Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Jiří Krůček

Support for Development of Enterprise Applications in Java

Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.
Study programme: Computer science

Prague 2009

ACKNOWLEDGEMENTS

I would like to thank my supervisor RNDr. Petr Hnětynka, Ph.D. for his invaluable help during writing this master thesis, and for his patient guidance during its many transformations. His careful editing contributed enormously to the production of the thesis. Above all, I would like to thank my family for their love and encouragement throughout a lengthy process of writing. This work would not have been possible without their constant support.

I declare that I have written this master thesis on my own and listed all the used sources. I agree with lending of the thesis.

In Prague on the 4th of August 2009 Jiří Krůček

Table of Contents

Index of Figures

Index of Tables

Index of Code Snippets

Abstract

Title: Support for Development of Enterprise Applications in Java
Author: Jiří Krůček
Department: Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Supervisor's e-mail address: hnetynka@dsrg.mff.cuni.cz
Abstract: The master thesis provides a comparative analysis of two broadly-used technologies for building Java-based enterprise applications, the Spring Framework 2.5 and Enterprise JavaBeans (EJB) 3.0. Its main goal is to serve as a valuable source of information about their key features, thus helping developers with the decision which technology to use according to given requirements on the application to be developed. First, an overview of Spring and EJB's basic ideas and core design concepts is given. Further, a detailed examination of their capabilities is carried out in three main areas of comparison: the thesis focuses on how they (1) address management of application business objects, (2) analyses a basic set of provided middleware services, and also (3) concerns with more practical issues related to various application development efforts, such as architecture, testing, and configuration of applications being developed by using these technologies. Each area of comparison is divided into several tightly-focused sections thoroughly discussing Spring and EJB's capabilities relevant to a particular subject, their strengths and weaknesses. At the end, their key features are summarized, evaluated, and some recommendations are given.
Keywords: Spring, EJB, frameworks, J2EE, JEE

Název: Podpora pro vývoj aplikací na platformě Java
Autor: Jiří Krůček
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Petr Hnětynka, Ph.D.
E-mail vedoucího: hnetynka@dsrg.mff.cuni.cz
Abstrakt: Diplomová práce se zabývá srovnáním dvou často používaných technologií při vývoji aplikací na platformě Java, frameworku Spring 2.5 a komponentové architektury Enterprise JavaBeans (EJB) 3.0. Jejím cílem je poskytnout podrobné informace o všech důležitých vlastnostech obou technologií a tím napomoci vývojářům při rozhodování, kterou technologii je vhodnější zvolit pro vývoj konkrétní aplikace dle zadaných požadavků. V úvodu práce jsou rozebrány základní myšlenky a koncepty, na nichž jsou Spring a EJB založeny. Dále následuje detailní analýza obou technologií, rozdělená do tří hlavních oblastí: (1) analýza běhového prostředí a způsobu jakým jsou spravovány objekty tvořící aplikaci, (2) analýza poskytovaných infrastrukturních služeb a (3) analýza různých souvislostí spojených s použitím dané technologie při vývoji jako jsou vliv na architekturu vyvíjené aplikace, způsob jejího testování a její konfiguraci. Každá z těchto tří hlavních oblastí je dále rozdělena na několik úzce orientovaných částí, které se zabývají konkrétními tématy z dané oblasti. V závěru práce je provedeno celkové zhodnocení obou technologií a na jejím základě jsou poskytnuta určitá doporučení pro jejich použití.
Klíčová slova: Spring, EJB, frameworky, J2EE, JEE

1 Introduction

The most propelling power of innovation and huge progress in the information technology industry is organizations, or more precisely their business needs. It is long time ago when to own an information system meant a competitive advantage for them; today, it is almost a necessity to have one to survive on the market. They run large software applications to control and support their everyday business. Enterprise resource planning, production management, supply chain management, and customer relationship management systems are characteristic representatives of such applications, generally called *enterprise applications* or *enterprise systems*.

1.1 Goals of an Enterprise Architecture

Although enterprise applications usually differ in their purpose, some architectural goals should be achieved in general, regardless of the type of application we are developing.

Firstly, every application should conform to customer requirements, be built on time, and within budget. These basic rules are mentioned in any book about software engineering [4].

Further, well-designed enterprise applications should meet these fundamental goals [1]:

- **Robustness:** They have to be reliable and bug-free. Their users must trust them.
- **Performance:** They have to meet performance and throughput expectations. A cumbersome and slow application will not be well adopted by its users.
- **Scalability:** They have to support increased load, given appropriate hardware. They should run efficiently in a cluster of servers.
- **Maintainability:** They have to be maintainable and easy (re)configurable. They are likely to be a key part of an organization's software mix for years.
- **Reusability:** They have to fit into an organization's long term strategy. Thus, it is important to foster reuse in order that code duplication is minimized and investment leveraged to the full.
- **Extensibility:** They have to be able to accommodate new business requirements. Their primary function is to support business of an organization and react flexible to its emergent needs.

Depending on business requirements, the following goals may need to be achieved as well:

- **Support for multiple client types:** An enterprise application usually supports a wide range of clients, such as web clients, standalone GUI applications, mobile and PDA clients, to make easy user access to an organization's data from everywhere.

- **Portability:** Sometimes it is necessary to run enterprise applications in different environments, for example on several operating systems, or achieve the ability to change important resources like databases, which applications rely on.

