outputText value="HtmlSelectOneRadio" />
12 <h:outputText value="Radio button group" />
13 <h:graphicImage url="/images/components/selectOneRadio.png"
14     width="100" height="100" title="HtmlSelectOneRadio" />
15
16 <h:outputText value="HtmlSelectOneMenu" />
17 <h:outputText value="Drop down menu" />
18 <h:graphicImage url="/images/components/selectOneMenu.png"
19     width="100" height="100" title="HtmlSelectOneMenu" />
20
21 <h:outputText value="HtmlSelectOneListbox" />
22 <h:outputText value="Single selection list box" />
23 <h:graphicImage url="/images/components/selectOneListbox.png"
24     width="100" height="100" title="HtmlSelectOneListbox" />
25
26 <f:facet name="footer">
27     <h:outputText value="CopyLeft Solms Training & Consulting" />
28 </f:facet>
29 </h:panelGrid>

```

40.3.5.2.3 Multiple components within grid cell If you want to insert multiple elements into a grid cell, you can group them within a panel group:

```

1  <h:panelGrid columns="2">
2
3      <h:outputLabel for="deliveryAddressField" value="#{msgs.deliveryAddress}" />
4      <
5          h:inputTextarea rows="4" cols="50"
6              id="deliveryAddressField" value="#{captureOrderBinding.deliveryAddress}" />
7
8      <h:outputLabel for="valueAddsField" value="#{msgs.valueAdds}" />
9
10     <h:panelGroup>
11         <h:selectBooleanCheckbox id="speedDeliverySelector"
12             value="#{captureOrderBinding.order.speedDeliveryRequired}" />
13         <h:selectBooleanCheckbox id="wrappingSelector"
14             value="#{captureOrderBinding.order.wrappingRequired}" />
15     </h:panelGroup>
16
17 </h:panelGrid>

```

40.3.6 UICommand

`UICommand` is a sub-class of `UIComponent` which has an action command associated with it. Sub-classes of `UICommand` include

- `HtmlCommandButton`, and
- `HtmlCommandLink`,

which are the Java classes associated with `<h:commandButton/>` and `<h:commandLink/>`. Menu items are also usually implemented as a sub-class of `UICommand`

Command components have an action which typically links to a service in the JSF binding component:

```

1 <h:panelGrid columns="2">
2   <h:commandButton id="processOrderButton" value="#{msgs.processOrder}"
3     action="#{captureOrderBinding.processOrder()}" />
4
5   <h:commandLink id="showOrderConditionsCommand" value="#{msgs.showOrderConditions}"
6     action="#{captureOrder.showOrderConditions()}" />
7 </h:panelGrid>

```

In addition to the main command action, `UICommands` also support the registration of additional action listeners.

40.3.6.1 Passing parameters to the next page

You can add parameters to commands as key-value pairs via:

```

1 <h:commandLink value="show book details" action="#{navigationBinding.showBookDetails}">
2   <f:param name="book" value="#{bookSelectionBinding.selectedBook}" />
3 </h:commandLink>

```

40.3.7 UIOutput

`UIOutput` is a `UIComponent` which is used to render read-only text output. It is the base class of

- `HtmlOutputText`,
- `HtmlOutputLabel`,
- `HtmlOutputLink`, and
- `HtmlOutputFormat`, as well as for
- `UIInput`.

40.3.7.1 HtmlOutputText

This is a very simple component used to render some text:

```

1 <h:outputText value="I am <em>tired</em>"/>

```

Note: Since the facelet is an *XHTML* document which is an *XML* document the standard *XML* rules for escaping characters need to be adhered to.

The text is rendered unformatted. If the text is to be formatted, this should be done with CSS styling for the component.

40.3.7.2 HtmlOutputLabel

40.3.7.2.1 HtmlOutputLabel `HtmlOutputLabels` are `HtmlOutputText` components which can be associated with other components (typically input fields) via a `for` attribute:

```

1 <h:outputLabel value="#{msgs.surname}" for="surnameField"/>

```

`HtmlOutputLinks` are `HtmlOutputText` components which render and provide navigation to an external URL. When the link is traversed, the current page is left and the associated component tree is disassembled.

```

1 <h:outputLink value="http://www.solms.co.za">
2 <h:outputText value="Solms Home Page"/>
3 </h:outputLink>

```

Note: We can insert any output into the `<h:outputLink>` tag including `<h:graphicImage>`.

In addition to the link to a URI, one can pass parameters with the resultant resource request. For example

```

1 <h:outputLink value="http://www.google.co.za/search?q=urdad+analysis+design">
2 <h:outputText>URDAD</h:outputText>
3 <f:param name="q" value="urdad+analysis+design">
4 </h:outputLink>

```

results in the following XHTML rendering:

```

1 <a href="http://www.google.co.za/search?q=urdad+analysis+design">URDAD</a>

```

40.3.7.3 UIOutputFormat

`UIOutputFormat` is a component which supports parametrized output text, i.e. and output text which takes a number of parameters which are inserted for parameter place holders in some text.

For example,

```

1 <h:outputFormat value="Dear {0}. Your order with order number {1} can be tracked using tracking number {2}"
2 >
3 <f:param value="#{orderBean.order.client.firstName}"/>
4 <f:param value="#{orderBean.order.orderNumber}"/>
5 <f:param value="#{orderBean.order.trackingNumber}"/>
6 </h:outputFormat>

```

40.3.8 UIGraphic

40.3.8.1 UIGraphic

`UIGraphic` has one concrete implementation class, `HtmlGraphicImage`. It lets the user specify a graphic image to be inserted into a page:

```

1 <h:graphicImage id="myOldDogImage" url="/images/myFaithfulOldDog.png"
2 width="100" height="100" title="My very old, yet very faithful, loving dog."/>

```

40.3.9 UIInput

`UIInput` is a sub-class of `UIOutput` - it is effectively an editable output field. The following are the standard sub-classes of `UIInput`:

- `HtmlInputText`
- `HtmlInputSecret`
- `HtmlInputTextarea`
- `HtmlInputHidden`
- `UISelectOne`

- `UISelectMany`
- `UISelectBoolean`

Input components are typically inserted into forms.

One can associate with any of the input fields

- a converter,
- multiple validators, and
- multiple value change listeners.

One can also specify for any input component the `immediate` attribute, requesting the immediate processing (conversion, validation, actions, and change listener notification) for that component, not requiring that the entire form is valid.

40.3.9.1 `HTMLInputText`

40.3.9.1.1 `HTMLInputText` This is the supporting class for the `<h:inputText/>` tag. It represents a single-line text input field. Often one specifies the `required`, `size`, and `maxLength` attributes:

```
1 <h:inputText id="surnameField" value="#{personDetailsBinding.person.surname}"
2  required="true" size="25" maxlength="40"/>
```

The `size` attribute specifies the width of the input field in terms of the average number of characters which would be visible.

The `required` and `maxlength` are shorthand mechanisms for requesting the corresponding validators which validate that the field has been provided by the user and that the text entered does not exceed some specified maximum length.

40.3.9.2 `HTMLInputSecret`

This is a `HtmlInputText` field for which the actual characters entered are not shown. Instead a placeholder character (typically `*`) is shown for each character entered.

This field is requested via the `<h:inputSecret/>` tag:

```
1 <h:inputSecret id="passwordField" value="#{loginBinding.password}"
2  required="true" size="16" minlength="6" maxlength="40"/>
```

The `HtmlInputSecret` component is typically rendered as a HTML input element of type *password*.

The `redisplay` attribute is set by default to `false`, ensuring that the password is not rendered on page-refresh.

40.3.9.3 `HTMLInputTextarea`

The `HtmlInputTextarea` component represents a multi-line input text field. One specifies the number of rows and number of columns for the field:

```
1 <h:inputTextarea id="descriptionField" value="#{captureAccidentDetailsBinding.description}"
2  required="false" rows="8" cols="60"/>
```

40.3.9.4 HTMLInputHidden

This is a `HtmlInputText` component which is not visible to the user. It is commonly used as an alternative to cookies to transmit a process id across different pages enabling the framework to tie up subsequent HTTP requests into a single process.

40.3.9.5 UISelectBoolean

The `UISelectBoolean` component represents a single *true* or *false* choice. The standard component library has one concrete subclass of `UISelectBoolean`, `HtmlSelectBooleanCheckbox` which is rendered as a HTML check box. The corresponding HTML component tag is `<h:selectBooleanCheckbox`

```
1 <h:selectBooleanCheckbox id="speedDeliverySelector"
2 value="#{captureOrderBinding.order.speedDeliveryRequired}"/>
```

Note: Recall that JavaBeans standard (and hence for the backing or binding bean) allows the getter of a *boolean* to be named *isXXX*, for example

```
1 @ManagedBean
2 @RequestScoped
3 public class CaptureOrderBinding
4 {
5 ...
6 public boolean isSpeedDeliveryRequired() {return speedDeliveryRequired;}
7
8 public void setSpeedDeliveryRequired(boolean speedDeliveryRequired)
9 {
10     this.speedDeliveryRequired = speedDeliveryRequired;
11 }
12
13 private boolean speedDeliveryRequired = false;
14 }
```

40.3.9.6 UISelectOne

40.3.9.6.1 UISelectOne The `UISelectOne` component facilitates the selection of one component from a choice of multiple components. It has three sub-classes:

- `HtmlSelectOneRadio` which represents a group of mutually exclusive radio buttons, and is requested by the `<h:selectOneRadio` tag,
- `HtmlSelectOneMenu` which represents a drop-down list box, and is requested by the `<h:selectOneMenu` tag, and
- `HtmlSelectOneListbox` which represents a list box allowing for single element selection which is requested by the `<h:selectOneListbox` tag,

All three component tags have the same attribute. A typical usage would be

```
1 <h:selectOneRadio id="moodSelector" value="capturePersonStateBinding.mood">
2     <f:selectItems value="#{capturePersonStateBinding.moods}"/>
3 </h:selectOneRadio>
```

where one could have the binding bean using a `Moods` enumeration:

```
1 public enum Moods {ecstatic, happy, content, irritable, sad, furious}
```


and the binding bean having a `mood` property as well as a `moods` service which returns the collection of possible moods:

```

1 @ManagedBean
2 @RequestScoped
3 public class CapturePersonStateBinding
4 {
5     public Mood getMood() {return mood;}
6
7     public void setMood(Mood mood) {this.mood = mood;}
8
9     public Mood[] getMoods() {return Mood.values();}
10
11     private Mood mood = Mood.content;
12 }

```

Note: *The same infrastructure could be used for `<h:selectOneMenu` and `<h:selectOneListbox`. They would only be rendered differently.*

Alternatively the selectable items could be obtained from a service returning a collection of objects or strings.

40.3.9.7 UISelectMany

Like `UISelectOne`, `UISelectMany` has three concrete sub-classes, `HtmlSelectManyCheckbox`, `HtmlSelectManyMenu` and `HtmlSelectManyListbox`, which are functionally equivalent, but which are differently rendered. **Note:** *It would have been nice if this was a pure styling decision supported in the style sheets.*

The corresponding component tags are `<h:selectManyCheckbox`, `<h:selectManyMenu`, and `<h:selectManyListbox` respectively. The typical usage is as follows:

```

1 <h:selectManyCheckbox id="sportTypesSelector" value="capturePersonDetailsBinding.sportTypesPlayed" >
2     <f:selectItems value="#{capturePersonDetailsBinding.sportTypes}" />
3 </h:selectManyCheckbox>

```

where the corresponding binding component (backing bean) would look something like

```

1 @ManagedBean
2 @RequestScoped
3 public class CapturePersonStateBinding
4 {
5     public List<SportType> getSportTypesPlayed() {return sportTypesPlayed;}
6
7     public void setSportTypesPlayed(List<SportType> sportTypesPlayed)
8     {
9         this.sportTypesPlayed = sportTypesPlayed;
10    }
11
12    public List<SportType> getSportTypes() {return sportServices.getSportTypes();}
13
14    @EJB
15    private SportServices sportServices;
16    private List<SportType> sportTypesPlayed = new LinkedList<SportType>();
17 }

```

40.3.10 Tables

Even though they are within separate component hierarchies, tables have a lot in common with panel grids.

```

1 <h:dataTable var="reading" value="#{weatherQueryBinding.weatherReadings}"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core">
6
7   <h:column>
8     <f:facet name="header">
9       <h:outputText value="#{msgs.dateTime}" />
10    </f:facet>
11    <h:outputText value="#{reading.dateTime}" />
12  </h:column>
13
14  <h:column>
15    <f:facet name="header">
16      <h:outputText value="#{msgs.location}" />
17    </f:facet>
18    <h:outputText value="#{reading.location}" />
19  </h:column>
20
21  <h:column>
22    <f:facet name="header">
23      <h:outputText value="#{msgs.temperature}" />
24    </f:facet>
25    <h:outputText value="#{reading.temperature}" />
26  </h:column>
27
28  <h:column>
29    <f:facet name="header">
30      <h:outputText value="#{msgs.humidity}" />
31    </f:facet>
32    <h:outputText value="#{reading.humidity}" />
33  </h:column>
34
35  <h:column>
36    <f:facet name="header">
37      <h:outputText value="#{msgs.ambiance}" />
38    </f:facet>
39    <h:outputText value="#{reading.ambiance}" />
40  </h:column>
41
42 </h:dataTable>

```

In addition to the column header facet, one can also specify table header and footer facets. Tables have the same pre-defined CSS style classes as panel grids, i.e. `headerClass`, `footerClass`, `rowClasses` and `columnClasses`

40.3.11 Messaging components

40.3.11.1 Messaging components

During the HTTP request life cycle messages are automatically created and inserted into the message queue. Additionally one may manually add messages in the binding components (backing beans).

Messages can be component specific, view specific or just general. Typically view-specific and general messages are rendered via a `<messages/>` tag whilst component specific messages are rendered via a `<message for="someComponentId"/>` tag.

These tags can be inserted anywhere in the facelet and are typically stylized via appropriate style sheet entries which match on message and messages elements.

40.4 JSF tag libraries

There is a wide variety of JSF tag libraries containing a huge amount of sophisticated user interface components including

- menus, pop-up menus and menu bars,
- calendars and schedules,
- components for file up and down load,
- tables, trees and spreadsheets,
- charts, playing media,
- bread crumbs, progress bars, and notification bars,
- tabbed panes, accordeons and carousells,
- tool tips,

and many more.

Some of the more mature JSF tag libraries include

- **PrimeFaces:** PrimeFaces is an Ajax-enabled tag library with over 100 user interface components. It is available from <http://www.primefaces.org>. They have a very nice show case of their components with source examples at <http://www.primefaces.org/showcase>.
- **Apache Trinidad:** A large component library which is part of the *Apache MyFaces* JSF implementation. It is available from <http://myfaces.apache.org/trinidad/>
- **JBoss RichFaces:** RichFaces originated from Ajax4jsf framework which was created and designed by Alexander Smirnov. The rich Ajax-enabled Java tag library available from <http://www.jboss.org/richfaces> and a component show case can be viewed at <http://www.jboss.org/richfaces/demos.html>.
- **ICE Faces:** ICE Faces is an Ajax enabled JSF component library available from <http://www.icefaces.org>. They also provide a component show case which makes it easy to see the components and example source codes for all components. The show case can be accessed at <http://component-showcase.icefaces.org/component-showcase>.
- **OpenFaces:** This tag library is available from <http://openfaces.org/>.

40.5 Custom components

JSF is a powerful component framework which makes it easy to define custom components. The simplest of these are reusable composite components which are defined as a reusable resource. For such components one only has to write an XHTML view – no Java coding is required.

In addition to composite components, JSF also supports the definition of published components which are published in a tag library. The work required to develop these published components is considerably more. These components are typically meant for very wide-spread reuse, justifying the additional effort required to develop them. Often these components are given Ajax support.

40.5.1 Custom composite components

Custom composite components are very easy to develop and reuse. They are composed as a combination of

- more primitive components, and
- resources like
 - images,
 - styling, and
 - JavaScript/Ajax resources.

Composite components are themselves stored as reusable resources and are easily embedded in multiple pages.

40.5.1.1 Developing composite components

In order to develop a composite component one needs to

1. Develop an XHTML facelet which uses the predefined JFS `composite` component tags for defining
 - a component interface (the various touch points to the component),
 - and the component implementation which defines the visual components and makes use of abstracted linkage to an ultimate backing bean as specified in the interface.
2. Persist the component as a resource in the resource library.

The composite component itself is a XHTML component with the following structure:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5 xmlns:h="http://java.sun.com/jsf/html"
6 xmlns:composite="http://java.sun.com/jsf/composite">
7 <head>
8 <title>A collapsible panel composite component</title>
9 </head>
10 <body>
11 <composite:interface>
12 <!-- DEFINE THE COMPONENT INTERFACE HERE -->
13 </composite:interface>
14 <composite:implementation>
15 <!-- DEFINE THE COMPONENT IMPLEMENTATION HERE -->
16 </composite:implementation>
17 </body>
18 </html>

```

The interface of a composite component is assembled from

- *component attributes*, which typically represent modifiable aspects of the component,
- references to *valueHolder* s and *editableValueHolders* which provide handles to output and/or input fields which can be bound to backing bean variables,
- component *facets* (a facet represents a named section within a container, and the name for the component facet of a reusable composite component can be specified when using the component), and

- *action sources* which are typically processed in associated binding or backing beans.

The component interface specification is of the following form:

The attributes will need to be provided by getters and setters matching the attribute names as per JavaBeans specification. You can optionally specify

- whether the attribute is required (i.e. must be provided) or not,
- a default value for the attribute, and
- a method signature specifying the data type and hence the default converter to be used for the attribute.

Note: *JSF will automatically add any attributes used in the implementation to the component interface. It is, however, strongly recommended that you do list all component attributes and action events in the interface definition for the component as the latter represents a contract for the component.*

The component view is specified in the `composite:implementation` tag. It is assembled from

- other, more basic components,
- resources like
 - images,
 - styling, and
 - JavaScript resources including AJAX bindings.

The view definition is thus a standard view definition with the exception that the binding is done to a component class generated by JSF for the composite component.