1.2 Multi-Tier Architecture

Most of the characteristics of enterprise applications mentioned in the previous section are determined by the proper architecture and structure of the application. Experience has shown the value of dividing applications into multiple *tiers* [1], [7]. This ensures a clean division of responsibility within applications and promotes their modularity. A typical *three-tier architecture* shown in figure 1.1 consist of:

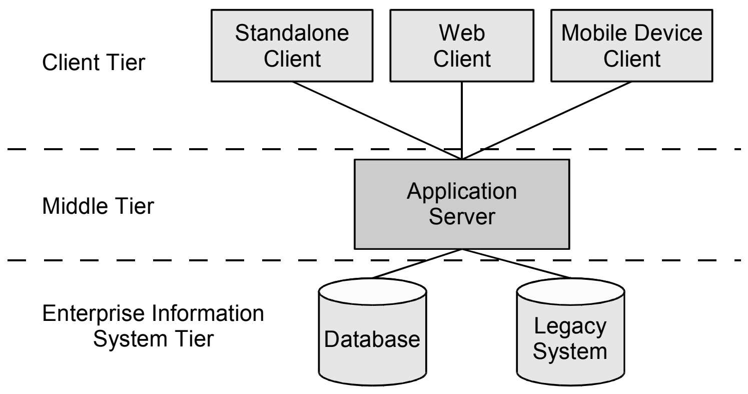

Figure 1.1: Three-tier architecture

- **Enterprise Information System (EIS) tier:** It comprises the enterprise resources that the application has to access to do its work. These include mostly databases and other legacy systems. Sometimes it is called the *integration* or *data tier* since applications often use resources to store and retrieve data.

- **Middle tier:** This tier contains business logic of the application. It is responsible for coordinating business processes, managing transactions, mediating access to EIS tier resources, exposing business services by means of remoting, ensuring application security, and performing a number of other functions. The middle tier is usually represented by an application server.

- **Client tier:** It consists of various application clients, which present data to users. Therefore, we may also meet with the term the *user interface tier*. Its basic responsibilities are: handling user requests, input data validation, calling business services, and displaying results to users.

1.3 Strong Application Infrastructure and JEE Platform

When developing a multi-tier application, developers face a lot of issues that usually have no connection to business requirements, but rather relate to the application's low-level infrastructure; for example, how to manage dependencies between application objects, how to demarcate transaction boundaries, how to enforce security policies, and similar. Hence, it is essential for any application to have a strong infrastructure that enables developers to spend more effort focusing on business logic, without the distraction of handling common concerns like persistence, transactional integrity, and security.

Today many enterprise applications are being developed in the *Java Enterprise Edition* (JEE) platform [18]. The JEE platform is a set of API standards specifying many technologies for building multi-tier applications, for example servlets, JSP, JSF, EJB, JTA, JNDI, JMS, JAX-WS, etc. The platform is backed by Sun Microsystems, but the process of API standardization is public throughout the *Java Community Process* (JCP), thus plenty of organizations, communities, as well as individuals can participate in. The benefit of this approach is a rich variety of both commercial and open source API implementations to choose from.

One of the several APIs in JEE is the *Enterprise JavaBeans* (EJB) [13] specification. EJB is a server-side component architecture for development and deployment of distributed enterprise applications. It allows developers to construct modular applications from well-defined, reusable components that encapsulate their business logic. Once assembled, applications can be deployed in an JEE-compliant application server, which serves as a runtime environment for residing components and provides them with a basic set of standardized services.

Unfortunately, the first versions of EJB, up to version 2.1 [5] inclusive, have not been adopted by developers very well. Although the specification promised more simple and faster development of high quality applications, the reality has differed. For newcomers, the APIs of the standard has been far more complex than what typical developers are used to. At all, some of the concerns addressed by EJB, such as *Object-Relational Mapping* (ORM), has been considered so poorly designed and over-complicated that many special JEE design patterns have originated to achieve satisfactory results.

The lack for simple and easy to use solution led to the emergence of so-called *lightweight frameworks*. Based on the clear and proven principles of *Plain Old Java Objects* (POJOs), *Dependency Injection* (DI), and *Aspect-Oriented Programming* (AOP), they have been quickly and widely accepted. Particularly *Spring* [15], an application framework for building applications' infrastructural backbones, has become very popular.

Successful lightweight solutions caused naturally an increased activity around the EJB specification. As a result, EJB 3.0 [13] has been introduced. Augmented by Java *annotations* and largely influenced by Spring and other lightweight frameworks, EJB 3.0 aims to rectify its reputation.

1.4 Goal of the Thesis

The goal of this thesis is to provide a comparative analysis of both above mentioned solutions for building the middle tier of enterprise applications, i.e., Spring 2.5 and EJB 3.0. We look under the hood to examine their design, business object management strategies, and provided middleware services. We discuss their strengths as well as their weaknesses, and their implications for application development.

Where appropriate the study concerns with EJB 2.1 [5] in order to contrast features of EJB 3.0 with those of the previous version. In addition, some core concepts of the EJB technology can be better explained in the context of EJB 2.1. Taking into account upcoming Spring 3.0 [19] and EJB 3.1 [20], new significant improvements and enhancements of these technologies are considered as well.

Before delving into a more in-depth comparison, it is also necessary to mention two important facts that we should keep in mind all the time:

- First, EJB claims to be a *component architecture* whereas Spring is an *application framework*. Nevertheless, they both intend to operate in the same enterprise Java development space, thus, their comparison is meaningful. Moreover, EJB is often denoted as a framework because it really has some framework characteristics; we are doing so as well, partly because of the need for an unified term for both analysed technologies.
- Second, EJB is a *specification* while Spring is an *implementation*, which are not the same things. Hence, we concentrate mainly on their design comparison.

1.5 Structure of the Thesis

To achieve the goals, the thesis is structured as follows:

Chapter “Concept of the Comparison” discusses the motivation for writing the comparison and describes the concept of the comparison in greater detail.

Chapter “Overview of the Examined Frameworks” introduces the JEE platform and both examined frameworks. It gives an overview of basic ideas and design concepts of each framework.

The next three chapters are devoted to each of three main comparison areas. Chapter “Business Object Management” focuses on how the frameworks manage application business objects. Chapter “Services” examines a basic set of middleware services provided by the frameworks. Chapter “Application Development” concerns with more practical issues of using the frameworks, relating to various application development efforts: architecture, testing, and configuration.

Chapter “Conclusion” summarises the results of the comparison. The fulfilment of the goals is analysed and possible future work is discussed.

2 Concept of the Comparison

Both Spring and EJB have their advantages as well as disadvantages. However, because of their complexity it is not easy for a newcomer to evaluate which solution is the most suitable for given requirements. Obviously, this makes a room for a side-by-side comparative analysis which has to contain the most relevant information about both solutions, and thus help to take the right decision.

2.1 Motivation for Writing the Comparative Analysis

The Internet provides a lot of material about Spring and EJB. Nevertheless, this material usually takes the form of short articles [3], [22], [23], or presentations [24], [27], which do not contain much information, often discuss only selected topics, and do not delve into details. Other studies [26] give a description of basic concepts and facilities of both frameworks separately, rather than comparing them to each other. As there is not enough objective information comparing the two, one has no choice but to study the reference documentation of the frameworks to take a more comprehensive view.

The purpose of this work is to fill up the blank and provide an in-depth study describing and comparing the frameworks from a technical as well as pragmatical perspective. The thesis aims to serve as a valuable source of the most important information about the frameworks for:

- a newcomer who wants to get familiar with both frameworks
- a developer who needs to help with the decision which framework is more appropriate for development of an application by given requirements
- a developer who is interested in the core concepts, main features, and differences of both frameworks

On the other hand, the goal of this thesis is not to provide a detailed description of all the analysed features of the frameworks; to get more information about any discussed feature please consult the reference documentation of the framework.

2.2 Areas of Comparison

First of all, we should make clear the concept of the comparison. With respect to the essential responsibilities of the frameworks, the comparative analysis is divided into these main areas:

Business Object Management

The main purpose of any application framework is to provide a runtime for application business objects. Hence, our first concern is the comparison of the frameworks' core infrastructure for business object management. We are interested in:

- **Managed object contract:** What contract application objects have to abide by to be managed by the frameworks and what restrictions result from that.
- **State management:** How the frameworks allow for maintaining a conversational state with clients.
- **Lifecycle management**: How the frameworks control the lifecycle of application objects running within them. In the simplest case, how managed objects are created and destroyed.
- **Dependency management:** What type of mechanism the frameworks use to manage relationships and dependencies between application objects, and how the references to managed objects can be obtained.
- **Concurrency management:** How the issue of concurrent access to managed objects by multiple clients at the same time is approached.

Services

The second point of interest is the examination of a basic set of middleware services provided by the frameworks. Particularly, we are interested in how each service is provided, how the frameworks facilitate its usage, what its main strengths and weaknesses are. We analyse:

- **AOP support:** How the frameworks embrace the AOP concept to allow developers to implement custom middleware services.
- **Data access:** What means the frameworks provide to facilitate data access.
- **Transaction management:** How the frameworks facilitate transaction processing.
- **Remoting:** How business services can be exposed to remote clients and through which protocols.
- **Messaging:** How business services can communicate with clients asynchronously via sending and receiving messages.
- **Security:** What type of mechanism the frameworks provide to authenticate users and control access to protected resources.
- **Scheduling:** How the frameworks allow for scheduling timed notifications important to application workflow management.

Application Development

Finally, we concentrate on more practical issues of using frameworks relating to various areas of application development efforts. We discuss:

- **Architecture:** What the frameworks' typical usage scenarios are, and what scalability strategies are used in case applications need to run in a cluster of servers.
- **Testing:** How the frameworks facilitate application testing, particularly unit and integration testing, fundamental to test automation.
- **Configuration:** What possibilities the frameworks provide to setup application configuration and declarative services.

Each group of comparison aspects is covered by an individual chapter divided into tightly focused sections. At the beginning of each section the point of interest is described, then the approaches of the frameworks are analysed, and finally a brief summary is provided.

3 Overview of the Examined Frameworks

In this chapter we briefly introduce the JEE platform as well as the examined frameworks. We familiarize ourselves with their basic ideas, aims, and fundamental architecture concepts. We hereby make a starting position for their further comparison.

3.1 Java Enterprise Edition

JEE is a set of technologies that allow for developing multi-tier enterprise applications in the Java programming language advantages of the *Java Standard Edition* (JSE) platform (like independence on OS). It defines a basic set of *application components* which may be used to build JEE applications, and specifies the runtime in which applications may be deployed. This results in a clear demarcation between the application and the runtime environment, allowing the runtime environment to abstract most of middleware services.