40.5.1.1.1 User Provided child components A composite component may provide component users the option to insert child components. This is done using the `<composite:insertChildren/>` tag. For example, the following code listing illustrates how one can specify child components for a composite component whilst allowing the component users to insert further children:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional/EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://java.sun.com/jsf/html"
6       xmlns:composite="http://java.sun.com/jsf/composite">
7   <head>
8     <title>A collapsible panel composite component</title>
9   </head>
10  <body>
11    <composite:interface>
12      <!-- DEFINE THE COMPONENT INTERFACE HERE -->
13    </composite:interface>
14    <composite:implementation>
15      <!-- SOME COMPONENT ELEMENTS -->
16
17      <composite:insertChildren/> <!-- Allow component user to insert further child components -->
18
19      <!-- SOME FURTHER COMPONENT ELEMENTS -->
20    </composite:implementation>
21  </body>
22 </html>

```

!-

40.5.1.1.2 Inserting facets into the view A facet represents a named section in a container component which have one child component. Facets are used to render components in a particular way.

The rendering of facets is requested via the `composite:renderFacet` tag:

```
1 <composite:renderFacet name="header"/>
```

```
-;
```

40.5.1.1.3 Generated component class JSF generates a component class which is accessible through the `cc` reference. In its core it contains an attributes map which contains component attributes and a model map which, in turn, contains the action event handlers.

- The component attributes embedded in an attribute map are accessed via

```
1 cc.attrs.attributeName
```

- action event handlers are coupled to action event sources via the model map contained the attributes map of the composite component class:

```
1 <actionComponent action="#{cc.attrs.model.actionEvent1}"/>
```

40.5.1.2 Using composite components

Composite components are stored in a library in the web application's resources folder. The component URI is assembled from the concatenation of

- the JSF composite URI, `http://java.sun.com/jsf/composite`,
- the name of the resource library (i.e. relative to the web-app resources, the path to the folder containing the composite component), and
- the component name which is the XHTML file name without the `.xhtml`.

One imports the name space for the component library into a namespace prefix and accesses the component via the namespace variable:

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4   ...
5   xmlns:comp="http://java.sun.com/jsf/composite/components">
6
7   <comp:myCompositeComponent model="#{myBackingBean}">
8
9       <!-- Child components, component variables, component facets ... -->
10
11   </comp:myCompositeComponent>
12 </html>
```

40.5.1.3 Example: Collapsible panel

40.5.1.3.1 Example: Collapsible panel This example is a composite component for a collapsible panel. The user inserts child components which are either hidden or shown.

40.5.1.3.1.1 The component specification The component specification was saved in a components resource library, i.e. in `webapp/resources/components`. It looks as follows:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://java.sun.com/jsf/html"
6       xmlns:composite="http://java.sun.com/jsf/composite">
7 <head>
8   <title>Collapsible panel</title>
9 </head>
10 <body>
11 <composite:interface>
12   <composite:attribute name="model" required="true">
13     <composite:attribute name="collapsed" required="true"/>
14     <composite:attribute name="toggle"
15       method-signature="java.lang.String toggle()" required="true"/>
16   </composite:attribute>
17   <composite:actionSource name="toggle"/>
18   <composite:facet name="header"/>
19 </composite:interface>
20 <composite:implementation>
21   <h:panelGroup layout="block"
22     styleClass="collapsiblePanel-header">
23     <h:commandButton id="toggle"
24       action="#{cc.attrs.model.toggle}"
25       styleClass="collapsiblePanel-img"
26       image="#{resource[cc.attrs.model.collapsed
27         ? 'components:plus.png'
28         : 'components:minus.png']}" />
29     <composite:renderFacet name="header"/>
30   </h:panelGroup>
31   <h:panelGroup layout="block"
32     rendered="#{!cc.attrs.model.collapsed}">
33     <composite:insertChildren/>
34   </h:panelGroup>
35 </composite:implementation>
36 </body>
37 </html>

```

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/
2   DTD/xhtml1-transitional.dtd">
3 <fieldset xmlns="http://www.w3.org/1999/xhtml"
4   xmlns:ui="http://java.sun.com/jsf/facelets"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:p="http://primefaces.org/ui"
8   xmlns:comp="http://java.sun.com/jsf/composite/components">
9 <legend>WeatherReading</legend>
10
11 <!-- The following line forces the initialization of the backing bean for this aggregate view. -->
12 <h:inputHidden value="#{weatherReadingBackingBean.dummy}" />
13
14 <ui:include src="/faces/locations/locationSelector.xhtml" />
15 <ui:include src="/faces/dateTime/dateTimePanel.xhtml" />
16
17 <comp:collapsiblePanel model="#{weatherReadingBackingBean}">
18   <f:facet name="header">
19     <h3><h:outputText value="#{msgs.weatherMeasurements}" /></h3>
20   </f:facet>
21
22   <h:panelGrid columns="2">
23
24     <h:outputLabel id="temperatureLabel" for="temperatureField"
25       value="#{msgs.temperature}" />
26     <h:inputText id="temperatureField"

```

```

27     value="#{weatherReadingBackingBean.weatherReading.temperature}" />
28
29     <h:outputLabel id="humidityLabel" for="humidityField"
30     value="#{msgs.humidity}" />
31     <h:inputText id="humidityField"
32     value="#{weatherReadingBackingBean.weatherReading.humidity}" />
33
34     <h:outputLabel id="ambianceLabel" for="ambianceField"
35     value="#{msgs.ambiance}" />
36     <h:selectOneMenu id="ambianceField"
37     value="#{weatherReadingBackingBean.weatherReading.ambiance}" >
38     <f:selectItems
39     value="#{weatherReadingBackingBean.ambianceValues}" />
40     </h:selectOneMenu>
41
42     </h:panelGrid>
43 </comp:collapsiblePanel>
44
45 </fieldset>

```

```

1 package za.co.solms.weather;
2
3 import java.io.Serializable;
4
5 import javax.faces.bean.ManagedBean;
6 import javax.faces.bean.ManagedProperty;
7 import javax.faces.bean.RequestScoped;
8 import javax.faces.bean.ViewScoped;
9
10 import za.co.solms.dateTime.DateTimeBackingBean;
11 import za.co.solms.locations.LocationSelectorBackingBean;
12
13 @ManagedBean
14 @RequestScoped
15 public class WeatherReadingBackingBean implements Serializable
16 {
17     public WeatherReadingBackingBean() {}
18
19     public WeatherReading getWeatherReading()
20     {
21         weatherReading.setLocation(locationSelectorBackingBean.getLocation());
22         weatherReading.setDateTime(dateTimeBackingBean.getDateTime());
23
24         return weatherReading;
25     }
26
27     public void setWeatherReading(WeatherReading weatherReading)
28     {
29         this.weatherReading = weatherReading;
30         if (weatherReading.getLocation() != null)
31             locationSelectorBackingBean.setLocationName(weatherReading.getLocation().getName());
32         dateTimeBackingBean.setDateTime(weatherReading.getDateTime());
33     }
34
35     public LocationSelectorBackingBean getLocationSelectorBackingBean()
36     {
37         return locationSelectorBackingBean;
38     }
39
40     public void setLocationSelectorBackingBean(
41         LocationSelectorBackingBean locationSelectorBackingBean)
42     {
43         this.locationSelectorBackingBean = locationSelectorBackingBean;
44     }
45
46     public DateTimeBackingBean getDateTimeBackingBean()
47     {
48         return dateTimeBackingBean;
49     }
50 }

```



```

49  }
50
51  public void setDateTimeBackingBean(DateTimeBackingBean dateTimeBackingBean)
52  {
53      this.dateTimeBackingBean = dateTimeBackingBean;
54  }
55
56  public boolean isDummy() {return dummy;}
57
58  public void setDummy(boolean dummy)
59  {
60      this.dummy = dummy;
61  }
62
63  public Ambiance[] getAmbianceValues()
64  {
65      return Ambiance.values();
66  }
67
68  public boolean isCollapsed()
69  {
70      return collapsed;
71  }
72
73  public void setCollapsed(boolean collapsed)
74  {
75      this.collapsed = collapsed;
76  }
77
78  public String toggle()
79  {
80      collapsed = !collapsed;
81      return null;
82  }
83
84  private boolean collapsed;
85  private boolean dummy;
86
87  private WeatherReading weatherReading = new WeatherReading();
88
89  @ManagedProperty(value="#{locationSelectorBackingBean}")
90  private LocationSelectorBackingBean locationSelectorBackingBean;
91
92  @ManagedProperty(value="#{dateTimeBackingBean}")
93  private DateTimeBackingBean dateTimeBackingBean;
94  }

```

40.5.2 Published custom components

Published components are either composite or basic components for which the component, renderer and tag handler classes are specified and which are registered in a tag library in order to facilitate general reuse.

To develop a new custom component which is available through a tag library, we need to

1. develop a component class,
2. develop a renderer class,
3. define tag,
4. write tag handler class,
5. register the component, the renderer and the tag in a tag library
6. import the tag library into page, and finally

7. use the component by embedding the tag within page.

40.5.2.1 Defining the component view

The component view is similar to that of the simpler composite components. It is provided as an XHTML facelet which defines the component interface and implementation.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://java.sun.com/jsf/html"
6       xmlns:composite="http://java.sun.com/jsf/composite">
7   <head>
8     <title>An input field with increment/decrement buttons</title>
9   </head>
10  <body>
11    <composite:interface>
12      <composite:attribute name="value" required="true" />
13      <composite:attribute name="increment" required="false" default="1" />
14      <composite:editableValueHolder name="input" />
15    </composite:interface>
16    <composite:implementation>
17      <h:outputStylesheet library="components" name="spinButtonField.css" />
18      <h:outputScript library="components"
19        name="spinButtonField.js" target="head" />
20      <h:panelGroup>
21        <h:inputText id="input" value="#{cc.attrs.value}"
22          styleClass="spinButtonField-input" />
23        <h:panelGroup id="buttons"
24          styleClass="spinButtonField-buttons">
25          <h:graphicImage library="jsf.components"
26            name="spinUp.png"
27            styleClass="spinButtonField-button"
28            onclick="return incrementField('#{cc.clientId}:input', #{cc.attrs.increment});" />
29          <h:graphicImage library="jsf.components"
30            name="spinDown.png"
31            styleClass="spinButtonField-button"
32            onclick="return incrementField('#{cc.clientId}:input', #{-cc.attrs.increment});" />
33        </h:panelGroup>
34      </h:panelGroup>
35    </composite:implementation>
36  </body>
37 </html>

```

40.5.2.2 The component class

For the component class one needs to specify

- the component family,
- the component type,
- the renderer type, and
- any component attributes.

40.5.2.2.1 The component family The component family is used in conjunction with the renderer type to select a suitable renderer for the component. It also allows for the case where a single renderer is used for the entire component family.

There are a range of standard component families for input, output, button, command, item selection components and so on. For example, the specializations of `javax.faces.component.UIInput` belong by default to the `javax.faces.UIInput` component family.

In order to change the component family one overrides the

```
1 String getFamily()
2 {
3     return "familyName"
4 }
```

service.

40.5.2.2.2 The component type The component type is a *unique identifier* for a component - there may not exist another component with the same component type. It is used during the *registration* of a component, for example, when the component is defined within a *tag library*.

The component type can also be used when one wants to *abstract the component creation from a particular implementation class for that component*. In such cases one generally creates component via the `Application` class obtained from the `FacesContext`:

```
1 String componentType = "some.componentType";
2
3 FacesContext fc = FacesContext.getCurrentInstance();
4
5 panel.getChildren().add
6     (fc.getCurrentInstance().getApplication().createComponent(componentType));
```

40.5.2.2.3 The renderer type The renderer type identifies a component renderer and enables one to abstract from a concrete renderer. If the renderer type is set to `null`, the component renders itself.

40.5.2.2.4 Selecting component base class One can write a component from scratch by deriving one's class from the abstract `javax.faces.component.UIComponentClass`. In most cases it is, however, beneficial for both, conceptual and code reuse reasons to subclass one of the concrete component classes. In general use as base class

- `UIOutput` if your class represents some data which should not be modified by the user,
- `UIInput` if your components represents some data which the user can potentially modify
Note: *UIInput* is a sub-class of *UIOutput*.
- `UISelectOne` (or `UISelectMany`) if the user can select one (or more) value(s) from a collection of values,
- `UICommand` if your component is meant to trigger an action event,
- `UIPanel` if your component is meant to host other components,
- `UIMessage` (or `UIMessages`) if your component is meant to provide message(s) to the user,

40.5.2.2.5 The component-state map Component properties are maintained within a properties map which is provided by the `UIComponent` base class. The component class needs to specify a unique key for each property and must provide setters and getters for each property:

```

1 package za.co.solms.jsf.components;
2
3 import javax.faces.component.UIInput;
4 import javax.faces.component.FacesComponent;
5 import javax.faces.context.FacesContext;
6
7 @FacesComponent("za.co.solms.SpinButtonField")
8 public class SpinButtonField extends UIInput
9 {
10     public static final String COMPONENT_TYPE = "za.co.solms.SpinButtonField";
11
12     enum PropertyKeys
13     {
14         increment
15     }
16     public SpinButtonField()
17     {
18         setRendererType(COMPONENT_TYPE);
19     }
20     public int getIncrement()
21     {
22         return (Integer)getStateHelper().eval(PropertyKeys.increment, 1);
23     }
24     public void setIncrement(int increment)
25     {
26         getStateHelper().put(PropertyKeys.increment, increment);
27     }
28 }

```

40.5.2.2.6 Example: SpinButtonField

40.5.2.3 Component renderer

The component renderer does the population of the view and the extraction of the information from the view.

```

1 package za.co.solms.jsf.components;
2
3 import javax.el.ValueExpression;
4 import javax.faces.application.Resource;
5 import javax.faces.application.ResourceDependencies;
6 import javax.faces.application.ResourceDependency;
7 import javax.faces.application.ResourceHandler;
8 import javax.faces.component.UIComponent;
9 import javax.faces.component.UIInput;
10 import javax.faces.context.FacesContext;
11 import javax.faces.context.ResponseWriter;
12 import javax.faces.convert.Converter;
13 import javax.faces.convert.ConverterException;
14 import javax.faces.render.Renderer;
15 import javax.faces.render.FacesRenderer;
16 import java.io.IOException;
17 import java.text.MessageFormat;
18 import java.util.Map;
19
20 @ResourceDependencies({
21     @ResourceDependency(library = "components",
22         name = "spinButtonField.js", target = "head"),
23     @ResourceDependency(library = "components",
24         name = "components.css")})
25 )
26 @FacesRenderer(componentFamily = "javax.faces.Input",

```

```

27     rendererType = "za.co.solms.SpinButtonField")
28     public class SpinButtonFieldRenderer extends Renderer
29     {
30
31         @Override
32         public void decode(FacesContext ctx, UIComponent component)
33         {
34             Map<String, String> params
35                 = ctx.getExternalContext().getRequestParameterMap();
36             String clientId = component.getClientId();
37             String value = params.get(clientId);
38             ((UIInput)component).setSubmittedValue(value);
39         }
40
41         @Override
42         public Object getConvertedValue(FacesContext ctx,
43             UIComponent component, Object submittedValue)
44             throws ConverterException
45         {
46             Converter converter = getConverter(ctx, component);
47             if (converter != null )
48             {
49                 return converter.getAsObject(ctx, component,
50                     (String) submittedValue);
51             }
52             else
53             {
54                 return submittedValue;
55             }
56         }
57
58         @Override
59         public void encodeBegin(FacesContext context, UIComponent component)
60             throws IOException
61         {
62             SpinButtonField spinButtonField = (SpinButtonField)component;
63             String clientId = spinButtonField.getClientId();
64
65             encodeInput(context, spinButtonField, clientId);
66             encodeButtons(context, spinButtonField, clientId);
67         }
68
69         private void encodeInput(FacesContext context,
70             SpinButtonField spinButtonField, String clientId) throws IOException
71         {
72             ResponseWriter writer = context.getResponseWriter();
73             writer.startElement("input", spinButtonField);
74             writer.writeAttribute("id", clientId, null);
75             writer.writeAttribute("name", clientId, null);
76             Object value = getValue(context, spinButtonField);
77             if (value != null) {
78                 writer.writeAttribute("value", value.toString(), null);
79             }
80             writer.writeAttribute("class", "spinButtonField-input", null);
81             writer.endElement("input");
82         }
83
84         private void encodeButtons(FacesContext ctx,
85             SpinButtonField spinButtonField, String clientId) throws IOException
86         {
87             ResponseWriter writer = ctx.getResponseWriter();
88             ResourceHandler resourceHandler
89                 = ctx.getApplication().getResourceHandler();
90             MessageFormat onclick
91                 = new MessageFormat("return changeNumber('{0}', {1});");
92
93             writer.startElement("span", spinButtonField);
94             writer.writeAttribute("class", "spinButtonField-buttons", null);
95
96             Resource spinUpResource = resourceHandler.createResource("spin-up.png",
97                 "components");
98             String onclickUp = onclick.format(new Object[]{clientId,
99                 spinButtonField.getIncrement()});

```

```

100     encodeSpinButtonField(spinButtonField, writer,
101         spinUpResource, onclickUp);
102
103     Resource spinDownResource
104         = resourceHandler.createResource("spin-down.png", "components");
105     String onclickDown = onclick.format(
106         new Object[]{clientId, -spinButtonField.getIncrement()});
107     encodeSpinButtonField(spinButtonField, writer,
108         spinDownResource, onclickDown);
109
110     writer.endElement("span");
111 }
112
113 private void encodeSpinButtonField(SpinButtonField spinButtonField,
114     ResponseWriter writer, Resource resource, String onclick) throws IOException
115 {
116     writer.startElement("img", spinButtonField);
117     writer.writeAttribute("class", "spinButtonField-button", null);
118     writer.writeAttribute("src", resource.getRequestPath(), null);
119     writer.writeAttribute("onclick", onclick, null);
120     writer.endElement("img");
121 }
122
123 private Object getValue(FacesContext ctx, SpinButtonField spinButtonField)
124 {
125     Object submittedValue = spinButtonField.getSubmittedValue();
126     if (submittedValue != null)
127     {
128         return submittedValue;
129     }
130     Object value = spinButtonField.getValue();
131     Converter converter = getConverter(ctx, spinButtonField);
132     if (converter != null)
133     {
134         return converter.getAsString(ctx, spinButtonField, value);
135     } else if (value != null)
136     {
137         return value.toString();
138     } else {
139         return "";
140     }
141 }
142
143 private Converter getConverter(FacesContext ctx,
144     UIComponent component)
145 {
146     Converter converter = ((UIInput)component).getConverter();
147     if (converter != null) return converter;
148
149     ValueExpression exp = component.getValueExpression("value");
150     if (exp == null) return null;
151
152     Class valueType = exp.getType(ctx.getELContext());
153     if (valueType == null) return null;
154
155     return ctx.getApplication().createConverter(valueType);
156 }
157
158 }

```

40.5.2.4 Component tag handler

The component tag handler defines and binds the component attributes:

```

1 package za.co.solms.jsf.components;
2
3 import javax.faces.view.facelets.*;
4
5 public class SpinButtonFieldTagHandler extends ComponentHandler {
6

```

```

7     private static final String ATTR_INCREMENT = "increment";
8
9     private TagAttribute increment;
10
11     public SpinButtonFieldTagHandler(ComponentConfig config) {
12         super(config);
13         this.increment = getRequiredAttribute(ATTR_INCREMENT);
14     }
15
16 }

```

40.5.2.5 The tag library descriptor

The tag library descriptor specifies

- a name space for the tag library,
- the different tags and for each tag, the tag components.