The JEE platform defines the following application components [18]:

- **Client-tier components:** *Application clients* and *applets* providing a graphical user interface (GUI) and presenting data to users.
- **Web-tier components:** *Java Servlets*, *JavaServer Faces* (JSF) and *JavaServer Pages* (JSP) for handling web requests and generating static and dynamic web pages.
- **Business-tier components:** *Enterprise JavaBeans* (EJB) encapsulating business logic that meets the business needs of an application.

For each type of an application component, the platform specifies its host environment, a so-called *container*. A container is responsible for managing the lifecycle of residing components and supports a predefined set of middleware services, which are accessible via standardized APIs. Figure 3.1 shows all four containers distinguished by the specification: an *applet container*, an *application client container*, a *web container*, and an *EJB container*.

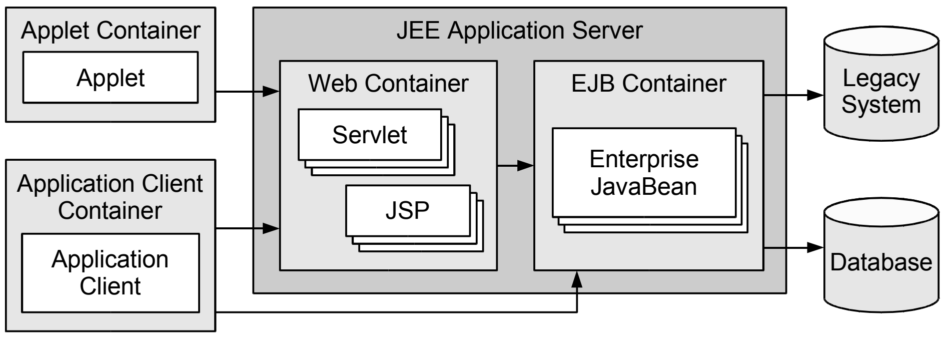

Figure 3.1: JEE containers

In practice we may more often encounter the terms *web server* for a product which is compliant with the web container specification, and *application server* which embodies a web container as well as an EJB container. When implementing a server, vendors usually make use of their existing infrastructure, encapsulating it into the prescribed APIs. The competition between server vendors results in the existence of various servers with different performance, scalability, failover, and other capabilities.

The most important JEE APIs (services) are as follows [18]:

- **The Java Naming and Directory Interface (JNDI)** provides a unified interface to multiple naming and directory services.
- **The Java Database Connectivity API (JDBC)** provides means for accessing relational databases.
- **The Java Transaction API (JTA)** provides an infrastructure for transaction management, especially distributed transactions spanning multiple resources.
- **The Remote Method Invocation (RMI)** is a technology enabling Java objects to communicate remotely with each other.
- **The Java Message Service API (JMS)** provides a common way to create, send, receive, and read messages through the use of message-oriented middleware.
- **The Java API for XML Web Services (JAX-WS)** provides support for web services.
- **The Java Authentication and Authorization Service (JAAS)** is a set of APIs that enables services to authenticate and enforce access controls upon users.
- **The JEE Connector Architecture (JCA)** allows for application servers and legacy systems integration.

From the deployment perspective, the basic JEE application building block is a *JEE module.* It consists of one or more application components destined for the same container type; for example, web-tier components are assembled into web modules, EJB components into EJB modules. Further, every JEE module contains a so-called *deployment descriptor* (DD) which describes deployment settings of the module. This description is used by the container to identify module content, and to set the runtime properties of each component. Because the deployment descriptor information is declarative, it can be changed without any source code modification and recompilation. This allows for better reusability of components so that they may be included into various JEE modules with different deployment parameters.

A *JEE application* is assembled from one or more JEE modules and contains also an *application deployment descriptor* which describes deployment settings of the entire application. At deployment time, the application server verifies that the JEE application and all the contained modules are well formed and in compliance with the specification, and installs the modules in the target containers. A JEE module without an application deployment descriptor can be also deployed as a standalone module.

3.2 Enterprise JavaBeans 2.1

The Enterprise JavaBeans architecture is a component architecture for development and deployment of distributed enterprise applications. The EJB specification was originally developed in 1997 by IBM and later adopted by Sun Microsystems. It defines a server-side component model for building applications from specialized components encapsulating their business logic and domain model. Further, it also specifies components' execution environment, an *EJB container*, which takes responsibility for system-level issues and provides a prescribed set of middleware services. The division of responsibilities between the component programmer and the EJB container is the central idea of EJB.

In EJB, components are called *enterprise beans*, shortly *beans.* The EJB specification defines three types of components from which applications can be assembled [13]:

- **Session beans** are intended for modelling business services and processes; they encapsulate the business logic of an application. They often perform actions such as calling a legacy system, accessing a database, and similar. Depending on whether the session bean can maintain a conversational state with the client, the EJB specification distinguishes *stateless* and *stateful session beans*.

- **Entity beans** model business data. They provide an object-oriented view of domain model entities stored in the database. Typically, session beans harness entity beans to achieve business goals, such as a stock-trading engine (a session bean) that deals with stocks (entity beans).

- **Message-driven beans** are similar to session beans in that they encapsulate business logic. The difference is that they behave asynchronously and clients communicate with them using messages (via JMS).