```

1 <?xml version="1.0"?>
2 <facelet-taglib version="2.0"
3     xmlns="http://java.sun.com/xml/ns/javaee"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6         http://java.sun.com/xml/ns/javaee/web-facelettaglibrary_2.0.xsd" >
7
8     <namespace>http://za.co.solms/jsf/components</namespace>
9
10    <tag>
11        <tag-name>spinButtonField</tag-name>
12        <component>
13            <component-type>
14                za.co.solms.SpinButtonField
15            </component-type>
16            <renderer-type>
17                za.co.solms.SpinButtonField
18            </renderer-type>
19            <handler-class>
20                za.co.solms.jsf.components.SpinButtonFieldTagHandler
21            </handler-class>
22        </component>
23    </tag>
24
25 </facelet-taglib>

```

40.5.2.6 Tag library registration

The tag library is registered in the `web.xml` file:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <web-app version="2.5"
4     xmlns="http://java.sun.com/xml/ns/javaee"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
7         http://java.sun.com/xml/ns/javaee/web-app_2.5.xsd" >
8
9     <display-name>WeatherBuro</display-name>
10
11     <servlet>
12         <servlet-name>Faces Servlet</servlet-name>
13         <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
14     </servlet>
15

```

```

16 <servlet-mapping>
17 <servlet-name>Faces Servlet</servlet-name>
18 <url-pattern>*.jsf</url-pattern>
19 </servlet-mapping>
20
21 <welcome-file-list>
22 <welcome-file>index.html</welcome-file>
23 </welcome-file-list>
24
25 <context-param>
26 <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
27 <param-value>.xhtml</param-value>
28 </context-param>
29
30
31 <context-param>
32 <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
33 <param-value>/WEB-INF/solms.taglib.xml</param-value>
34 </context-param>
35
36 </web-app>

```

40.5.2.7 Using a component from a custom tag library

In order to use a component from a custom tag library you have to

- within a facelet using custom components from the tag library, import the tag library name space into a namespace prefix variable.
- use the tag specifying attributes, action events, facets and child components, and
- provide the required backing in a backing bean.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <fieldset xmlns="http://www.w3.org/1999/xhtml"
4   xmlns:ui="http://java.sun.com/jsf/facelets"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:sl="http://za.co.solms/jsf/components"
8   xmlns:p="http://primefaces.org/ui">
9
10   <legend>Date/Time</legend>
11
12   <h:panelGrid columns="2">
13     <h:outputLabel id="dateLabel" for="dateField" value="#{msgs.date}"/>
14     <p:calendar id="dateField" value="#{dateTimeBackingBean.date}"
15       title="#{msgs.date}"/>
16
17     <h:outputLabel id="hourLabel" for="hourField" value="#{msgs.hour}"/>
18     <sl:spinButtonField id="hourField" value="#{dateTimeBackingBean.hour}"
19       increment="1"/>
20
21     <h:outputLabel id="minuteLabel" for="minuteField"
22       value="#{msgs.minute}"/>
23     <sl:spinButtonField id="minuteField"
24       value="#{dateTimeBackingBean.minute}"
25       increment="15"/>
26   </h:panelGrid>
27 </fieldset>

```



```
1  /**
2  *
3  */
4  package za.co.solms.dateTime;
5
6  import java.io.Serializable;
7  import java.util.Calendar;
8  import java.util.Date;
9  import java.util.GregorianCalendar;
10
11  import javax.faces.bean.ManagedBean;
12  import javax.faces.bean.RequestScoped;
13  import javax.faces.bean.ViewScoped;
14
15  /**
16   * @author fritz@solms.co.za
17   *
18   */
19  @ManagedBean
20  @RequestScoped
21  public class DateTimeBackingBean implements Serializable
22  {
23      public DateTimeBackingBean(){}
24
25      public Date getDateTime()
26      {
27          GregorianCalendar cal = new GregorianCalendar();
28          cal.setTime(date);
29          cal.set(Calendar.HOUR, hour);
30          cal.set(Calendar.MINUTE, minute);
31
32          return cal.getTime();
33      }
34
35      public void setDateTime(Date dateTime)
36      {
37          GregorianCalendar cal = new GregorianCalendar();
38          cal.setTime(dateTime);
39          hour=cal.get(Calendar.HOUR);
40          minute = cal.get(Calendar.MINUTE);
41          cal.set(Calendar.HOUR, 0);
42          cal.set(Calendar.MINUTE, 0);
43          cal.set(Calendar.SECOND, 0);
44          cal.set(Calendar.MILLISECOND, 0);
45          date = cal.getTime();
46      }
47
48      public int getMinute()
49      {
50          return minute;
51      }
52
53      public void setMinute(int minute)
54      {
55          this.minute = minute;
56      }
57
58      public int getHour()
59      {
60          return hour;
61      }
62
63      public void setHour(int hour)
64      {
65          this.hour = hour;
66      }
67
68      public Date getDate()
69      {
70          return date;
71      }
72
73      public void setDate(Date date)
```

```
74 {  
75     this.date = date;  
76 }  
77  
78 private int minute, hour;  
79 private Date date = new Date();  
80  
81 private static final long serialVersionUID = 1L;  
82 }
```

Chapter 41

Converters

The user enters text on a keyboard (or potentially other signals through other devices). One needs to convert the set of characters (or some signal) to the internal data type which is being captured. To this end the JSF framework supports

- a range of standard converters for the primitive data types and some basic classes,
- the ability to define custom converters.

Converters are responsible for the conversion in both directions, i.e.

- to convert the user input to the internal data types, and
- to convert the internal data types to a text (or other) representation used to populate the user interface components.

41.1 Standard converters

JSF has standard converters for all *primitive data types* as well as the `BigInteger` and `BigDecimal` classes. All of these are named `<DataTypeName>Converter` and are used transparently (behind the scenes).

In addition to the above JSF provides `convertNumber` and `convertDateTime` as flexible number and date/time converters.

41.1.1 The number converter

Number converters provide a flexible converter for converting numbers including percentages, currencies and numbers with thousands separators to floating point numbers.

For example, if one wants to specify that the ticket price for a show is in South African Rand, one could specify the following number converter

```
1 <h:inputText value="#{bookTicketsBinding.presentation.price}">
2   <f:convertNumber currencySymbol="R" type="currency" groupingsUsed="#{true}" maxFractionDigits="2" />
3 </h:inputText>
```

The `<f:convertNumber>` tag has the following potential attributes:

- **type:** The default is `number`, but it can be changed to either `percentage` or `currency`.

- **minIntegerDigits** and **maxIntegerDigits**: The minimum and maximum number of digits before the decimal point.
- **minFractionDigits** and **maxFractionDigits**: The minimum and maximum digits after the decimal point.
- **currenctSymbol** and **currencyCode**: The symbol and code used for the currency like \$ or R and USD or ZAR.
- **groupingUsed**: This boolean attribute specifying whether grouping symbols like commas or spaces are used - the default value for this attribute is **true**.
- **locale**: The locale to be used for the number pattern. It can be specified as a locale string like **en-za** or **fr-ca** or via an expression which resolves a locale object, e.g. **#session.locale**.
- **pattern**: Instead of a combination of the above attributes, one can also specify the number format via a number format pattern. For example **pattern="#.##0.00"**

41.1.2 Date/Time conversion

Date time conversion can be complex due to the wide range of date/time formats, different use of calendars, day time saving and many other factors. The `<f:convertDateTime>` converter takes the following attributes:

- **type**: This attribute specifies whether the date, time or both are specified by the user.
- **dateStyle**: This attribute takes the values **short**, **medium** (the default), **long** and **full**. It can only be specified if the type is either **date** or **both**.
- **timeStyle**: Like **dateStyle**, this attribute takes the values **short**, **medium** (the default), **long** and **full**. It can only be specified if the type is either **time** or **both**.
- **timeZone**: The time zone which should be used for the conversion.
- **locale**:
- **pattern**: Instead of a combination of **type**, **dateStyle** and **timeStyle**, one can specify the actual date/time pattern (e.g. **pattern="dd/MM/yyyy"**)

41.2 Custom converters

In addition to supporting the range of standard converters, JSF also allows one to plug in custom converters.

Custom converters need to implement the **Converter** interface which requires one to implement the conversion methods for the conversions to and from text:

Once the converter has been defined, it can be assigned as the converter for input and output fields:

```

1 <h:inputText value="myBindingBean.myAttribute1">
2   <f:converter converterId="myConverter" />
3 </h:inputText>
4
5 <h:outputText value="myBindingBean.myAttribute2">
6   <f:converter converterId="myConverter" />
7 </h:outputText>

```

41.3 Examples

In this simple example we capture book details including title, full author names, publication date and price and we use

- a custom converter to map a single String onto a Names class containing first, middle and last names,
- a date/time converter to capture and populate the publication date component, and
- a number converter to map between the user view of a price and the internal double precision floating point number.

41.3.1 Names class

```
1 package za.co.solms.publishing.books.model;
2
3 import sun.java2d.Surface;
4
5 public class Names
6 {
7     public Names(String firstName, String middleNames, String lastName)
8     {
9         setFirstName(firstName);
10        setMiddleNames(middleNames);
11        setLastName(lastName);
12    }
13
14    public String getFirstName()
15    {
16        return firstName;
17    }
18
19    public void setFirstName(String firstName)
20    {
21        this.firstName = firstName;
22    }
23
24    public String getMiddleNames()
25    {
26        return middleNames;
27    }
28
29    public void setMiddleNames(String middleNames)
30    {
31        this.middleNames = middleNames;
32    }
33
34    public String getLastName()
35    {
36        return lastName;
37    }
38
39    public void setLastName(String lastName)
40    {
41        this.lastName = lastName;
42    }
43
44    public boolean equals(Object o)
45    {
46        try
47        {
48            Names arg = (Names)o;
49            return firstName.equals(arg.firstName) &&
50                middleNames.equals(arg.middleNames) &
51                lastName.equals(arg.lastName);
```

```
52     }
53     catch (ClassCastException e)
54     {
55         return false;
56     }
57 }
58
59 public int hashCode()
60 {
61     return firstName.hashCode() + middleNames.hashCode() + lastName.hashCode();
62 }
63
64 public String toString()
65 {
66     return firstName + " " + middleNames + " " + lastName;
67 }
68
69 private String firstName, middleNames, lastName;
70 }
```

41.3.2 Book details class

```
1 package za.co.solms.publishing.books.model;
2
3 import java.util.Date;
4
5 public class BookDetails
6 {
7     public BookDetails(String title, Names authorName, Date publicationDate, double price)
8     {
9         setTitle(title);
10        setAuthorName(authorName);
11        setPublicationDate(publicationDate);
12        setPrice(price);
13    }
14
15    public String getTitle()
16    {
17        return title;
18    }
19
20    public void setTitle(String title)
21    {
22        this.title = title;
23    }
24
25    public Names getAuthorName()
26    {
27        return authorName;
28    }
29    public void setAuthorName(Names authorName)
30    {
31        this.authorName = authorName;
32    }
33    public Date getPublicationDate()
34    {
35        return publicationDate;
36    }
37    public void setPublicationDate(Date publicationDate)
38    {
39        this.publicationDate = publicationDate;
40    }
41    public double getPrice()
42    {
43        return price;
44    }
45    public void setPrice(double price)
46    {
47        this.price = price;
48    }
49 }
```

```

50 public String toString() {return title;}
51
52 public boolean equals(Object o)
53 {
54     try
55     {
56         BookDetails arg = (BookDetails)o;
57         return title.equals(arg.title) && authorName.equals(arg.authorName)
58             && publicationDate.equals(arg.publicationDate)
59             && price == arg.price;
60     }
61     catch (ClassCastException e)
62     {
63         return false;
64     }
65 }
66
67 public int hashCode()
68 {
69     return title.hashCode() + authorName.hashCode();
70 }
71
72 private String title;
73 private Names authorName;
74 private Date publicationDate;
75 private double price;
76 }

```

41.3.3 FullNamesConverter

```

1 package za.co.solms.publishing.books.ui.web;
2
3 import java.io.Serializable;
4 import java.util.StringTokenizer;
5
6 import javax.faces.component.UIComponent;
7 import javax.faces.context.FacesContext;
8 import javax.faces.convert.Converter;
9 import javax.faces.convert.FacesConverter;
10
11 import za.co.solms.publishing.books.model.Names;
12
13 @FacesConverter(value="za.co.solms.fullNamesConverter")
14 public class FullNamesConverter implements Converter, Serializable
15 {
16
17     @Override
18     public Object getAsObject(FacesContext ctx, UIComponent cmp, String value)
19     {
20         String firstName="", middleNames="", lastName="";
21
22         StringTokenizer tokenizer = new StringTokenizer(value);
23
24         int numNames = tokenizer.countTokens();
25         if (numNames > 0)
26             firstName = tokenizer.nextToken(" ");
27
28         StringBuffer midNames = new StringBuffer();
29         for (int numName=1; numName<numNames-1; ++numName)
30         {
31             if (numName>1)
32                 midNames.append(" ");
33             midNames.append(tokenizer.nextToken());
34         }
35         middleNames = midNames.toString();
36
37         if (numNames > 1)
38             lastName = tokenizer.nextToken();
39
40         return new Names(firstName, middleNames, lastName);
41     }

```

```

42
43     @Override
44     public String getAsString(FacesContext ctx, UIComponent cmp, Object value)
45     {
46         Names names = (Names)value;
47         return names.getFirstName() + " " + names.getMiddleNames() + " " + names.getLastName();
48     }
49
50 }

```

41.3.4 CaptureBookDetailsBinding

```

1  package za.co.solms.publishing.books.ui.web;
2
3  import java.io.Serializable;
4  import java.util.Date;
5
6  import javax.faces.bean.ManagedBean;
7  import javax.faces.bean.SessionScoped;
8
9  import za.co.solms.publishing.books.model.BookDetails;
10 import za.co.solms.publishing.books.model.Names;
11
12 @ManagedBean
13 @SessionScoped
14 public class CaptureBookDetailsBinding implements Serializable
15 {
16     public CaptureBookDetailsBinding() {}
17
18     public String saveBookDetails()
19     {
20         return "bookDetailsSavedConfirmation";
21     }
22
23     public BookDetails getBookDetails()
24     {
25         return bookDetails;
26     }
27
28     public void setBookDetails(BookDetails bookDetails)
29     {
30         this.bookDetails = bookDetails;
31     }
32
33     private BookDetails bookDetails
34     = new BookDetails("", new Names("", "", ""), new Date(), 0.00);
35 }

```

41.3.5 captureBookDetails

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html
3      PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5  <html xmlns="http://www.w3.org/1999/xhtml"
6        xmlns:f="http://java.sun.com/jsf/core"
7        xmlns:h="http://java.sun.com/jsf/html">
8  <head>
9  <title>Capture book details</title>
10 </head>
11 <body>
12 <h1><h:outputText id="header1" value="Capture Book Details"/></h1>
13
14 <h:messages/>
15
16 <h:form id="form">
17 <h:panelGrid id="grid" columns="3">
18 <h:outputLabel id="bookTitleLabel" value="Book Title" for="bookTitleFieldField"/>

```



```

19      <h:inputText id="bookTitleField" required="true" value="#{captureBookDetailsBinding.
20          bookDetails.title}"/>
21      <h:message for="bookTitleField"/>
22      <h:outputLabel id="fullAuthorNameLabel" value="Full Author Name:" for="fullAuthorNameField"
23          />
24      <h:inputText id="fullAuthorNameField" value="#{captureBookDetailsBinding.bookDetails.
25          authorName}"/>
26      <f:converter converterId="za.co.solms.fullNamesConverter" />
27      </h:inputText>
28      <h:message for="fullAuthorNameField"/>
29      <h:outputLabel id="publicationDateLabel" value="Publication Date:" for="publicationDateField"
30          />
31      <h:inputText id="publicationDateField" value="#{captureBookDetailsBinding.bookDetails.
32          publicationDate}"/>
33      <f:convertDateTime pattern="yyyy-MM-dd" />
34      </h:inputText>
35      <h:message for="publicationDateField"/>
36      <h:outputLabel id="priceLabel" value="Price:" for="priceField"/>
37      <h:inputText id="priceField" value="#{captureBookDetailsBinding.bookDetails.price}"/>
38      <f:convertNumber type="currency" currencySymbol="R" />
39      </h:inputText>
40      <h:message for="priceField"/>
41      </h:panelGrid>
42      <h:commandButton id="save" action="#{captureBookDetailsBinding.saveBookDetails}" value="save" />
43  </h:form>
44  </body>
45  </html>

```

41.3.6 bookDetailsSavedConfirmation

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html
3      PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5  <html xmlns="http://www.w3.org/1999/xhtml"
6      xmlns:f="http://java.sun.com/jsf/core"
7      xmlns:h="http://java.sun.com/jsf/html">
8  <head>
9      <title>Book details saved confirmation</title>
10 </head>
11 <body>
12     <h1><h:outputText value="Book details saved confirmation"/></h1>
13
14     <h:messages/>
15
16     <h:form id="form">
17         <h:panelGrid id="grid" columns="3">
18             <h:outputLabel id="bookTitleLabel" value="Book Title:" for="bookTitleFieldField"/>
19             <h:outputText id="bookTitleField" value="#{captureBookDetailsBinding.bookDetails.title}"/>
20
21             <h:outputLabel id="lastNameLabel" value="Last Name:" for="lastNameField"/>
22             <h:outputText id="lastNameField" value="#{captureBookDetailsBinding.bookDetails.authorName.
23                 lastName}"/>
24
25             <h:outputLabel id="firstNameLabel" value="First Name:" for="firstNameField"/>
26             <h:outputText id="firstNameField" value="#{captureBookDetailsBinding.bookDetails.authorName.
27                 firstName}"/>
28
29             <h:outputLabel id="surnameLabel" value="Publication Date:" for="publicationDateField"/>
30             <h:outputText id="publicationDateField" value="#{captureBookDetailsBinding.bookDetails.
31                 publicationDate}"/>
32             <f:convertDateTime pattern="yyyy-MM-dd" />
33             </h:outputText>
34
35             <h:outputLabel id="priceLabel" value="Price:" for="priceField"/>
36             <h:outputText id="priceField" value="#{captureBookDetailsBinding.bookDetails.price}"/>
37         </h:panelGrid>
38     </h:form>
39 </body>
40 </html>

```

```
34         <f:convertNumber type="currency" currencySymbol="R"/>
35     </h:outputText>
36 </h:panelGrid>
37 </h:form>
38 </body>
39 </html>
```

41.4 Exercises

Write a simple web application front-end which captures weather readings at geographic coordinates. The weather reading should contain

- the temperature captured either in degrees Celsius if a capital C is appended or in degrees Fahrenheit if a capital F is appended,
- the weather condition which is one of sunny, overcast, raining or snowing, and
- the geographic location with a longitude and a latitude captured as a string surrounded in round brackets containing the degrees longitude and latitude in a comma-delimited way, e.g. (23.4,33.7)

Chapter 42

Validators

42.1 UI-based validators

UI validators perform the validation at the user interface level. Typically these validations need to be redone at the services and entity levels.

42.1.1 Standard validators

As of JSF 2, five standard validators are provided with JSF:

- `validateRequired`
- `validateLength`
- `validateRange`
- `validateDoubleRange`
- `validateRegex`

None of these require any code to be written in the backing bean (managed or backing bean). One simply assigns the validators in the facelet to the respective fields.

42.1.1.1 Validating required fields

As of JSF 2, there are two mechanisms which can be used to validate that required fields are indeed provided by the user

1. You can set a `required` attribute on the `UIComponent` to `true` :

```
1 <h:inputText required="true" value="#{myBindingComponent.myProperty}"/>
```

2. You can add a `validateRequired` validator to the `UIComponent`:

```
1 <h:inputText value="#{myBindingComponent.myProperty}">
2   <f:validateRequired/>
3 </h:inputText>
```

Note: *This approach treats required field validation symmetrical to other validations.*

42.1.1.2 Length validators

To validate that the length of a character string entered into a field is within some range between some minimum and maximum length we use

```
1 <h:inputText value="#{myBindingComponent.myProperty}">
2   <f:validateLength minimum="2" maximum="30" />
3 </h:inputText>
```

42.1.1.3 Integer range validators

To validate that the user entered an integer number which is within some specified range within the domain of long integers, one uses the `validateLongRange` validator:

```
1 <h:inputText value="#{myBindingComponent.myProperty}">
2   <f:validateLongRange minimum="-999" maximum="999" />
3 </h:inputText>
```

42.1.1.4 Floating point range validators

To validate that the user entered a floating point within some specified range one uses the `validateDoubleRange` validator:

```
1 <h:inputText value="#{myBindingComponent.myProperty}">
2   <f:validateDoubleRange minimum="0.0" maximum="3.5" />
3 </h:inputText>
```

42.1.1.5 Regex validators

To validate that the entered a character string which satisfies some regular expression pattern, one uses the `validateRegex` validator:

```
1 <h:inputText value="#{myBindingComponent.myProperty}">
2   <f:validateRegex pattern="[A-Z]{3} [0-9]{3} GP" />
3 </h:inputText>
```

42.1.2 Custom validators

JSF also enables developers to write custom validators. With the availability of regular expression validation, there is only in very exceptional cases a need to write a custom validator which validates a particular field entry.

However, one can use custom validators to validate values entered into one field, taking into account values entered into other fields.

A custom validator can be implemented either as

- a validation method in the backing bean (managed or backing bean),
- a stand-alone validator class which implements the `javax.faces.Validator` interface.

42.1.3 Simple validation example

Consider a simple credit card capture form which validates that

- both, the name and credit card number are provided,
- that the name is between 2 and 20 characters long.
- that the credit card number is 10 digits long, and
- that the sum of the first 8 digits add up to the number contained in digits 9 and 10.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html
3   PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5   <html xmlns="http://www.w3.org/1999/xhtml"
6     xmlns:f="http://java.sun.com/jsf/core"
7     xmlns:h="http://java.sun.com/jsf/html">
8   <head>
9     <title>Std and Custom Validation: Capture Credit Card</title>
10  </head>
11  <body>
12    <h1><h:outputText id="header1" value="Form with standard and custom validation"/></h1>
13    <h2><h:outputText id="header2" value="Capture credit card"/></h2>
14
15    <h:messages/>
16
17    <h:form id="form">
18      <h:panelGrid id="cardGrid" columns="3">
19        <h:outputLabel id="nameLabel" value="Name:" for="nameField"/>
20        <h:inputText id="nameField" required="true" value="#{captureCreditCardBinding.creditCard.name}">
21          <f:validateLength minimum="2" maximum="20"/>
22        </h:inputText>
23        <h:message for="nameField"/>
24
25        <h:outputLabel id="numberLabel" value="Number:" for="numberField"/>
26        <h:inputText id="numberField" required="true" value="#{captureCreditCardBinding.creditCard.number}">
27          <f:validator validatorId="checkSumValidator"/>
28          <f:validateLength minimum="10" maximum="10"/>
29          <f:validateRegex pattern="[0-9]{10}" />
30        </h:inputText>
31        <h:message for="numberField"/>
32      </h:panelGrid>
33
34      <h:commandButton id="save" action="#{captureCreditCardBinding.captureCreditCard}" value="save" />
35    </h:form>
36  </body>
37 </html>

```

All except the last validation are done using standard validators. The validators throw validation exceptions which automatically push the associated messages onto the message queue in order to be shown in the page-wide and component specific message tags.

42.1.3.1 Simple check-sum validator

Check sums are used on things like credit card numbers, passport or identity document numbers and so on. The use of check-sums reduces the probability that an incorrect number is typed in.

Assume we have a particularly 10 digit code with the last 2 digits being a very simple check-sum which must be equal to the sum of the first 8 digits. We can write a custom check-sum

Figure 42.1: Standard validators doing their job
Form with standard and custom validation

Capture credit card

- form.nameField: Validation Error: Value is required.
- Not a 10 digit credit card number
- Regex Pattern not matched

| | | |
|-------------------------------------|---|--|
| Name: | <input type="text"/> | form.nameField: Validation Error: Value is required. |
| Number: | <input type="text" value="234324s324"/> | Enter 10 digit credit card number |
| <input type="button" value="save"/> | | |

validator which validates that the first 8 digits add up to the number contained in the last 2 digits.

Our custom `ChecksumValidator` is shown below:

```

1 package za.co.solms.finance.ui.web;
2
3 import javax.faces.application.FacesMessage;
4 import javax.faces.component.UIComponent;
5 import javax.faces.context.FacesContext;
6 import javax.faces.validator.FacesValidator;
7 import javax.faces.validator.Validator;
8 import javax.faces.validator.ValidatorException;
9
10 import com.sun.org.apache.xerces.internal.impl.xpath.regex.ParseException;
11
12 @FacesValidator(value = "checksumValidator")
13 public class CheckSumValidator implements Validator
14 {
15
16     @Override
17     public void validate(FacesContext context, UIComponent component, Object value)
18         throws ValidatorException
19     {
20         String componentValue = value.toString();
21
22         FacesMessage not10DigitsMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
23             "Not a 10 digit credit card number", "Enter 10 digit credit card number");
24
25         if (componentValue.length() < 10) throw new ValidatorException(not10DigitsMsg);
26
27         int sum = 0;
28         for (int i=0; i<8; ++i)
29         {
30             try
31             {
32                 sum += Integer.parseInt(componentValue.substring(i, i+1));
33             }
34             catch (NumberFormatException e)
35             {
36                 throw new ValidatorException(not10DigitsMsg);
37             }
38         }
39         try
40         {
41             int checkSum = Integer.parseInt(componentValue.substring(8, 10));
42
43             if (sum != checkSum)
44             {
45                 FacesMessage checkSumFailedMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
46                     "Invalid credit card number", "Check sum failed");
47                 throw new ValidatorException(checkSumFailedMsg);
48             }
49         }
50     }
51 }

```

```

50     catch (NumberFormatException e)
51     {
52         throw new ValidatorException(not10DigitsMsg);
53     }
54 }
55
56 }

```

Figure 42.2: Failed check-sum validation

Form with standard and custom validation

Capture credit card

- Invalid credit card number

| | | |
|-------------------------------------|--|------------------|
| Name: | <input type="text" value="Tandi Smith"/> | |
| Number: | <input type="text" value="1111111107"/> | Check sum failed |
| <input type="button" value="save"/> | | |

It fails the user input if the first eight digits do not add up to the number contained in digits 9 and 10. On the other hand, if the check sum validation passes, we receive the notification that the credit card has been saved.

Figure 42.3: Passed with check sum satisfied

Form with standard and custom validation

Credit card saved confirmation

Credit card 1111111108 has been saved.

42.1.4 Exercises

- Add an expiry date to the credit card capture form. Make it required and add an appropriate regular expression validator. Check that it works.
- Replace the regular expression validator with a custom validator.

42.2 Bean Validation

One of the disadvantages of UI-based validation include is the potential duplication of the specification of data structure constraints across different layers like presentation, services and domain objects layers, and the resulting maintenance costs and inconsistency risks. As of JavaEE 6, JavaEE supports enables one to specify data structure constraints independent of the layer in which they are used. One should thus strongly consider using bean validation. Bean validation (JSR-303) enables one to annotate the fields of a data class which implements the Java Beans API or an entity with either some standard constraints (e.g. value, range or pattern constraints), or some custom constraints which are validated using custom validators. For example, when annotating an entity, JavaEE will enforce any beans validation based constraints at the persistence level. If that entity is passed as a value object to the presentation layers, then the data structure

constraints specified within the JSR-303 spec are still enforced when calling the setters on the bean. Any exception throw by the validation interception are typically rendered within the presentation layer framework (e.g. within JSF). The reference implementation of bean validation is provided by *Hibernate Validator*. Bean validation support is provided by JavaEE 6 implementing application servers. If one wants to use bean validation outside a JavaEE container, one can use a bean validation framework like *Hibernate Validator*.

42.2.1 Standard constraint annotations

JSR 303 defines a series of constraint annotations which can be applied to the fields of a data structure class or a backing bean.

- **@AssertFalse and @AssertTrue:** To constrain the annotated value to either `true` or `false`. For example

```
1 @AssertTrue
2 private boolean happy;
```

- **@Null and @NotNull:** These are used to specify that the annotated field must either be equal to null or may not be null. For example

```
1 @NotNull
2 private Person composer;
```

- **@Size:** This annotation can be applied to strings and collections. It specifies that the number of characters in a string or the number of elements in a collection must sum value between `min` and `max`. For example

```
1 @Size(min=4, max=6, message="Password must be between 4 and 6 characters long")
2 private String password;
```

- **@Pattern:** This constraint is applied to strings to specify that the entered set of characters must comply to a specified regular expression pattern. For example, you could use a pattern constraint to specify the legal values for a vehicle registration number:

```
1 @Pattern(regexp="[A-Z]{3} [0-9]{3} (GP|EC|WC|NC|ML|KN|LM)")
2 private String vehicleRegistrationNumber;
```

- **@Min, @Max, @DecimalMin, @DecimalMax:** These annotations are applied to integers and floating numbers respectively in order to constrain the respective range of values.

```
1 @DecimalMin(value="0.0", message="Percentage must be between 0 and 100")
2 @DecimalMax(value="100.0", message="Percentage must be between 0 and 100")
3 private double percentageMark;
```

- **@Past and @Future:** These are applied to dates in order to specify that the value of the annotated date field must be either a date in the past or in the future:

```
1 @Past(message="You cannot report an accident which you expect to happen in the future")
2 private Date accidentDate;
```


42.2.2 Specifying custom bean constraints

In order to specify custom bean constraints, one needs to define In order to specify custom bean constraints, one needs to define

- a constraint annotation for the custom constraint,
- a validator class which validates the custom constraint.

42.2.2.1 Defining an annotation for a custom constraint

For a custom constraint, one has to define a custom annotation through which that particular constraint is annotated.

```

1 @Constraint(validatedBy = CreditCardNumberValidator.class)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target(ElementType.FIELD)
4 public @interface MyConstraint
5 {
6     String message() default "My default msg if constraint not met.";
7
8     Class<?>[] groups() default {};
9
10    Class<? extends Payload>[] payload() default {};
11 }

```

Note that

- We specify the validator which will be used to validate our custom constraint.
- The retention policy needs to be `RUNTIME` in order for it to be available to validation frameworks at run time.

42.2.2.2 Defining a validator for a custom constraint

The validator needs to implement the `ConstraintValidator` template binding

- the first template parameter to the constraint annotation, and
- the second template parameter to the type whose instances are being constrained.

The code for the validator would look like this:

```

1 public class MyConstraintValidator
2     implements ConstraintValidator<MyConstraint, TypeBeingConstrained>
3 {
4     @Override
5     public void initialize(MyConstraint constraint) {}
6
7     @Override
8     public boolean isValid(TypeBeingConstrained constrainedObject,
9                           ConstraintValidatorContext context)
10    {
11        // validation logic returning true if valid and false otherwise
12    }
13 }

```

42.2.3 Annotating fields of backing bean

One can apply JSR 303 constraint annotations directly to the backing bean fields. Doing this has the disadvantage that the data structure constraints will still need to be duplicated across layers. It does, on the other hand, remove those constraints from the view elements, allowing view constructors to focus on the actual view without worrying about any data structure constraints applied to the data captured through the view.

42.2.3.1 Example

Here we show a minimalistic example which applies some bean validations to the backing bean of a presentation layer process. The backing bean has data fields for the name on a credit card and the credit card number and constrains these with some simple constraints.

42.2.3.1.1 The POM The POM needs to now include the dependencies on the validation spec, and, in the case where we intend to deploy outside a JavaEE compliant application server, also the dependency on a validation framework implementation:

```

1 <?xml version="1.0"?>
2 <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.
  xsd"
3   xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>za.co.solms.training.jsf</groupId>
8   <artifactId>backingBeanValidation</artifactId>
9   <packaging>war</packaging>
10  <version>1.0-SNAPSHOT</version>
11
12  <name>backingBeanValidation</name>
13  <url>http://maven.apache.org</url>
14
15  <parent>
16    <groupId>za.co.solms.training.jsf</groupId>
17    <artifactId>examples</artifactId>
18    <version>1.0-SNAPSHOT</version>
19    <relativePath>../pom.xml</relativePath>
20  </parent>
21
22  <dependencies>
23    <dependency>
24      <groupId>javax.validation</groupId>
25      <artifactId>validation-api</artifactId>
26      <version>1.1.0.Final</version>
27      <scope>compile</scope>
28    </dependency>
29
30    <dependency>
31      <groupId>org.hibernate</groupId>
32      <artifactId>hibernate-validator</artifactId>
33      <version>5.1.0.Final</version>
34    </dependency>
35
36  </dependencies>
37 </project>

```

42.2.3.1.2 A custom credit card number constraint annotation Our custom credit card number constraint is an annotation for which we specify:

- the validator class for the constraint,

- that the annotation needs to be retained up to run-time so that it is available to the application server or validation framework during run-time,
- that the target of the annotation is a field **Note:** *For the validation specification we will see that the field must be of type `String`.*

```

1 package za.co.solms.finance.model;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 import javax.validation.Constraint;
9 import javax.validation.Payload;
10
11 @Constraint(validatedBy = CreditCardNumberValidator.class)
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.FIELD)
14 public @interface CreditCardNumberConstraint
15 {
16     String message() default "Invalid credit card number";
17
18     Class<?>[] groups() default {};
19
20     Class<? extends Payload>[] payload() default {};
21 }

```

42.2.3.1.3 A custom credit card number validator The validator implements the `ConstraintValidator` template interface, taking the constraint as first template parameter and the type of the field to be validated with the validator as second template parameter. The `isValid` service gets the field instance and its context as parameters and encodes the validation logic:

```

1 package za.co.solms.finance.model;
2
3 import javax.validation.ConstraintValidator;
4 import javax.validation.ConstraintValidatorContext;
5
6 public class CreditCardNumberValidator
7     implements ConstraintValidator<CreditCardNumberConstraint, String>
8 {
9     @Override
10     public void initialize(CreditCardNumberConstraint constraint) {}
11
12     @Override
13     public boolean isValid(String cardNumber, ConstraintValidatorContext context)
14     {
15         if (cardNumber.length() != 10)
16             return false;
17
18         boolean valid = true;
19         int sum = 0;
20         for (int i=0; i<8; ++i)
21         {
22             try
23             {
24                 sum += Integer.parseInt(cardNumber.substring(i, i+1));
25             }
26             catch (NumberFormatException e)
27             {
28                 return false;
29             }
30         }
31         try
32

```

```

33     {
34         int checksum = Integer.parseInt(cardNumber.substring(8, 10));
35
36         if (sum != checksum)
37         {
38             return false;
39         }
40         else
41         {
42             return true;
43         }
44     }
45     catch (NumberFormatException e)
46     {
47         return false;
48     }
49 }
50
51 }

```

42.2.3.1.4 The view The view is now devoid of any validators:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.
5  sun.com/jsf/html">
6  <head>
7  <title>Bean Validation on Backing Bean: Capture Credit Card</title>
8  </head>
9  <body>
10     <h1><h:outputText id="header1" value="Form with standard and custom validation"/></h1>
11     <h2><h:outputText id="header2" value="Capture credit card"/></h2>
12
13     <h:messages/>
14
15     <h:form id="form">
16         <h:panelGrid id="cardGrid" columns="3">
17             <h:outputLabel id="nameLabel" value="Name:" for="nameField"/>
18             <h:inputText id="nameField" value="#{captureCreditCardBinding.nameOnCreditCard}"/>
19             <h:message for="nameField"/>
20
21             <h:outputLabel id="numberLabel" value="Number:" for="numberField"/>
22             <h:inputText id="numberField" value="#{captureCreditCardBinding.creditCardNumber}"/>
23             <h:message for="numberField"/>
24         </h:panelGrid>
25
26         <h:commandButton id="save" action="#{captureCreditCardBinding.captureCreditCard}" value="save"
27         />
28     </h:form>
29 </body>
30 </html>

```

42.2.3.1.5 The backing bean The fields of the backing bean have now been annotated with constraint annotations including some standard constraint annotations and our custom `CreditCardNumberConstraint` annotation:

```

1  package za.co.solms.finance.ui.web;
2
3  import java.io.Serializable;
4
5  import javax.faces.bean.ManagedBean;
6  import javax.faces.bean.SessionScoped;
7  import javax.validation.constraints.Pattern;
8  import javax.validation.constraints.Size;

```

```

9
10 import za.co.solms.finance.model.CreditCard;
11 import za.co.solms.finance.model.CreditCardNumberConstraint;
12
13 @ManagedBean
14 @SessionScoped
15 public class CaptureCreditCardBinding implements Serializable
16 {
17     public String captureCreditCard()
18     {
19         CreditCard creditCard = new CreditCard(creditCardNumber, nameOnCreditCard);
20         //persist credit card
21         return "creditCardSavedConfirmation";
22     }
23
24     public String getNameOnCreditCard()
25     {
26         return nameOnCreditCard;
27     }
28
29     public void setNameOnCreditCard(String nameOnCreditCard)
30     {
31         this.nameOnCreditCard = nameOnCreditCard;
32     }
33
34     public String getCreditCardNumber()
35     {
36         return creditCardNumber;
37     }
38
39     public void setCreditCardNumber(String creditCardNumber)
40     {
41         this.creditCardNumber = creditCardNumber;
42     }
43
44     @Size(min=2, max=30, message="Please enter name on card")
45     private String nameOnCreditCard;
46
47     @Pattern(regexp="[0-9]{10}", message="Please enter 10 digit credit card number")
48     @CreditCardNumberConstraint
49     private String creditCardNumber;
50 }

```

42.2.3.2 Exercise

Write a little web application which captures the details of magic shows at a particular school. You should capture

- the name of the performer,
- the date and times of the next performance,
- the total number of performances which will be given, and
- a contact email address which can be used to make reservations for the show.

Apply sensible data structure constraints using JSR 303 bean validations on the backing bean.

42.2.4 Annotating a data class (value object or entity)

JSR 303 also supports the annotation of a class with state constraints. This is done by

- declaring the target of the constraint annotation a TYPE

```

1 @Constraint(validatedBy = MyConstraintValidator.class)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target(ElementType.TYPE)
4 public @interface MyConstraints
5 {
6     String message() default "my default message";
7
8     Class<?>[] groups() default {};
9
10    Class<? extends Payload>[] payload() default {};
11 }

```

and

- annotating a data class (e.g. detachable entity) with that constraint:

```

1 @MyConstraint
2 public class myDataClass
3 {
4     ...
5 }

```

42.2.4.1 Example

As a simple example, consider a `Person` class for which there is an invariance constraint (data integrity constraint) that the first 6 digits of the identity number are determined by the date of birth. In this case there is a constraint across field of a class - the constraint is not specific to a particular field.