To be deployed in an EJB container, beans have to be assembled into an *EJB module.* EJB modules are packaged as JAR files with a .jar extension. In figure 3.2 we can see an EJB container and the APIs that the container provides to bean instances at runtime. We have briefly described some of these APIs in the previous section. A more comprehensive information about all the APIs can be found at Sun's homepage [45] and in [17], [18].

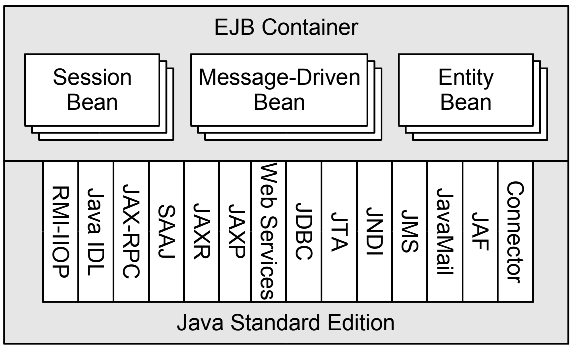

Figure 3.2: EJB container

The EJB architecture aims to be an “one-size-fits-all” solution, i.e., it provides its own solution to every problem that developers usually face to when implementing the middle tier of enterprise applications. Thus, it incorporates concepts from a lot of areas, including component-driven software, distributed computing, remoting, persistence, security, and more. Now, let us see the most important concepts in detail.

3.2.1 Distributed Architecture

EJB components have been originally designed as *distributed objects* to be accessed by remote clients, located elsewhere on the network. The communication between remote clients and beans is realized through the RMI technology [30]. To allow remote method invocation, EJB prescribes that beans have to consist of two parts: a *remote interface* and a *bean class*. The remote interface serves for specifying those business methods of the bean that are accessible by its remote clients. The bean class implements the methods declared in the remote interface. A typical implementation model of RMI in the context of EJB is shown in figure 3.3; a brief description follows in the next paragraph.

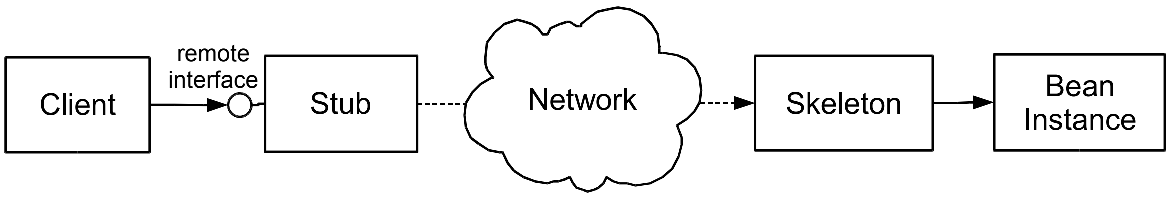

Figure 3.3: Calling a bean by a remote client

When invoking a method on a remote bean, a client does not work with the bean instance directly, but it calls a *stub*, which is a client-side proxy object. This is legitimate because the stub implements the same remote interface as the bean class. The stub calls over the network the corresponding server-side proxy object, a skeleton, which delegates the call to the appropriate bean instance. It does its work, and then returns control to the skeleton, which returns to the stub, which then returns control to the client. As we may see, the stub and the skeleton mask network communication between the remote client and the bean instance. They are also responsible for converting transferred data, such as parameters of the called method, into and from their network representation; this process is known as data *marshalling* and *unmarshalling*.

Further, remote clients are able to disconnect from beans and reconnect later using that bean. For this purpose, the EJB specification introduces the concept of *EJB object handles*. Handles are persistent references to beans' server-side proxy objects, called *EJB objects*. If for some reason a client disconnects from the EJB container, it can use the handle to reconnect to the EJB object so that it does not lose its conversational state with that bean. The following code snippet illustrates using EJB object handles.

```
// get the EJB object handle
javax.ejb.Handle myHandle = myEJBObject.getHandle();
// serialize and save the handle in permanent storage
ObjectOutputStream outputStream = ...;
outputStream.writeObject(myHandle);

...

// deserialize the handle to use the EJB object again
ObjectInputStream inputStream = ...;
myHandle = (Handle) inputStream.readObject();
// convert the handle to the EJB object
MyRemoteInterface myEJBObject = (MyRemoteInterface) javax.rmi.
  PortableRemoteObject.narrow(myHandle.getEJBObject(),
    MyRemoteInterface.class);
```

Code Snippet 3.1: Concept of EJB object handles

Unfortunately, a lot of applications [1], [2] do not require this distribution computing functionality, which brings a lot of performance penalties resulting from the network communication. In many cases, there is no need for building the business service layer from distributed objects. It is more usual that most of the clients, such as web-tier components, access business services locally, running in the same Java Virtual Machine (JVM) as the application. The EJB 2.0 specification addressed this concern by introducing another type of *component interface*, a *local interface*. The beans that support this interface can be called by local clients without any network communication overhead.

To conclude the concept of the component interface, beans can define both the remote interface to specify the methods available to remote clients and the local interface to specify the methods available to local clients.