```

1 package za.co.solms.persons.model;
2
3 import javax.validation.Constraint;
4 import javax.validation.Payload;
5
6 import java.lang.annotation.ElementType;
7 import java.lang.annotation.Retention;
8 import java.lang.annotation.RetentionPolicy;
9 import java.lang.annotation.Target;
10
11 @Constraint(validatedBy = ConsistentIdentityNumberValidator.class)
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target({ElementType.TYPE})
14 public @interface ConsistentIdentityNumberConstraint
15 {
16     String message() default "Identity number not consistent with date of birth.";
17
18     Class<?>[] groups() default {};
19
20     Class<? extends Payload>[] payload() default {};
21 }

```

```

1 package za.co.solms.persons.model;
2
3 import javax.validation.ConstraintValidator;
4 import javax.validation.ConstraintValidatorContext;
5 import java.util.Calendar;
6 import java.util.GregorianCalendar;
7
8 public class ConsistentIdentityNumberValidator

```

```

9  implements ConstraintValidator<ConsistentIdentityNumberConstraint, Person>
10 {
11
12     @Override
13     public void initialize(ConsistentIdentityNumberConstraint constraint){}
14
15     @Override
16     public boolean isValid(Person person, ConstraintValidatorContext context)
17     {
18         System.out.println("##### Validating");
19         Calendar cal = new GregorianCalendar();
20         cal.setTime(person.getDateOfBirth());
21         int lastTwoDigitsOfBirthYear = cal.get(Calendar.YEAR) / 100;
22         int birthMonth = cal.get(Calendar.MONTH);
23         int birthDayOfMonth = cal.get(Calendar.DAY_OF_MONTH);
24
25         String idNo = person.getIdentityNumber();
26
27         try
28         {
29             return (Integer.parseInt(idNo.substring(0,1)) == lastTwoDigitsOfBirthYear)
30                 && (Integer.parseInt(idNo.substring(2,3)) == birthMonth)
31                 && (Integer.parseInt(idNo.substring(5,6)) == birthDayOfMonth);
32         }
33         catch (NumberFormatException e)
34         {
35             System.out.println("##### NFE");
36             return false;
37         }
38     }
39
40 }

```

```

1  package za.co.solms.persons.model;
2
3  import javax.validation.constraints.Past;
4  import javax.validation.constraints.Pattern;
5  import java.util.Date;
6
7  @ConsistentIdentityNumberConstraint
8  public class Person
9  {
10     public Person(){}
11
12     public Person(String firstName, String surname, Date dateOfBirth, String identityNumber)
13     {
14         setFirstName(firstName);
15         setSurname(surname);
16         setDateOfBirth(dateOfBirth);
17         setIdentityNumber(identityNumber);
18     }
19
20     public String getFirstName()
21     {
22         return firstName;
23     }
24
25     public void setFirstName(String firstName)
26     {
27         this.firstName = firstName;
28     }
29
30     public String getSurname()
31     {
32         return surname;
33     }
34
35     public void setSurname(String surname)

```

```

36 {
37     this.surname = surname;
38 }
39
40 public Date getDateOfBirth()
41 {
42     return dateOfBirth;
43 }
44
45 public void setDateOfBirth(Date dateOfBirth)
46 {
47     this.dateOfBirth = dateOfBirth;
48 }
49
50 public String getIdentityNumber()
51 {
52     return identityNumber;
53 }
54
55 public void setIdentityNumber(String identityNumber)
56 {
57     this.identityNumber = identityNumber;
58 }
59
60 public boolean equals(Object o)
61 {
62     try
63     {
64         Person arg = (Person) o;
65         return firstName.equals(arg.firstName) &&
66             surname.equals(arg.surname) &&
67             dateOfBirth.equals(arg.dateOfBirth) &&
68             identityNumber.equals(arg.identityNumber);
69     }
70     catch (ClassCastException e)
71     {
72         return false;
73     }
74 }
75
76 public int hashCode()
77 {
78     return identityNumber.hashCode();
79 }
80
81 // @Pattern(regexp="[A-Z][a-z]{*}")
82 private String firstName;
83 // @Pattern(regexp="[A-Z][a-z]{*}")
84 private String surname;
85 @Past
86 private Date dateOfBirth;
87 @Pattern(regexp="[0-9]{13}")
88 private String identityNumber;
89 }

```

42.2.4.2 Exercises

Define the standard and custom validators needed to validate the weather readings front-end elements discussed in the exercise for the converters chapter.

42.2.5 Bean Validation Example

Consider a weather reading entity which has JSR-303 based data constraints applied to the temperature, humidity and weather reading date:


```

1  /**
2   *
3   */
4  package za.co.solms.weather;
5
6  import java.io.Serializable;
7  import java.util.Date;
8
9  import javax.persistence.Column;
10 import javax.persistence.Entity;
11 import javax.persistence.GeneratedValue;
12 import javax.persistence.Id;
13 import javax.persistence.JoinColumn;
14 import javax.persistence.ManyToOne;
15 import javax.persistence.NamedQueries;
16 import javax.persistence.NamedQuery;
17 import javax.persistence.Temporal;
18 import javax.persistence.TemporalType;
19 import javax.validation.constraints.Past;
20
21 import org.hibernate.validator.constraints.Range;
22
23 import za.co.solms.location.Location;
24
25 /**
26  * @author fritz@solms.co.za
27  *
28  */
29 @Entity
30 @NamedQueries({
31     @NamedQuery(name="findAllWeatherReadings",
32         query="Select wr from WeatherReading wr"),
33     @NamedQuery(name="findAllWeatherReadingsForLocation",
34         query="select wr from WeatherReading wr where wr.location = :location"),
35     @NamedQuery(name="findAllWeatherReadingsForLocationAndPeriod",
36         query="select wr from WeatherReading wr where wr.location = :location"
37             + " and wr.dateTime > :after and wr.dateTime <= :onOrBefore")
38 })
39
40 public class WeatherReading implements Serializable
41 {
42     public WeatherReading() {}
43
44     public WeatherReading(Location location, Date dateTime,
45         double temperature, double humidity, Ambiance ambiance)
46     {
47         setLocation(location);
48         setDateTime(dateTime);
49         setTemperature(temperature);
50         setHumidity(humidity);
51         setAmbiance(ambiance);
52     }
53
54     @Id
55     @GeneratedValue
56     public int getId()
57     {
58         return id;
59     }
60
61     public void setId(int id)
62     {
63         this.id = id;
64     }
65
66     public double getTemperature()
67     {
68         return temperature;
69     }
70
71     public void setTemperature(double temperature)
72     {
73         this.temperature = temperature;

```

```

74 }
75
76 public double getHumidity()
77 {
78     return humidity;
79 }
80
81 public void setHumidity(double humidity)
82 {
83     this.humidity = humidity;
84 }
85
86 @Temporal(TemporalType.TIMESTAMP)
87 @Column(nullable=false)
88 public Date getDateTime()
89 {
90     return dateTime;
91 }
92
93 public void setDateTime(Date dateTime)
94 {
95     this.dateTime = dateTime;
96 }
97
98 public Ambiance getAmbiance()
99 {
100     return ambiance;
101 }
102
103 public void setAmbiance(Ambiance ambiance)
104 {
105     this.ambiance = ambiance;
106 }
107
108 @ManyToOne
109 @JoinColumn(nullable=false)
110 public Location getLocation()
111 {
112     return location;
113 }
114
115 public void setLocation(Location location)
116 {
117     this.location = location;
118 }
119
120 public boolean equals(Object o)
121 {
122     try
123     {
124         WeatherReading arg = (WeatherReading)o;
125         return (this.id == arg.id) && (this.temperature == arg.temperature)
126             && (this.humidity == arg.humidity) && (this.ambiance == arg.ambiance);
127     }
128     catch (ClassCastException e)
129     {
130         return false;
131     }
132 }
133
134 public int hashCode()
135 {
136     return id + dateTime.hashCode() + new Double(temperature).hashCode()
137         + new Double(humidity).hashCode();
138 }
139
140 private int id;
141
142 @Range(min=-60,max=60)
143 private double temperature;
144 @Range(min=0, max=100)
145 private double humidity;
146 @Past

```

```

147 private Date dateTime;
148 private Ambiance ambiance;
149 private Location location;
150 }

```

The data constraints will be enforced during persistence, i.e. the persistence framework (e.g. object-relational mapper) will raise an exception if a weather reading which does not satisfy the JSR-303 constraints is requested to be persisted or updated.

But these same business rules based constraints are enforced at the presentation layer. For example, if we access the bean properties from a JSF facelet via UEL-based expression:


```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/
  DTD/xhtml1-transitional.dtd" >
2 <fieldset xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core"
6   xmlns:p="http://primefaces.org/ui"
7   xmlns:comp="http://java.sun.com/jsf/composite/components" >
8
9   <legend>WeatherReading</legend>
10
11   <!-- The following line forces the initialization of the backing bean for this aggregate view. -->
12   <h:inputHidden value="#{weatherReadingBackingBean.dummy}" />
13
14   <ui:include src="/faces/locations/locationSelector.xhtml" />
15   <ui:include src="/faces/dateTime/dateTimePanel.xhtml" />
16
17   <comp:collapsiblePanel model="#{weatherReadingBackingBean}" >
18     <f:facet name="header">
19       <h3><h:outputText value="#{msgs.weatherMeasurements}" /></h3>
20     </f:facet>
21
22     <h:panelGrid columns="2" >
23
24       <h:outputLabel id="temperatureLabel" for="temperatureField"
25         value="#{msgs.temperature}" />
26       <h:inputText id="temperatureField"
27         value="#{weatherReadingBackingBean.weatherReading.temperature}" />
28
29       <h:outputLabel id="humidityLabel" for="humidityField"
30         value="#{msgs.humidity}" />
31       <h:inputText id="humidityField"
32         value="#{weatherReadingBackingBean.weatherReading.humidity}" />
33
34       <h:outputLabel id="ambianceLabel" for="ambianceField"
35         value="#{msgs.ambiance}" />
36       <h:selectOneMenu id="ambianceField"
37         value="#{weatherReadingBackingBean.weatherReading.ambiance}" >
38         <f:selectItems
39           value="#{weatherReadingBackingBean.ambianceValues}" />
40       </h:selectOneMenu>
41
42     </h:panelGrid>
43   </comp:collapsiblePanel>
44
45 </fieldset>

```

and attempt to violate any of the bean constraints we end up with the corresponding error messages shown in the figure below.

Figure 42.4: Bean constraints enforced in the JSF based presentation layer



Heaven's Connection Weather Buro

Weather Buro ▾ Weather ▾ Locations ▾

- must be between -60 and 60
- must be between 0 and 100

WeatherReading

Location selector

wereeeeeeee ▾

Date/Time

Date 03/17/2011

Hour 0 ▴ ▾

Minute 0 ▴ ▾

Weather measurements

Temperature 99

Humidity -10

Ambiance sunny ▾

add weather reading cancel

Chapter 43

Internationalization

43.1 Overview

Internationalization is the name given to the mechanism to render views across multiple languages. The language used is determined by the user's locale settings as set for the user agent (e.g. the browser).

Internationalization requires the following to be done

1. Informing the JSF engine about the supported locales and the location and name of the properties files representing the resource bundles.
2. Create the resource bundle properties files for the different supported locales and populate them with key-value pairs for the text to be used for the abstracted text-keys used in the web application.
3. Abstracting all labels, output texts and message texts to refer to resource bundle keys.
4. Specifying the resource bundles for the different supported locales as key-value pairs.

43.2 Specifying supported locales

The supported locales are specified in the `faces-config.xml`. One needs to specify

- the base file name and location of the properties files representing the resource bundles,
- the variable which is used to refer to the resource bundle,
- the default locale,
- any other supported locales.

Consider the following `faces-config` file:

```
1 <?xml version="1.0"?>
2 <faces-config version="2.0" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
  ns/javaee/web-facesconfig_2.0.xsd">
3   <application>
4     <locale-config>
```

```

5 <default-locale>en-ZA</default-locale>
6 <supported-locale>xh-ZA</supported-locale>
7 <supported-locale>af-ZA</supported-locale>
8 </locale-config>
9
10 <resource-bundle>
11   <base-name>za.co.solms.training.jsf.messages</base-name>
12   <var>msgs</var>
13 </resource-bundle>
14 </application>
15 </faces-config>

```

Here we specify that

- the default locale is South-African English.
- Xhosa and Afrikaans are also supported.
- the resource bundle base name, i.e. the name of the properties file for the resource bundle for the default locale is `messages.properties` and that it is available at `za/co/solms/training/jsf/` (relative to a directory which is in the class path),
- the variable used in the EL expressions to access the key-value pairs in the resource bundle is `msgs`, e.g. `#msgs.detailedInvoice`.

43.3 Creating the resource bundles for the different locales

For each supported locale one needs to create a properties file containing the key-value mappings for the text representations to be used for the various keys.

Thus, for the default locale, `en-ZA` we would have to create a properties file named `messages.properties` which is located in a directory `za/co/solms/training/jsf/` (relative to a directory which is in the class path), whose content could look something like the following:

```

1 bread=bread
2 pleaseBringYourChildren=Please bring your children.
3 price=price
4 surname=surname
5 workAddress=work address
6 yourName=your name

```

The corresponding properties file for the Xhosa locale would be stored as `messages_en-XH.properties` and would be something like

```

1 bread=isonka
2 pleaseBringYourChildren=Sicela, zisa abantwana bakho.
3 price=ixabiso
4 surname=ifani
5 workAddress=i-adresi lasembenzini
6 yourName=igama lakho

```

43.4 Abstracting facelet texts

Any text which is to be inserted into a view needs to be abstracted via the variable through which we pull in the value associated with a key in the resource bundle to be used for a particular user.

For example:

```

1 <h:form>
2
3     <fieldset>
4         <legend>Location selector</legend>
5         <h:selectOneMenu id="locationSelector" value="#{locationSelectorBinding.locationName}">
6             <f:selectItems value="#{locationSelectorBinding.locations}" />
7         </h:selectOneMenu>
8     </fieldset>
9
10    <fieldset>
11        <legend>Date/Time</legend>
12
13        <h:panelGrid columns="2">
14            <h:outputLabel id="dateLabel" for="dateField" value="#{msgs.date}" />
15            <p:calendar id="dateField" value="#{dateTimeBinding.date}" title="#{msgs.date}" />
16
17            <h:outputLabel id="hourLabel" for="hourField" value="#{msgs.hour}" />
18            <p:spinner id="hourField" value="#{dateTimeBinding.hour}" />
19
20            <h:outputLabel id="minuteLabel" for="minuteField" value="#{msgs.minute}" />
21            <p:spinner id="minuteField" value="#{dateTimeBinding.minute}" />
22        </h:panelGrid>
23    </fieldset>
24
25    <fieldset>
26        <legend>DWeather conditions</legend>
27        <h:panelGrid columns="2">
28
29            <h:outputLabel id="temperatureLabel" for="temperatureField" value="#{msgs.temperature}" />
30            <h:inputText id="temperatureField" value="#{weatherDetailsBinding.temperature}" />
31
32            <h:outputLabel id="humidityLabel" for="humidityField" value="#{msgs.humidity}" />
33            <h:inputText id="humidityField" value="#{weatherDetailsBinding.humidity}" />
34
35            <h:outputLabel id="ambianceLabel" for="ambianceField" value="#{msgs.ambiance}" />
36            <h:selectOneMenu id="ambianceField" value="#{weatherDetailsBinding.ambiance}">
37                <f:selectItems value="#{weatherDetailsBinding.ambianceValues}" />
38            </h:selectOneMenu>
39        </h:panelGrid>
40    </fieldset>
41
42    <h:commandButton id="addWeatherReadingButton" value="#{msgs.addWeatherReading}"
43        action="#{addWeatherReadingBinding.addWeatherReading}" />
44
45    <h:commandButton id="cancel" value="#{msgs.cancel}" action="/faces/weather/weatherQuery" />
46
47 </h:form>

```

43.5 Abstracting text generated in the binding bean (e.g. messages)

At times one needs to generate text messages in a binding bean. In those cases one needs to

- get hold of the appropriate resource bundle, specifying the base/default resource bundle name (the framework will provide the appropriate resource bundle for the locale used by the user), and
- retrieve the localized text for a text key from the resource bundle:

```

1 public String registerPerson()
2 {
3     Person person = personDetailsBinding.getPerson();
4

```

```
5 RegisterPersonRequest registrationRequest
6   = new RegisterPersonRequest(person);
7
8 if (!validRegisterPersonInfo())
9   return "";
10
11 try
12 {
13   personServices.registerPerson(registrationRequest);
14 }
15 catch (UserExistsException e)
16 {
17   ResourceBundle resourceBundle = ResourceBundle.getBundle
18     ("za.co.solms.training.jsf.messages",
19     FacesContext.getCurrentInstance().getViewRoot().getLocale());
20
21   String shortMsg = resourceBundle.getString("userExists");
22   String longMsg = resourceBundle.getString("thereIsAlreadyAUserWithThatUsername");
23
24   FacesMessage msg = new FacesMessage(FacesMessage.SEVERITY_WARN, shortMsg, longMsg);
25   FacesContext.getCurrentInstance().addMessage(null, msg );
26   return "";
27 }
28 return "registerPersonConfirmation";
29 }
```


Chapter 44

Templating

44.1 Overview

A template is effectively a parametrized page which

- specifies a layout with different abstract parametrized layout domains or panels,
- the ability to dynamically insert components into specific positions within a layout,
- facilitates the specification of default components in particular layout positions, and
- allows for the positioning, sizing and styling of the layout panels to be specified in a style sheet.

44.1.1 Example use case

For example, one could specify a template which has a logo panel on the top of the page, a main navigation panel as a horizontal panel below the logo panel, a detailed navigation panel as a left panel, a content panel in the center of the page, a related features panel as a right panel and a footer panel. For each of these positions in the layout one can specify a default component which will be used if no specific component is requested to be inserted in that template position. When defining a particular page one can then insert only those components which are specific for the page.

For example, the logo panel, main navigation panel and footer panel could, for arguments sake be the same across pages, the main content panel could be provided by the developer and the detailed navigation panel as well as the related content panel could be generated by a menu generator and a content generator respectively.

Templates thus provide a mechanism through which one can achieve a consistent layout across pages. Using templates provides a range of benefits including

- improves the user experience by providing a consistent infrastructure across pages, reducing complexity and improving efficiency,
- provides a more modular approach to presentation layer development, decoupling layout from content,
- improves the maintainability by reducing redundancy and enabling developers or designers to easily change the layout of an entire site,

44.2 Specifying a template

A template is a facelet (an XHTML file) which contains insertion point specifications in the form of `<ui:insert/>` which can be populated in the various pages which use the template via a `<ui:define/>` tag.

Consider, for example, the following main template used for the default layout of a site:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:ui="http://java.sun.com/jsf/facelets"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:p="http://primefaces.prime.com.tr/ui">
7   <h:head>
8
9     <link href="/sbss/resources/css/default.css" rel="stylesheet"
10         type="text/css" />
11     <link href="/sbss/resources/css/cssLayout.css" rel="stylesheet"
12         type="text/css" />
13
14     <title>Weather Buro</title>
15   </h:head>
16   <body>
17     <h:outputStylesheet library="css" name="cssLayout.css" target="head" />
18     <div id="page">
19
20       <div id="header" class="decorator">
21         <ui:insert name="header">
22           header
23         </ui:insert>
24       </div>
25
26       <div id="navigation" class="outerFrame">
27         <ui:insert name="navigation">
28           navigation
29         </ui:insert>
30       </div>
31
32       <div id="container">
33
34         <div id="content" class="content">
35           <ui:insert name="content">
36             Center content goes here
37           </ui:insert>
38         </div>
39
40         <div id="auxillaryContent" class="outerFrame">
41           <ui:insert name="auxillaryContent">
42             <ui:include src="news.xhtml" />
43           </ui:insert>
44         </div>
45
46       </div>
47
48       <div id="footer" class="outerFrame">
49         <ui:insert name="footer">
50           footer
51         </ui:insert>
52       </div>
53
54     </div>
55   </body>
56 </html>

```

It defines a region for the entire page with 3 sub-regions (header, container and footer) with the container having both, the main and the auxiliary content.

The regions defined are thus abstract regions without specifying

- where and how they are placed,

- the content they hold.

The placement, sizing and styling will be specified by a *style sheet*.

Instead of specifying the content, an insertion point is included for each region, supporting the plugging in of actual content into those regions.

44.3 Partial template population

Note that our template did not specify any content at all. However, a lot of the content like the header with the logo, the main navigation and the footer, would potentially be the same across a large percentage of pages.

We can now specify a partially populated template where we leave out the main content as well as potentially some other regions (e.g. the auxiliary content). This is done by defining the content for a template insertion point and including the actual content at that point.

For example, to insert a the main-menu contained in a separate `mainMenu.xhtml` file into the region labeled *"navigation"*, we can use

```
1 <ui:define name="navigation">
2   <ui:include src="mainMenu.xhtml" />
3 </ui:define>
```

In our case we specify the content of the header, navigation, footer and auxiliary regions:

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:f="http://java.sun.com/jsf/core"
6       xmlns:h="http://java.sun.com/jsf/html"
7       xmlns:ui="http://java.sun.com/jsf/facelets">
8 <head>
9   <title>Default page</title>
10 </head>
11 <body>
12   <ui:composition template="mainTemplate.xhtml">
13
14     <ui:define name="header">
15       <ui:include src="header.xhtml" />
16     </ui:define>
17
18     <ui:define name="navigation">
19       <ui:include src="mainMenu.xhtml" />
20     </ui:define>
21
22     <ui:define name="auxillaryContent">
23       <ui:include src="news.xhtml" />
24     </ui:define>
25
26     <ui:define name="footer">
27       <ui:include src="copyrightNotice.xhtml" />
28     </ui:define>
29
30   </ui:composition>
31 </body>
32 </html>
```

44.4 Adding the main content

The actual application forms and panels are inserted into the main content of the partially populated template.

For example, in the following listing we add the `addWeatherReadingPanel` to the `addWeatherReading` page which is built up from our partially populated template:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4   xmlns:ui="http://java.sun.com/jsf/facelets"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:p="http://primefaces.prime.com.tr/ui">
8
9   <h:head>
10     <meta http-equiv="Content-Type" content="text/html;
11       charset=iso-8859-1" />
12     <title>Add weather reading</title>
13   </h:head>
14
15   <h:body>
16     <ui:composition template="/faces/partiallyPopulatedTemplate.xhtml">
17
18       <ui:define name="content">
19         <ui:include src="addWeatherReadingPanel.xhtml" />
20       </ui:define>
21     </ui:composition>
22   </h:body>
23 </html>

```

44.5 Multi-level templates and template parameters

The content of a template region can itself be assembled from another template. We can thus insert into a template insertion point a content which is itself rendered according to a lower level template.

44.6 Repeated template application

The `<ui:composition/>` tag cannot be used to insert a content multiple times as the latest insertion will replace the previous insertion.

If we want to have similar content to be inserted multiple times into a page, we may want to be able to define that each insertion should conform to a template. For this purpose the `<ui:decorate/>` tag is used.

In our example we have an `auxillaryContent` region in our main template which is meant to host auxiliary information to the main information conveyed with the page. It can, for example, be used to render news items of our weather bureau. Each auxiliary content item, however, has a header and a body and we may want to define a template on how an auxiliary content item is to be rendered. To this end we would define an `auxillaryContentTemplate`:

```

1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:f="http://java.sun.com/jsf/core"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8   <head>
9     <title>Auxillary content</title>
10  </head>
11  <body>
12    <ui:composition>

```

```

13 <div class="side_box">
14   <p class="auxillaryContentHeader">#{title}</p>
15
16   <ui:insert>Default body</ui:insert>
17 </div>
18 </ui:composition>
19 </body>
20 </html>

```

Here `#{title}` specifies a `title` parameter for our template.

Our page needs to allow us to insert multiple auxiliary content (news) items into the page. We supply each item with an `<ui:decorate/>` tag, specifying the title as a template parameter:

```

1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:f="http://java.sun.com/jsf/core"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8   <head>
9     <title>News</title>
10  </head>
11  <body>
12    <ui:composition>
13      <ui:decorate template="auxillaryContentTemplate.xhtml">
14        <ui:param name="title" value="Weather Buro improved connections to heavens"/>
15        <p>
16          The weather buro is proud to announce that its connections with the
17          heavens has been improved through a partnership relationship whose
18          details can unfortunately not be revealed.
19        </p>
20      </ui:decorate>
21      <ui:decorate template="auxillaryContentTemplate.xhtml">
22        <ui:param name="title" value="Rough seas predicted"/>
23        <p>
24          Certain unethical behaviour in some high-profile
25          people seems to have angered the heaven's and we
26          can expect a few weeks of very rough seas.
27        </p>
28      </ui:decorate>
29      <ui:decorate template="auxillaryContentTemplate.xhtml">
30        <ui:param name="title" value="Get sunshine into your life"/>
31        <p>
32          Various cultures have worshipped the sun. Notice that those
33          regions (Mexico, Brazil, Egypt, ...) have been blessed with it.
34          It is not sufficient to workshop the sun only on the beach in
35          December — get sunshine back into your life by worshipping
36          the sun every day ...
37        </p>
38      </ui:decorate>
39    </ui:composition>
40  </body>
41 </html>

```


Chapter 45

Reusable components

45.1 Overview

This chapter discusses a simple mechanism for developing reusable, fully functional components across levels of granularity. For this we need

- recursive assemblies of view components together with the backing beans which manage the state of the view components,
- the lower level backing beans for lower level view components to be injected into the backing beans for the higher level view components which contain the lower level view components, and
- binding components which are managed beans which manage the presentation layer process across views and provide the binding to the services/business processes layer.

45.1.1 Business versus presentation layer processes

Business processes should not be encoded in the presentation layer, but should be defined within reusable services published by a services layer. This could be, for example, in the form of web services or stateless Java EE request processors (stateless session beans or message driven beans - both are stateless).

These services should be accessible through a variety of adapters including both, system and human adapters. System adapters could be web services adapters, RMI access channels or message queues. Human adapters would be human user interfaces which could be application or web clients.

Having the business process encoded in the services layer enables us to reuse the process logic across different user access channels.

For example, there would be a business process to process an order or an insurance claim. These processes should be encoded in the services/business process layer and not in the presentation layer. The business process would be triggered by receiving an appropriate request object (e.g. a *processClaimRequest* or a `processOrderRequest` which would contain the claim and the order respectively in addition to potentially some further meta-data for the request).

However, there may still be presentation layer processes which would, for example, assemble an order or a claim across different screens, potentially requesting in-between information from the services layer (e.g. the available products or product prices). The purpose of these processes

is to assemble the appropriate request object. This is indeed a responsibility of the adapter (e.g. the presentation layer). In JSF these processes would typically be encoded within managed or backing beans whose responsibility it is to manage the presentation layer process.

Thus, in addition to having backing beans for views, we also have backing beans for presentation layer processes, i.e. the controllers for those processes.

45.1.2 Assembling higher level views from lower level views

We can easily aggregate composite views from lower level view components. However, each component has an associated backing bean which manages the state for that component. The backing beans for the higher level view components needs to extract the lower level state components from the lower level backing beans in order to assemble higher level data structures representing the composite views. To this end the lower level binding components need to be injected into the binding components of the higher level views.

45.1.3 Binding components controlling presentation layer processes

The backing beans themselves purely manage the data for the associated views. Higher level managed beans act as *binding components*

- binding across consecutive views into a process which assemble service request objects used to request services from the services/business process layer,
- requesting the service the user requires from the business process layer, and
- rendering the response of the service in a form required by the user.

In order to do this we need to

- have the backing beans for views injected into the binding components (i.e. managed beans which manage presentation layer processes), and
- pass data onto the binding component of a next view in order to have the next view populated with the appropriate data.

45.2 Defining reusable components across levels of granularity

Composite presentation layer components are recursively assembled from view/backing-bean pairs with lower level views and backing beans inserted into aggregate views and backing beans.

45.2.1 Composite views

To assemble higher level view components from lower level view components is straight-forward. We simply include them in the appropriate places of the higher level components.

For example, we can

1. insert the `addWeatherReadingPanel` into the content of a page which is based on a partially populated template


```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:ui="http://java.sun.com/jsf/facelets"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:f="http://java.sun.com/jsf/core"
7     xmlns:p="http://primefaces.prime.com.tr/ui">
8
9     <h:head>
10         <meta http-equiv="Content-Type" content="text/html;
11             charset=iso-8859-1" />
12         <title>Add weather reading</title>
13     </h:head>
14
15     <h:body>
16         <ui:composition template="/faces/partiallyPopulatedTemplate.xhtml">
17
18             <ui:define name="content">
19                 <ui:include src="addWeatherReadingPanel.xhtml" />
20             </ui:define>
21         </ui:composition>
22     </h:body>
23 </html>

```

2. The `addWeatherReadingPanel` adds to the weather reading panel containing the fields for the weather reading the button through which services are requested

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <fieldset xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:ui="http://java.sun.com/jsf/facelets"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:f="http://java.sun.com/jsf/core"
7     xmlns:p="http://primefaces.prime.com.tr/ui">
8
9     <legend>WeatherReading</legend>
10
11     <!-- The next line forces initialization of the backing bean for this aggregate view -->
12     <h:inputHidden value="#{weatherReadingBackingBean.dummy}" />
13
14     <ui:include src="/faces/locations/locationSelector.xhtml" />
15     <ui:include src="/faces/dateTime/dateTimePanel.xhtml" />
16     <ui:include src="weatherDetailsPanel.xhtml" />
17
18 </fieldset>

```

3. The `weatherReadingPanel` is, in turn, assembled from a location selector, a date-time panel and a weather details panel which are the leave panel capturing the core information components required for a request to capture a new weather reading. For example, the weather details panel looks as follows:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <fieldset xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:ui="http://java.sun.com/jsf/facelets"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:f="http://java.sun.com/jsf/core"
7     xmlns:p="http://primefaces.prime.com.tr/ui">
8
9     <legend>Weather details</legend>
10
11     <h:panelGrid columns="2">
12
13         <h:outputLabel id="temperatureLabel" for="temperatureField"
14             value="#{msgs.temperature}" />
15         <h:inputText id="temperatureField"
16             value="#{weatherDetailsBackingBean.temperature}" />

```

```

17
18     <h:outputLabel id="humidityLabel" for="humidityField"
19         value="#{msgs.humidity}"/>
20     <h:inputText id="humidityField"
21         value="#{weatherDetailsBackingBean.humidity}"/>
22
23     <h:outputLabel id="ambianceLabel" for="ambianceField"
24         value="#{msgs.ambiance}"/>
25     <h:selectOneMenu id="ambianceField"
26         value="#{weatherDetailsBackingBean.ambiance}">
27         <f:selectItems
28             value="#{weatherDetailsBackingBean.ambianceValues}"/>
29     </h:selectOneMenu>
30
31 </h:panelGrid>
32
33 </fieldset>

```

Note: *If we need some flexibility around the components which are inserted, we can use the JSF templating mechanism.*

45.2.2 Composite backing beans

We need to mirror the recursive aggregation of views by recursive aggregation of their corresponding backing beans. To this end we need to be able to inject the backing beans of lower level views into the higher level backing beans.

To this end we need to

- inject the lower level backing beans as managed properties into higher level backing beans via
- add recursively higher level getters and setters which assemble/disassemble higher level value objects from lower level value objects obtained from/provided to the lower level backing beans, and
- add to composite views a hidden field which is linked to a dummy attribute of the composite view's backing bean in order to ensure that its backing bean is loaded.

For example,

1. The `weatherReadingBackingBean` manages the view of the weather reading panel which I assembled from a location selector, a date-time panel and a weather details panel. The weather reading backing bean thus needs to have the backing beans for those lower level view components injected as managed properties:

```

1 package za.co.solms.weather;
2
3 import java.io.Serializable;
4
5 import javax.faces.bean.ManagedBean;
6 import javax.faces.bean.ManagedProperty;
7 import javax.faces.bean.RequestScoped;
8
9 import za.co.solms.dateTime.DateTimeBackingBean;
10 import za.co.solms.locations.LocationSelectorBackingBean;
11
12 @ManagedBean
13 @RequestScoped
14 public class WeatherReadingBackingBean implements Serializable
15 {
16     public WeatherReadingBackingBean() {}

```

```

17     public WeatherReading getWeatherReading()
18     {
19         WeatherReading weatherReading
20             = new WeatherReading(locationSelectorBackingBean.getLocation(),
21                                 dateTimeBackingBean.getDateTime(),
22                                 weatherDetailsBackingBean.getTemperature(),
23                                 weatherDetailsBackingBean.getHumidity(),
24                                 weatherDetailsBackingBean.getAmbiance());
25         return weatherReading;
26     }
27
28
29     public void setWeatherReading(WeatherReading weatherReading)
30     {
31         if (weatherReading.getLocation() != null)
32             locationSelectorBackingBean.setLocationName(weatherReading.getLocation().getName());
33         dateTimeBackingBean.setDateTime(weatherReading.getDateTime());
34         weatherDetailsBackingBean.setTemperature(weatherReading.getTemperature());
35         weatherDetailsBackingBean.setHumidity(weatherReading.getHumidity());
36         weatherDetailsBackingBean.setAmbiance(weatherReading.getAmbiance());
37     }
38
39     public LocationSelectorBackingBean getLocationSelectorBackingBean()
40     {
41         return locationSelectorBackingBean;
42     }
43
44     public void setLocationSelectorBackingBean(
45         LocationSelectorBackingBean locationSelectorBackingBean)
46     {
47         this.locationSelectorBackingBean = locationSelectorBackingBean;
48     }
49
50     public DateTimeBackingBean getDateTimeBackingBean()
51     {
52         return dateTimeBackingBean;
53     }
54
55     public void setDateTimeBackingBean(DateTimeBackingBean dateTimeBackingBean)
56     {
57         this.dateTimeBackingBean = dateTimeBackingBean;
58     }
59
60     public WeatherDetailsBackingBean getWeatherDetailsBackingBean()
61     {
62         return weatherDetailsBackingBean;
63     }
64
65     public void setWeatherDetailsBackingBean(WeatherDetailsBackingBean weatherDetailsBackingBean)
66     {
67         this.weatherDetailsBackingBean = weatherDetailsBackingBean;
68     }
69
70     public boolean isDummy() {return dummy;}
71
72     public void setDummy(boolean dummy)
73     {
74         this.dummy = dummy;
75     }
76
77     private boolean dummy;
78
79     @ManagedProperty(value="#{locationSelectorBackingBean}")
80     private LocationSelectorBackingBean locationSelectorBackingBean;
81
82     @ManagedProperty(value="#{dateTimeBackingBean}")
83     private DateTimeBackingBean dateTimeBackingBean;
84
85     @ManagedProperty(value="#{weatherDetailsBackingBean}")
86     private WeatherDetailsBackingBean weatherDetailsBackingBean;
87 }