3.2.2 Implicit Middleware

The EJB container provides several types of system-level services to residing beans; they range from core services like thread management and instance pooling to more high-level ones like transaction management and security. The mechanism by which the EJB container provides its services is based on three basic ideas [8]:

- **Contracts:** The specification clearly defines responsibilities between the various parts of an application using EJB components: the client, the EJB component, and the EJB container. The definition of these responsibilities is formally known as a *contract*. An application developer follows the rules of the bean development contract so that the container is able to provide its services.
- **Declarative configuration:** The services provided by the container are defined in such a way that they are *orthogonal* to a bean. In other words, the services are separated from the Java files that implement the business logic of the bean. For every bean, the required services can be *configured declaratively* in a deployment descriptor of the EJB module.
- **Container interception:** A client never uses a bean instance directly, but it works with its corresponding proxy object, termed as an *EJB object.* The EJB object interposes on each and every business method call to perform

configured middleware services before and after it delegates the call to the bean instance. Depending on whether the bean supports local or remote clients, the container generates either the network-unaware or network-aware EJB object. A network-aware EJB object consists of a client-side (stub) and a server-side (skeleton) proxy object, thus, the EJB object is rather an abstraction than a regular object.

The concept of *declarative middleware*, also referred to as *implicit middleware* [9], is considered as the best thing in EJB. There is no need for writing any code to middleware APIs, rather, we declare what we need in a descriptor file. The separation of business logic and middleware logic makes applications more maintainable. Finally, the underlying middleware can be changed by tweaking the respective descriptor file, without modifying application code.

3.2.3 Locating Components

Another important issue is how a client acquires a reference to a bean instance, or more precisely to an EJB object. For this purpose the client utilizes the JNDI service [31] which serves for publishing various objects and object references, binding them to user-friendly names and allowing for their further lookup under these names.

Nevertheless, EJB objects are not published directly. Instead, the EJB specification defines a *home interface* which provides methods for creating, destroying, and finding EJB objects; the home interface acts as a lifecycle interface for EJB objects. According to how beans can be called, there are specified a *remote* and a *local home interface*. Every bean is supposed to have a corresponding home interface, local, remote, or both. At deployment time, for each home interface, the container generates an implementation, a so-called *home object*, and publishes it through the JNDI service. Clients can obtain references to EJB objects via these home objects.

3.2.4 Persistence Concepts

As mentioned above, the EJB architecture provides an O/R mapping solution based on special data access components, entity beans. The specification describes two types of entity beans: entity beans with *Container-Managed Persistence* (CMP) and entity beans with *Bean-Managed Persistence* (BMP). The EJB container handles persistence for entity beans with CMP, requiring the developer only to define the bean properties to be persisted. In the case of entity beans with BMP, the developer is completely responsible for handling persistence by implementing callback methods to load state and store state to the database. Finally, the specification introduces a portable query language, *EJB QL*, for use by entity beans with CMP, relieving developers of writing database-specific SQL queries.

3.3 Spring

JEE is a very powerful platform, however, its overall complexity makes it harder to understand and use. Particularly, this holds for the EJB 2.1 specification, which has not been considered to be well-designed and its adoption rather complicates the development process. Spring is a layered application framework which is aimed at simplifying the development of applications in the Java platform, ranged from desktop to server-side ones.

The first parts of the Spring Framework were written by Rod Johnson and published in his book Expert One on One J2EE Design and Development [1] in 2003. In this book Rod Johnson expressed his views how applications that work with various parts of the JEE platform could become simpler and more consistent than developers and companies were used to at that time. Shortly afterwards the code shipped with the book had been extended and in March 2004 the first version of the Spring Framework was released. Based on accepted best practices, Spring has been widely adopted in the Java community.

Spring offers many alternative approaches to disparate EJB efforts and provides a rich set of service abstractions over the JEE technologies. Furthermore, it does not endeavour to address every goal by its own solution, but allows for integration with various proven frameworks and technologies. Though its huge variability, which gives developers much freedom, it provides a consistent way of working with most things to encourage productivity and focus on design of applications being developed, rather than on the employed technology.

Open to further enhancements, Spring is a very flexible framework. To achieve loose coupling, Spring follows the first principle of reusable OO design [6]: "Program to an interface, not an implementation". The backbone of the framework is formed by proven design patterns, such as Factory, Template Method, Strategy, Observer, improving its modularity and extensibility. Written in the Java programming language, Spring also takes advantage of many valuable Java features like the Reflection API, dynamic proxies, JavaBeans, inner classes, etc. Moreover, Spring has popularized many OO design techniques and concepts that are heavily utilized by the framework itself.

3.3.1 Return of Plain Old Java Objects

Central in the Spring Framework is an *Inversion of Control* (IoC) *container* which is, similar to the EJB container, responsible for business object management. Nevertheless, Spring does not introduce any special types of components for building applications, but prefers the style of ordinary Java objects, hence an acronym *Plain Old Java Objects* (POJOs) [10]. A typical POJO is a JavaBean object [32] that has a no-argument constructor and allows access to properties using getter and setter methods. The business contract of a POJO can be defined using a regular Java interface. The concept of POJOs advocates the idea that the simpler the design, the better.

In the terminology of Spring managed objects are called *beans*. In order not to confuse a Spring bean and an EJB bean, we will be using this term for the objects managed by Spring as less as possible.