```

The getters/setters of the higher level backing bean obtain the state from / pass the state

onto the backing beans of the lower level components. Note that we need to ensure that this backing bean is indeed loaded. This is only done if it is used by the view. To this end we insert a dummy hidden input field into the view which is bound to a dummy property of the backing bean:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <fieldset xmlns="http://www.w3.org/1999/xhtml"
4   xmlns:ui="http://java.sun.com/jsf/facelets"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:p="http://primefaces.prime.com.tr/ui">
8
9   <legend>WeatherReading</legend>
10
11 <!-- The next line forces initialization of the backing bean for this aggregate view -->
12   <h:inputHidden value="#{weatherReadingBackingBean.dummy}"/>
13
14   <ui:include src="/faces/locations/locationSelector.xhtml"/>
15   <ui:include src="/faces/dateTime/dateTimePanel.xhtml"/>
16   <ui:include src="weatherDetailsPanel.xhtml"/>
17
18 </fieldset>

```

45.3 Binding beans

Binding beans are managed beans which

- manage presentation layer processes which assembles the information required for the request object of a service required by the user - this process may be across multiple views,
- construct the request object for the service to be requested from the services layer and request the service from the services layer,
- receive the result from the services layer and render the result to the user.

To this end we have a binding bean which is a managed bean which

- gets the stateless session bean or the web service for the business service in the services/business processes layer injected, via the `@EJB` and `@WebServiceRef` annotations respectively,
- gets the backing bean for the view used to construct request view injected as a managed property via the `@ManagedProperty` annotation,
- has a action handler for the command component (e.g. button) through which the user requests the service within which the request object is assembled, the service is requested from the services layer and the result view is brought up by returning a relative path to it.

For example, below we show the `AddWeatherReadingBinding` which

- gets the `WeatherServices` stateless session bean and the `WeatherReadingBackingBean` injected,
- extracts the weather reading from the backing bean for the weather reading panel,
- requests the `persistWeatherReading` service from the stateless session bean, and
- pushes an informational message into the faces context, informing the user that the weather reading was successfully stored,
- returns the current page in order to capture the next weather reading.

45.4 Forwarding information to managed beans

At times one needs to forward information from one managed bean to another. For example, assume you have a `showPersons` service which shows a list of persons in, say, a table, and the user wants to select a person and edit it.

In this case the `showPersonsBinding` needs to forward the selected person (i.e. the person to edit) to `editPersonBinding` which is the presentation layer controller for the `editPerson` service. This can be done via the request map which is accessible via the faces context:

```

1 public class ShowPersonsBinding
2 {
3     public String editPerson()
4     {
5         ...
6         FacesContext facesContext = FacesContext.getCurrentInstance();
7         facesContext.getExternalContext().getRequestMap().put
8             ("personToEdit", person);
9
10        logger.info("***** Have put personToEdit in request map: " + person);
11
12        return "editPerson";
13    }
14
15    ...
16 }
```

We can annotate some initialization service in the `EditPersonBinding` class with `@PostConstruct` within which the person to edit is extracted from the request map:

```

1 @ManagedBean
2 @ViewScoped
3 public class EditPersonBinding implements Serializable
4 {
5     public EditPersonBinding() {}
6
7     /**
8     * Upon having created this managed bean, extract the personToEdit
9     * from the request map and insert it into the personDetailsBinding.
10    * Note that this binding object will only be loaded if aspects are
11    * required by its View.
12    * In this case the view retrieves the edited person from this binding
13    * component which, in turn, requests it from the embedded
14    * personDetailsBinding which, in turn, was initialized by this binding
15    * objects postConstruct initializer (i.e.\ this method).
16    */
17    @PostConstruct
18    public void initialize()
19    {
20        {
21            FacesContext fc = FacesContext.getCurrentInstance()
22            personToEdit = (Person)
23                fc.getExternalContext().getRequestMap().get("personToEdit");
24            ...
25        }
26        ...
27    }
```


Chapter 46

Ajax

46.1 Overview

AJAX, the *Asynchronous JavaScript And XML* framework through which light-weight, more dynamic web-based user interfaces are supported. *AJAX* does not provide a fully implemented framework, but rather represents an approach of doing light-weight, dynamic web front ends using a combination of base technologies including

- XML,
- XHTML and its underlying DOM,
- JavaScript, and
- XMLHttpRequest

AJAX is a general framework which can be supported across different server side technologies. *JavaEE* explicitly supports *AJAX* through an *AJAX* infrastructure which support making *AJAX* requests through special *AJAX* tags.

46.2 What does the AJAX framework provide?

The JSF *AJAX* framework provides

- a standard interface to
 - make *AJAX* requests, and to
 - process *AJAX* responses,
- an optimized *JSF* life-cycle which caters for
 - partial request processing, and
 - partial view rendering.

The framework enables one to, from a page, make a light-weight synchronous request for some small amount of information which is used to update the page. In the context of *JSF* this will be in the form of

- a request to render some element on a page with
- the data ultimately being obtained through the binding to the JSF backing bean which
- may or may not make a further request to the services layer in order to obtain some required information.

46.3 How does AJAX work?

AJAX is a framework where

1. XHTML components make JavaScript service requests to a AJAX function,
2. the AJAX component (JavaScript) running in the browser makes an XML/HTTP request to the server requesting some low level data or functionality and specifying which aspect of the page should be re-rendered, i.e. without requesting a new XHTML page,
3. the server side makes the appropriate bindings to the binding components, mapping information requests to requests on the binding components,
4. The server side Ajax support returns the fine grained data as XML to the Ajax component, and
5. requests the browser to redraw aspects of the current page.

46.4 JSF Ajax support

As of JSF 2, JSF has full built-in support for AJAX. The support is in the form of both, a JSF-AJAX JavaScript library and server side support for processing AJAX requests.

46.4.1 JSF AJAX JavaScript Library

46.4.1.1 The JSF-Ajax JavaScript library

JSF provides a JSF-AJAX JavaScript library which provides the following services to the client:

- **jsf.ajax.request(source, event, options):** This is the core AJAX service which makes an AJAX request to the server.
- **jsf.ajax.response(request, context):** This is a service used internally by the framework to process the response obtained by the server - it is not directly used.
- **jsf.ajax.addOnEvent(callBack):** This method is used mainly internally to register call-back functions used to process AJAX events.
- **jsf.ajax.addOnError(callBack):** This method is used mainly internally to register call-back functions used to process any errors encountered when processing the AJAX request.

46.4.2 Server-Side Processing of AJAX Request

The standard JSF request processing life cycle is defined in such a way that the full life cycle need not be followed for all components in the component tree (and hence in the view). For example, the `immediate` on a component requests that the full life cycle is followed for that component, but that the life cycle for all other components is short-circuited to include only the rendering phase.

The same is done for AJAX requests. The parameters of the AJAX request will determine for which components the full life cycle is executed and for which components the short-circuited life cycle is used.

46.5 Partial AJAX Request Life Cycle

An AJAX request is meant to be a light-weight request which does not require

- the processing of the full page (i.e. all components contained on the page), or
- the re-rendering of the full page.

The standard JSF request life cycle is split up into two core components called the processing and rendering stages of the request life cycle. The first 5 phases up to *processApplication* which is used to process service requests to the services layer are grouped into the processing stage whilst the last phase is the rendering phase.

AJAX supports partial processing and partial rendering. For an AJAX request one can thus specify

- which components from the current should be processed, and
- which components from the current page should be re-rendered.

46.6 The JSF JavaScript API for AJAX

JSF specifies a JavaScript library which provides the client side API to AJAX functionality. AJAX requests can be requested either

- directly through calling the JSF JavaScript API for Ajax, or
- implicitly though the `<ajax>` tag contained in the JSF core library.

The API is directly accessible through the JSF core library. Hence AJAX requests can be directly embedded into any facelet page:

```

1 <h:form xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets" xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core" xmlns:p="http://primefaces.prime.com.tr/ui">
2   ...
3   <f:ajax ... />

```

Alternatively you can make direct AJAX JavaScript requests within your page using the normal HTML attributes like `onChange`:

```

1 <h:selectOne ... onChange="jsf.ajax.request(sourceElement, event, options)"> ... </h:selectOne>

```

46.6.1 The JSF ajax tag

Any JSF component can be given Ajax support by adding a `<f:ajax/>` tag. The component containing the `<f:ajax/>` tag is the source of the AJAX request and is the component for which partial processing is executed. By default it is also the element for which partial rendering is done, though the latter can be specified through the `render` attribute of the `<ajax/>` tag.

46.6.1.1 The JSF ajax tag attributes

The AJAX tag attributes allow you to specify the details of the AJAX request. They include

- **execute::** This attribute contains a space separated list of component ids specifying the components which should be processed during the partial processing of the AJAX request. Instead of specifying the actual component IDS you can also specify component types via standard attributes like `@all`, `@form` or `@this`. **Note:** *If this parameter is not specified then the component containing the `ajax` tag is processed, i.e. the default value is `@this`.*
- **render::** The render attribute specifies which component(s) should be re-rendered after the processing of the AJAX request. It should contain a space-separated list of component ids specifying the components which should be re-rendered during the partial rendering phase. Alternatively one can use one of the predefined constants including `@all`, `@form`, `@this` and `@none`. **Note:** *The default value for this attribute is `@none`, i.e. by default nothing is re-rendered.*
- **event::** The event type which should trigger the AJAX request. This could, for example, be `action` to specify that the AJAX request should be executed on the action event for the component or `valueChange` if the AJAX request should be triggered by a value change of the component. **Note:** *The default event is the default event for the component, i.e. `action` for buttons or links and `valueChange` for selection and input components.*
- **onevent::** This parameter is used to specify a JavaScript callback function which should be called upon receipt of the AJAX response.
- **onError::** This parameter is used to specify a JavaScript callback function which should be called upon receiving an AJAX error.
- **disabled::** This parameter is used to control whether AJAX processing is switched on or off.

46.7 Using an AJAX request

Assume that we want to show a view containing a list box with all the location names and selecting (clicking on) a particular location should populate a locations detail panel on the page which would show the details for only that particular location.

To this end we would add a AJAX request which is executed on the `onChange` event which requests the re-rendering of just the location-details panel, not the data for the entire page, but only the data for that particular component:

```

1 <h:selectOneListbox id="locationNameSelectorListbox"
2   value="#{ajaxLocationsBinding.locationName}" >
3   <f:selectItems value="#{ajaxLocationsBinding.locationNames}" />
4   <f:ajax render="@this locationDetails" />
5 </h:selectOneListbox>

```

Note that we also have a value change listener which updates the location on the managed bean itself.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <h:form xmlns="http://www.w3.org/1999/xhtml"
4   xmlns:ui="http://java.sun.com/jsf/facelets"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:p="http://primefaces.prime.com.tr/ui">
8
9   <h2>Ajax Locations</h2>
10
11   <h:messages style="color:red" />
12
13   <fieldset>
14     <legend>Locations</legend>
15
16     <h:selectOneListbox id="locationNameSelectorListbox"
17       value="#{ajaxLocationsBinding.locationName}">
18       <f:selectItems value="#{ajaxLocationsBinding.locationNames}" />
19       <f:ajax render="@this locationDetails" />
20     </h:selectOneListbox>
21   </fieldset>
22
23   <fieldset>
24     <legend>Location details</legend>
25     <h:panelGroup>
26       <h:panelGrid columns="2" id="locationDetails">
27         <h:outputLabel id="nameLabel"
28           for="nameField" value="#{msgs.name}" />
29         <h:outputText id="nameField"
30           value="#{ajaxLocationsBinding.location.name}" />
31         <h:outputLabel id="addressLabel"
32           for="addressField" value="#{msgs.address}" />
33         <h:inputTextarea rows="5" id="addressField"
34           value="#{ajaxLocationsBinding.location.address}" />
35       </h:panelGrid>
36
37       <fieldset>
38         <legend>Geographic coordinates</legend>
39         <h:panelGrid columns="2">
40           <h:outputLabel id="longitudeLabel" for="longitudeField"
41             value="#{msgs.longitude}" />
42           <h:outputText id="longitudeField"
43             value="#{ajaxLocationsBinding.location.geographicLocation.longitude}" />
44           <h:outputLabel id="latitudeLabel" for="latitudeField"
45             value="#{msgs.latitude}" />
46           <h:outputText id="latitudeField"
47             value="#{ajaxLocationsBinding.location.geographicLocation.latitude}" />
48         </h:panelGrid>
49       </fieldset>
50     </h:panelGroup>
51   </fieldset>
52
53   <h:panelGrid columns="3">
54     <h:commandButton id="refreshPage" value="#{msgs.refreshPage}"
55       actions="#{ajaxLocationsBinding.refreshPage()}" />
56   </h:panelGrid>
57
58 </h:form>

```

The corresponding binding bean has nothing new:

```

1 package za.co.solms.locations;
2
3 import java.io.Serializable;
4 import java.util.List;
5 import java.util.logging.Logger;
6

```

```

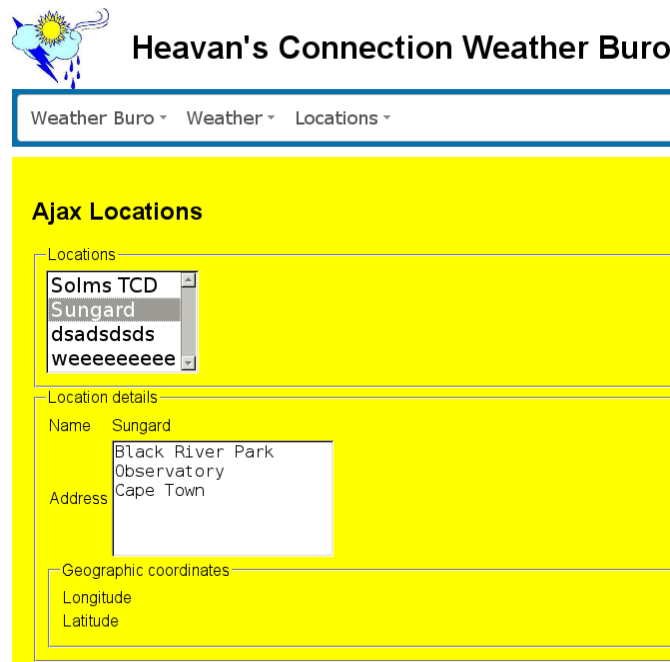
7 import javax.ejb.EJB;
8 import javax.faces.application.FacesMessage;
9 import javax.faces.bean.ManagedBean;
10 import javax.faces.bean.ViewScoped;
11 import javax.faces.context.FacesContext;
12
13 import za.co.solms.location.Location;
14 import za.co.solms.location.LocationServices;
15 import za.co.solms.location.LocationServices.NoSuchLocationException;
16
17 @ManagedBean
18 @ViewScoped
19 public class AjaxLocationsBinding implements Serializable
20 {
21     public AjaxLocationsBinding() {}
22
23     public List<String> getLocationNames()
24     {
25         logger.info("Getting location names");
26         return locationServices.getAllLocationNames();
27     }
28
29     public void changeLocationName()
30     {
31         logger.info("changing location name to " + locationName);
32     }
33
34     public String refreshPage()
35     {
36         return "";
37     }
38
39     public String getLocationName()
40     {
41         return locationName;
42     }
43
44     public void setLocationName(String locationName)
45     {
46         logger.info("Setting location name to " + locationName);
47         this.locationName = locationName;
48     }
49
50     public Location getLocation()
51     {
52         try
53         {
54             return locationServices.getLocationByName(locationName);
55         }
56         catch (NoSuchLocationException e)
57         {
58             FacesMessage msg = new FacesMessage
59                 ("No such location – should not have happened");
60             FacesContext.getCurrentInstance().addMessage(null, msg);
61             return null;
62         }
63     }
64
65     private List<String> locationNames;
66
67     private String locationName;
68
69     @EJB
70     private LocationServices locationServices;
71
72     private Logger logger
73         = Logger.getLogger(AjaxLocationsBinding.class.getName());
74 }

```

46.8 Example

Consider a panel where the details of a location should be shown for the location selected in a list of locations. One would not want the full processing of the component tree for the page (only really the selected location) and similarly, the entire page need not be re-rendered - one only needs to populate the fields containing the details of the location.

Figure 46.1: A facelet which benefits from partial processing and rendering facilitated by AJAX



Heaven's Connection Weather Buro

Weather Buro ▾ Weather ▾ Locations ▾

Ajax Locations

Locations

Solms TCD
Sungard
dsadsdsds
weeeeeeeee

Location details

Name Sungard

Address Black River Park Observatory
Cape Town

Geographic coordinates

Longitude
Latitude

46.8.1 The facelet with AJAX requests

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/
  DTD/xhtml1-transitional.dtd">
2 <h:form xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core"
6   xmlns:p="http://primefaces.org/ui">
7
8   <h2>Ajax Locations</h2>
9
10   <h:messages style="color:red" />
11
12   <fieldset>
13     <legend>Locations</legend>
14
15     <h:selectOneListbox id="locationNameSelectorListbox" value="#{ajaxLocationsBinding.locationName
16       }">
17       <f:selectItems value="#{ajaxLocationsBinding.locationNames}" />
18       <f:ajax render="@this locationDetails" />
19     </h:selectOneListbox>
20   </fieldset>

```

```

21 <fieldset>
22 <legend>Location details</legend>
23 <h:panelGroup>
24 <h:panelGrid columns="2" id="locationDetails">
25 <h:outputLabel id="nameLabel" for="nameField" value="#{msgs.name}"/>
26 <h:outputText id="nameField" value="#{ajaxLocationsBinding.location.name}"/>
27 <h:outputLabel id="addressLabel" for="addressField" value="#{msgs.address}"/>
28 <h:inputTextarea rows="5" id="addressField" value="#{ajaxLocationsBinding.location.address}"/>
29 </h:panelGrid>
30
31 <fieldset>
32 <legend>Geographic coordinates</legend>
33 <h:panelGrid columns="2">
34 <h:outputLabel id="longitudeLabel" for="longitudeField" value="#{msgs.longitude}"/>
35 <h:outputText id="longitudeField" value="#{ajaxLocationsBinding.location.geographicLocation.longitude}"/>
36 <h:outputLabel id="latitudeLabel" for="latitudeField" value="#{msgs.latitude}"/>
37 <h:outputText id="latitudeField" value="#{ajaxLocationsBinding.location.geographicLocation.latitude}"/>
38 </h:panelGrid>
39 </fieldset>
40 </h:panelGroup>
41 </fieldset>
42
43 <h:panelGrid columns="3">
44 <h:commandButton id="refreshPage" value="#{msgs.refreshPage}"
45 actions="#{ajaxLocationsBinding.refreshPage()}" />
46 </h:panelGrid>
47
48 </h:form>

```

46.8.2 The supporting backing bean

```

1 package za.co.solms.locations;
2
3 import java.io.Serializable;
4 import java.util.List;
5 import java.util.logging.Logger;
6
7 import javax.ejb.EJB;
8 import javax.faces.application.FacesMessage;
9 import javax.faces.bean.ManagedBean;
10 import javax.faces.bean.RequestScoped;
11 import javax.faces.bean.ViewScoped;
12 import javax.faces.context.FacesContext;
13
14 import za.co.solms.location.Location;
15 import za.co.solms.location.LocationServices;
16 import za.co.solms.location.LocationServices.NoSuchLocationException;
17
18 @ManagedBean
19 @RequestScoped
20 public class AjaxLocationsBinding implements Serializable
21 {
22     public AjaxLocationsBinding() {}
23
24     public List<String> getLocationNames()
25     {
26         logger.info("Getting location names");
27         return locationServices.getAllLocationNames();
28     }
29
30     public void changeLocationName()
31     {
32         logger.info("changing location name to " + locationName);
33     }
34
35     public String refreshPage()
36     {
37         return "";
38     }
39 }

```

```
38 }
39
40 public String getLocationName()
41 {
42     return locationName;
43 }
44
45 public void setLocationName(String locationName)
46 {
47     logger.info("Setting location name to " + locationName);
48     this.locationName = locationName;
49 }
50
51 public Location getLocation()
52 {
53     try
54     {
55         return locationServices.getLocationByName(locationName);
56     }
57     catch (NoSuchLocationException e)
58     {
59         FacesMessage msg = new FacesMessage("No such location – should not have happened");
60         FacesContext.getCurrentInstance().addMessage(null, msg);
61         return null;
62     }
63 }
64
65 private List<String> locationNames;
66
67 private String locationName;
68
69 @EJB
70 private LocationServices locationServices;
71
72 private Logger logger = Logger.getLogger(AjaxLocationsBinding.class.getName());
73 }
```


Part VII

JAX-RS

Chapter 47

Overview

47.1 Introduction

RESTful (Representative State Transfer) web services has gained a lot of support as a light-weight alternative to the more heavy-weight contract-based web services which generally use SOAP. JAX-RS is the standard for RESTful web services specified as a set of annotations in JSR-311 titled "*JAX-RS: The Java API for RESTful Web Services*".

47.2 Core Principles

- **Standard web API:** RESTful web services are accessed through the standard web services, GET, POST, PUT and DELETE. They support the standard semantics of the web including bookmarking, resource caching, and hyperlinks.
- **Everything is an addressable resource:** Both data and functionality are considered resources which are accessed through a *Uniform Resource Identifier* (URI). A consequence of this is that all domain objects will acquire a globally unique URI.
- **Self-describing, flexible messages:** Resources are decoupled from their representation and the content can potentially be accessed through a variety of formats including XML, JSON, PDF, etc.
- **Stateless interactions:** Interactions are stateless resulting in individual requests being treated in their own context. Any state which needs to be maintained across service requests needs to be either transferred to and fro between the parties or is maintained within the environment.

47.3 Java Rest Frameworks

The reference implementation for JAX-RS is *Jersey*. *Jersey* is, however, a production quality framework which can be deployed into an application server. As of Java-EE 7 JAX-RS is part of the Java-EE spec.

Chapter 48

Introduction

JAX-RS is an API based on a set of annotations which makes it easy to publish services as RESTful web services without having to code aspects of the architecture into the business logic of the services themselves. **Note:** *This keeps the plumbing (infrastructure) logic separate from the functional or business logic.*

Chapter 49

JAX-RS Resources

A JAX-RS resource needs to be accessible via a URI. The resource can be either a information/-data resource or a service.

49.1 Paths

49.1.1 How is the absolute resource path assembled?

The Resource URI is specified as a relative path via the `@Path` annotation which is an absolute path is assembled which is assigned to the Java resource class.

The absolute path is assembled from

1. the URI of the server on which the resource is deployed,
2. the context root of the WAR file,
3. the URL pattern to which the Jersey helper servlet responds, and
4. the relative path specified for the resource.

49.1.2 Basic paths

A basic path is a simple relative path. For example

```
1 @Path("/echo")
2 public class EchoResource
3 {
4     ...
5 }
```

specifies the relative path for the `EchoResource` as `/echo`.

If the resource is deployed

- onto a server whose URL is `http://www.solms.co.za`,
- with context root for the web application set to `/myApps`
- and the servlet mapping for the the Jersey web application set to `/*`

then the resultant absolute URL is

```
1 http://www.solms.co.za/myApps/echo
```

49.1.3 Embedded path variables

One can embed path variables in the resource path. For example,

```
1 @Path("/products/{productCode}")
2 public class ProductResource
3 {
4     ...
5     @Get
6     @Produces("text/xml")
7     public String getProductDetails
8         (@PathParam("productCode") String productCode)
9     {
10         ...
11     }
12 }
```

will lead to paths like

```
1 http://vendorX.com/products/jam1434
```

Note the `@PathParam` mapping annotation for the `productCode` method parameter.

49.2 HTTP method support

REST-based web services use the standard HTTP resource management API. The HTTP method processed by a Java method in the Java resource class is specified by annotating the Java method with one of

- **@GET:** A service which returns the state of a resource, leaving the resource unchanged and hence being a potential candidate for caching.
- **@POST:** A service which may potentially modify a resource maintained by the server.
- **@PUT:** A service which adds a resource to the server.
- **@DELETE:** A service which removes a resource from the server.

Chapter 50

Supported inputs and outputs

The `@Consumes` and `@Produces` annotations are used to specify the input and output MIME types for a resource.

For example, in the `EchoResource` shown below, the two `@GET` implementations produce plain text and XML respectively.

```
1 package za.co.solms.training.jaxrs.echo;
2
3 import javax.ws.rs.DefaultValue;
4 import javax.ws.rs.GET;
5 import javax.ws.rs.Produces;
6 import javax.ws.rs.Path;
7 import javax.ws.rs.QueryParam;
8
9 // The Echo resource is hosted at the relative URI "/echo"
10 @Path("/echo")
11 public class EchoResource
12 {
13     // Processes HTTP GET requests which accept text/plain
14     @GET
15     @Produces("text/plain")
16     public String echoAsTextMessage
17     (@DefaultValue("Mollo") @QueryParam("msg") String message)
18     {
19         return "Echo: " + message;
20     }
21
22     // Processes HTTP GET requests which accept application.xml
23     @GET
24     @Produces("application/xml")
25     public ResponseMessage echoAsXmlMessage
26     (@DefaultValue("Mollo") @QueryParam("msg") String message)
27     {
28         return new ResponseMessage("Echo:", message);
29     }
30 }
```

A service can also consume and produce XML. For example

```
1 package za.co.solms.jaxrs.locations;
2
3 import java.io.IOException;
4 import java.net.URI;
5 import java.util.LinkedList;
6 import java.util.List;
7 import java.util.logging.Logger;
8
9 import javax.ejb.EJB;
```

```

10 import javax.ws.rs.Consumes;
11 import javax.ws.rs.DELETE;
12 import javax.ws.rs.GET;
13 import javax.ws.rs.PUT;
14 import javax.ws.rs.Path;
15 import javax.ws.rs.PathParam;
16 import javax.ws.rs.Produces;
17 import javax.ws.rs.core.Context;
18 import javax.ws.rs.core.Response;
19 import javax.ws.rs.core.UriBuilder;
20 import javax.ws.rs.core.UriInfo;
21
22 import za.co.solms.location.Location;
23 import za.co.solms.location.LocationServices;
24 import za.co.solms.location.LocationServices.LocationNameInUseException;
25 import za.co.solms.location.LocationServices.NoSuchLocationException;
26
27 @Path("/locations")
28 public class LocationResource
29 {
30     public LocationResource() {}
31
32     @Path("/{locationName}")
33     @Produces("application/xml")
34     @GET
35     public Object getLocation(@PathParam("locationName")
36         String locationName)
37     {
38         logger.info("*** Requesting location " + locationName);
39         try
40         {
41             return locationServices.getLocationByName(locationName);
42         }
43         catch (NoSuchLocationException e)
44         {
45             return Response.status(Response.Status.NOT_FOUND).build();
46         }
47     }
48
49     @Path("/add")
50     @Consumes("application/xml")
51     @Produces("application/xml")
52     @PUT
53     public Response createLocation(Location location) throws IOException
54     {
55         logger.info("*** Adding location " + location);
56         try
57         {
58             location = locationServices.persistLocation(location);
59
60             UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
61             URI locationUri = uriBuilder.path(location.getName()).build();
62
63             return Response.created(locationUri).entity(location).build();
64         }
65         catch (LocationNameInUseException e)
66         {
67             return Response.status(Response.Status.CONFLICT).build();
68         }
69     }
70
71     @Path("/{locationName}")
72     @DELETE
73     public Response removeLocation
74         (@PathParam("locationName") String locationName)
75     {
76         logger.info("*** Removing location " + locationName);
77
78         try
79         {
80             Location location
81                 = locationServices.getLocationByName(locationName);
82             locationServices.removeLocation(location);

```

```

83     }
84     catch (NoSuchLocationException e) {}
85
86     return Response.ok().build();
87 }
88
89 @EJB
90 private LocationServices locationServices;
91
92 @Context
93 UriInfo uriInfo;
94
95 private static final Logger logger
96     = Logger.getLogger(LocationResource.class.getName());
97 }

```

the `addLocation` service consumes and produces XML. The corresponding exchanged data objects must be instances of classes which have been annotated with `XMLRootElement`:

```

1 package za.co.solms.location;
2
3 import java.io.Serializable;
4
5 import javax.persistence.CascadeType;
6 import javax.persistence.Column;
7 import javax.persistence.Entity;
8 import javax.persistence.GeneratedValue;
9 import javax.persistence.Id;
10 import javax.persistence.NamedQueries;
11 import javax.persistence.NamedQuery;
12 import javax.persistence.OneToOne;
13 import javax.xml.bind.annotation.XmlRootElement;
14
15 /**
16  * Represents a location with a name, address and geographic location.
17  *
18  * @author fritz@solms.co.za
19  */
20
21 @XmlRootElement
22 @Entity
23 @NamedQueries({
24     @NamedQuery(name="findAllLocationNames",
25         query="select l.name from Location l"),
26     @NamedQuery(name="findAllLocations",
27         query="Select l from Location l"),
28     @NamedQuery(name="findLocationByName",
29         query="select l from Location l where l.name = :locationName")
30 })
31 public class Location implements Serializable
32 {
33     public Location() {}
34
35     public Location(String name, String address,
36         GeographicCoordinates geographicCoordinates)
37     {
38         this.name = name;
39         this.address = address;
40         this.geographicCoordinates = geographicCoordinates;
41     }
42
43     @Id
44     @GeneratedValue
45     public int getId()
46     {
47         return id;
48     }
49
50     public void setId(int id)
51     {
52         this.id = id;

```

```

53     }
54
55     @Column(unique=true, nullable=false)
56     public String getName()
57     {
58         return name;
59     }
60
61     public void setName(String name)
62     {
63         this.name = name;
64     }
65
66     public String getAddress()
67     {
68         return address;
69     }
70
71     public void setAddress(String address)
72     {
73         this.address = address;
74     }
75
76     @OneToOne(cascade=CascadeType.ALL)
77     public GeographicCoordinates getGeographicLocation()
78     {
79         return geographicCoordinates;
80     }
81
82     public void setGeographicLocation(GeographicCoordinates geographicCoordinates)
83     {
84         this.geographicCoordinates = geographicCoordinates;
85     }
86
87     public boolean equals(Object o)
88     {
89         try
90         {
91             Location arg = (Location)o;
92             return this.getName().equals(arg.getName())
93                 && this.getAddress().equals(arg.getAddress())
94                 && this.getGeographicLocation().equals(arg.getGeographicLocation());
95         }
96         catch (ClassCastException e)
97         {
98             return false;
99         }
100     }
101
102     public int hashCode()
103     {
104         return name.hashCode() + address.hashCode()
105             + geographicCoordinates.hashCode();
106     }
107
108     public String toString() {return name;}
109
110     private int id;
111     private String name, address;
112     private GeographicCoordinates geographicCoordinates;
113
114     private static final long serialVersionUID = 1L;
115 }

```

In our case we simply annotated an entity to be also an `XmlRootElement`.

A single method can also consume or produce multiple MIME types. The different acceptable MIME types are separated by commas.

```

1  @GET
2  @Produces("application/xml,application/xhtml+xml")
3  public PersonDetails getPersonDetails(...)

```

```
4 {  
5   ...  
6 }
```

50.1 How is the output type selected?

The MIME type the HTTP client (e.g. browser or your REST web service client) is able to consume is specified in the ACCEPT header element. Different output formats can be assigned relative preferences via a weighting (q) factor which is between 0 and 1 with the higher number given preference. supporting different content types

```
1 GET /sales/processOrder HTTP/1.1  
2 Host: vendorX.com  
3 Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;  
4 q=0.8,image/png;q=0.5
```

This specifies that the HTTP client is able to consume XML, XHTML, HTML, plain text and images with a preference for those with a higher weighting with the first two options having the highest rating of 1.

Chapter 51

Parameters

Parameters can be provided in a variety of ways in HTTP. All these standard ways are supported in JAX-RS.

51.1 Path Parameters

One simple and often intuitive way to specify a parameter is to include it in the actual path (URI). For example, one could use

```
1 http://www.people.org/contacts/jimSmith01
```

where the last part of the path is used as a parameter to a JAX-RS web service.

For example

```
1 package za.co.solms.jaxrs.locations;
2
3 import za.co.solms.location.LocationServices;
4 ...
5
6 @Path("/locations")
7 public class LocationResource
8 {
9     public LocationResource() {}
10
11     @Path("/{locationName}")
12     @Produces("application/xml")
13     @GET
14     public Location getLocation(@PathParam("locationName") String locationName)
15     {
16         logger.info("*** Requesting location " + locationName);
17         try
18         {
19             return locationServices.getLocationByName(locationName);
20         }
21         catch (NoSuchLocationException e)
22         {
23             return null;
24         }
25     }
26 }
```

we specify that the `getLocation` service is resolved from the relative URI

```
1 /locations/{locationName}
```

and we bind the path parameter to the `locationName` method parameter.

A particular location can now be queried through a URI like

```
1 http://localhost:8080/weatherBuro/jaxrs/locations/pe
```

51.2 Request parameters

Parameters are often provided in a parameters list of the HTTP request. For example

```
1 http://localhost:8080/jax-rs/echo?msg=bonjour
```

supplies the `msg` as request parameter. It can be bound to method parameters via the `QueryParam` annotation:

```
1 public class EchoResource
2 {
3     @GET
4     @Produces("text/plain")
5     public String echoAsTextMessage
6         (@DefaultValue("Mollo") @QueryParam("msg") String message)
7     {
8         return "Echo: " + message;
9     }
10 }
```

51.3 Form parameters

Often data is provided by users by filling in forms. JAX-RS provides a mechanism to bind form parameters to method parameters.

For example, in the following method we bind the two method parameters to two form parameters:

```
1 /**
2  * Returns the contact details of all persons within
3  * a lexicographical name range.
4  */
5 @POST
6 @Consumes("application/x-www-form-urlencoded")
7 public ContactDetails getContactDetails
8     (@FormParam("from") String nameLowerBound,
9     @FormParam("to") String nameUpperBound)
10 {
11     ...
12 }
```

Alternatively one can also obtain a general map of form parameters

```
1 @POST
2 @Consumes("application/x-www-form-urlencoded")
3 public void processOrder(MultivaluedMap<String, String> formParams)
4 {
5     ...
6 }
```


51.4 Cookie parameters

Another common method to transfer information is to use cookies which are small data objects stored in the client browser. They are typically used to maintain information across sessions like user identity and preferences.

Cookies are also used to maintain and transmit process ids, enabling the server side to tie up different messages received to the same process.

For example,

```
1  /**
2   * Returns the contact details of all persons within
3   * a lexicographical name range.
4   */
5   @POST
6   @Consumes("application/x-www-form-urlencoded")
7   public Invoice confirmOrder
8       (@CookieParam("processId") String processId)
9   {
10      ...
11  }
```

51.5 Header parameters

Header parameters as well as cookies can be extracted from the `HttpHeaders` as follows:

```
1  @GET
2  public String get(@Context HttpHeaders hh)
3  {
4      MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
5      Map<String, Cookie> pathParams = hh.getCookies();
6  }
```

51.6 Default parameter values

JAX-RS allows one to specify default values for parameters which are used should the respective parameter not be provided:

```
1  @GET
2  @Produces("text/plain")
3  public String echoAsTextMessage
4      (@DefaultValue("Mollo") @QueryParam("msg") String message)
5  {
6      return "Echo: " + message;
7  }
```


Chapter 52

Responses

Responses are commonly provided by returning either a `String` or an instance of a class which is annotated as an `XmlRootElement`. The latter is used to return `XML` or `JSON`. **Note:** *A simple way of generating `JSON` is to return an `XmlRootElement` annotated object. The `JAX-RS` framework performs a default mapping between `XML` and `JSON`.*

52.1 Building URIs and Responses

At times one needs to

- return a standard HTTP response like 201 (created) or 404 (not found),
- add elements to the header of the response like a time stamp.

Response and URI building is illustrated in the following code extract

```
1 package za.co.solms.jaxrs.locations;
2
3 import java.io.IOException;
4 import java.net.URI;
5 import java.util.LinkedList;
6 import java.util.List;
7 import java.util.logging.Logger;
8
9 import javax.ejb.EJB;
10 import javax.ws.rs.Consumes;
11 import javax.ws.rs.DELETE;
12 import javax.ws.rs.GET;
13 import javax.ws.rs.PUT;
14 import javax.ws.rs.Path;
15 import javax.ws.rs.PathParam;
16 import javax.ws.rs.Produces;
17 import javax.ws.rs.core.Context;
18 import javax.ws.rs.core.Response;
19 import javax.ws.rs.core.UriBuilder;
20 import javax.ws.rs.core.UriInfo;
21
22 import za.co.solms.location.Location;
23 import za.co.solms.location.LocationServices;
24 import za.co.solms.location.LocationServices.LocationNameInUseException;
25 import za.co.solms.location.LocationServices.NoSuchLocationException;
26
27 @Path("/locations")
28 public class LocationResource
29 {
```

```

30 public LocationResource() {}
31
32 @Path("/{locationName}")
33 @Produces("application/xml")
34 @GET
35 public Object getLocation(@PathParam("locationName")
36     String locationName)
37 {
38     logger.info("*** Requesting location " + locationName);
39     try
40     {
41         return locationServices.getLocationByName(locationName);
42     }
43     catch (NoSuchLocationException e)
44     {
45         return Response.status(Response.Status.NOT_FOUND).build();
46     }
47 }
48
49 @Path("/add")
50 @Consumes("application/xml")
51 @Produces("application/xml")
52 @PUT
53 public Response createLocation(Location location) throws IOException
54 {
55     logger.info("*** Adding location " + location);
56     try
57     {
58         location = locationServices.persistLocation(location);
59
60         UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
61         URI locationUri = uriBuilder.path(location.getName()).build();
62
63         return Response.created(locationUri).entity(location).build();
64     }
65     catch (LocationNameInUseException e)
66     {
67         return Response.status(Response.Status.CONFLICT).build();
68     }
69 }
70
71 @Path("/{locationName}")
72 @DELETE
73 public Response removeLocation
74     (@PathParam("locationName") String locationName)
75 {
76     logger.info("*** Removing location " + locationName);
77
78     try
79     {
80         Location location
81             = locationServices.getLocationByName(locationName);
82         locationServices.removeLocation(location);
83     }
84     catch (NoSuchLocationException e) {}
85
86     return Response.ok().build();
87 }
88
89 @EJB
90 private LocationServices locationServices;
91
92 @Context
93 UriInfo uriInfo;
94
95 private static final Logger logger
96     = Logger.getLogger(LocationResource.class.getName());
97 }

```