3.3.2 Dependency Injection Configuration

Another responsibility of the IoC container is to configure application business objects running within it, without the need for lookups in application code as in the case of the JNDI mechanism. This approach is known as a *Dependency Injection* pattern, a particular form of Inversion of Control [29]. Any dependencies of an object, such as business service objects and data access objects, are declared via metadata in the source code or in an XML descriptor file. According to the configuration the IoC container “injects” the dependencies into the object during its creation or initialization phase, using the appropriate constructor and setter methods, respectively. Following OO design recommendations, managed objects usually collaborate with each other through well-defined interfaces. This allows the IoC container to set the collaborators at runtime, keeping the target object unaware of the concrete service implementation.

3.3.3 Declarative Middleware Using AOP Concepts

To provide middleware services, Spring has its own AOP [28] framework. Transaction demarcation, security checks, logging, and other cross-cutting concerns are modularized in *aspects*. An aspect can alter the behaviour of the base code of any managed object (the non-aspect part of a program) by applying *advice* (additional behaviour) at various *join points* (points in a program) specified in a quantification or query called a *pointcut* (that detects whether a given join point matches). Spring provides a close integration between its AOP framework and IoC container to make easy AOP configuration and allow for advising managed objects in a transparent, non-invasive, declarative way.

Similar to EJB, an advice can be applied at method level; a method represents a single join point. Consequently, a pointcut specifies, usually in a regular expression syntax manner, a set of methods to which the advice should be applied.

According to when the advice may proceed, Spring supports several advice types: *before advice* (executes before a joint point is invoked), *after returning advice* (executes after a join point completes normally), *after throwing advice* (executes after throwing a particular exception), *after (finally) advice* (executes regardless of the means by which a join point exits - normal or exceptional return), and *around advice* (it surrounds a join point such as a method invocation). The around advice is the most powerful type of advice because it can implement all other advice types. Moreover, it is responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Linking aspects with managed objects to create an advised object is called *weaving*. Spring performs a runtime weaving based on *AOP proxies* and *interceptors*. Figure 3.4 illustrates this mechanism.

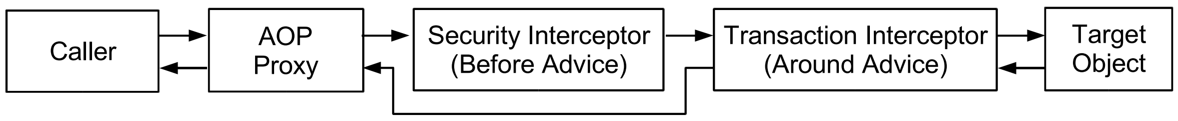
Caller
AOP Proxy
Security Interceptor (Before Advice)
Transaction Interceptor (Around Advice)
Target Object

Figure 3.4: Spring AOP concept

At runtime, for each object being advised, also referred to as a *target object*, an AOP proxy is created by the Spring AOP framework to implement the aspect contracts. Thus, a caller does not work with the target object directly, but with the AOP proxy, which has the same business method signatures as the target object. When an advised method is invoked, the AOP proxy delegates the call to the first interceptor so that it can provide its service (apply the advice). After the first interceptor completes, it delegates the call to the next interceptor in the chain. Finally, the last interceptor delegates the call to the target object, which does its work. Figure 3.4 shows how two advices, security and transactional, are applied to the target object. Let us notice, the security interceptor is skipped when the call is returning; it implements before advice.

Spring offers many important middleware services, such as transaction management, already modularized, but it also allows for implementing custom services.

3.3.4 Modular Architecture

The architecture of Spring is organized in several well-defined modules, as shown in figure 3.5.

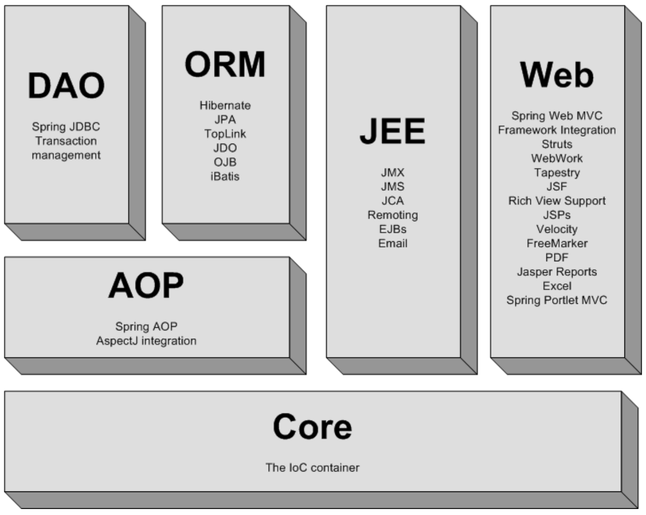

Figure 3.5: Overview of the Spring Framework

The *Core* module is the most fundamental part of the framework that comprises the IoC container implementation. Other modules, namely *AOP* (Spring AOP capabilities implementation), *DAO* (support for effective work with JDBC), *ORM* (integration with various ORM frameworks), *JEE* (integration with JEE technologies), and *Web* module (web-tier presentation facilities and integration with disparate web frameworks and technologies), encapsulate additional Spring's features and are optional.

In addition, there exist many extra sub-projects, such as Spring Web Flow, Spring LDAP, Spring Rich Client, Spring Security etc., to enlarge Spring capabilities. However, their detailed examination is beyond the scope of this work. Spring's homepage [43] would be a worthwhile starting point to get familiarized with them.

3.4 Enterprise JavaBeans 3.0

The success of alternative, more simple solutions for building enterprise applications caused naturally an increased activity around the EJB specification. As a result, a new EJB 3.0 specification was released in 2006. As the main objective, the EJB architecture has been improved by reducing its complexity from the developer's point of view. The EJB programming model has been enhanced with many new features, which are reminiscent of that offered by Spring.

3.4.1 Use of Java Annotations

EJB 3.0 relies heavily on the use of *Java annotations* [12], a feature added to the Java programming language with its 5.0 release. Annotations are a way of providing additional contextual information to various program elements, such as classes, methods, fields, etc., within the source code to help further explain or characterize them. The EJB 3.0 specification defines many built-in annotations which are used to configure beans and specify their behaviour at runtime, moving most configuration information from the EJB deployment descriptor.

The idea of Java annotations is not new. Another way of adding metadata to the source code is through some special code comments. Nevertheless, there is a difference. A comment-style annotation is removed during the source code compilation, however, a Java annotation remains available in the compiled .class file. Thus, Java annotations can be also processed at deployment time or at runtime. For example, a transaction demarcation annotation may be used to inform the EJB container that some method has to be executed within a transaction:

```
...
@TransactionAttribute(REQUIRES)
public void doSomething() {...}
...
```

Code Snippet 3.2: Java annotation

As we may see, a Java annotation begins with an @ sign followed by the annotation name and annotation parameters, if any. The author can also specify a default value for each annotation parameter, which is applied if no value is supplied by the developer. For example, the value `REQUIRED` of the parameter of the transaction annotation shown in code snippet 3.2 can be omitted because it is its default value. In EJB 3.0, specification of programmatic defaults, including annotations, to reduce the need for the developer to specify common, expected behaviour and requirements on the EJB container is taken whenever possible. In other words, the developer only needs to specify unconventional aspects of the application. Hence, this approach is known as “configuration by exception” or “convention over configuration”.

3.4.2 New Features

As mentioned above, the architects of EJB 3.0 have been highly inspired by the architecture of Spring. Many valuable concepts introduced or popularized by Spring have been incorporated into the EJB 3.0 architecture. The most notable architecture and design changes are:

- **POJO-style components:** The EJB component model has been simplified to conform to the concept of POJOs. There is no more any need for defining home interfaces. An EJB 3.0 component is a plain Java object with the contract specified through a regular Java interface.

- **Dependency injection:** The EJB 3.0 specification has introduced its own dependency injection mechanism. To take advantage of this functionality, EJB 3.0 components has only to mark components and resources which they depend on via annotations and let the EJB container do the rest of work.

- **Interceptors:** To allow for custom services development, EJB functionality has been enhanced by an interceptor facility for session and message-driven beans. Interceptors are used to interpose on business method invocations and lifecycle events that occur on an bean instance.

- **Java Persistence API (JPA):** Too complicated and heavyweight entity bean model has been considerably revised and simplified. A new standard API for the management of persistence and O/R mapping, the Java Persistence API [14], has been specified to support lightweight, POJO-based domain modelling.

Though a lot of architecture and design changes, EJB 3.0 is backward compatible with the previous versions. That is, existing EJB 2.1 and earlier applications have to be supported to run unchanged in EJB 3.0 containers.

4 Business Object Management

The central part of EJB as well as of Spring is a container which serves as a runtime environment for application business objects and concerns with a lot of fundamental issues relating to their management. Each container has to manage the lifecycle of residing objects, provide a way of obtaining managed objects, resolve dependencies between managed objects, and more (we use the term *managed object* for an object running inside a container although they are often termed “components”, as in the case of EJB). The goal of this chapter is to compare such a core functionality of both containers and discuss related issues.

This core functionality is particularly aimed at management of the business objects that represent services, such as business service objects and data access objects, rather than persistent domain objects, which are usually managed by a dedicated persistence framework. Therefore, when we use the term *business object*, we basically mean a service object (in the case of EJB, a session bean) that somehow works with persistent domain objects. The persistence management of domain objects, for example by ORM frameworks such as Hibernate [16] or JPA [14] is beyond the scope of this thesis.

4.1 Managed Object Contract

To be managed by a container, application objects generally have to abide by a contract. The contract defines requirements for seamless cooperation between the container and managed objects, and also can specify requirements on interaction between managed objects and their clients. As a result, application objects are usually supposed to have a prescribed structure; for example, they have to implement a generic interface to be informed about important lifecycle state changes, extend a base superclass to allow the container to provide its services, or expose their business methods through a well-defined interface. In addition, the contract often specifies some programming restrictions which developers have to follow to ensure that the container can do its work.

The amount of classes and interfaces that application objects are forced to incorporate into their structure determines how the framework is invasive [1]. That is, how easy it is to reuse managed objects in another application based on a different application framework.

4.1.1 Contract

As described in section 3.2, EJB 2.1 beans are rather complicated. Every session bean is assembled at least from three Java files: the home interface, the component interface, and the bean class. Figure 4.1 illustrates the structure of a remote session bean consisting of `MyBeanHomeInterface` (the remote home interface), `MyBeanInterface` (the remote interface), and `MyBean` (the bean class). If the `MyBean` bean were a local session bean, `MyBeanHomeInterface` would have to extend the `javax.ejb.EJBLocalHome` interface and `MyBeanInterface` would have to extend the `javax.ejb.EJBLocalObject` interface.

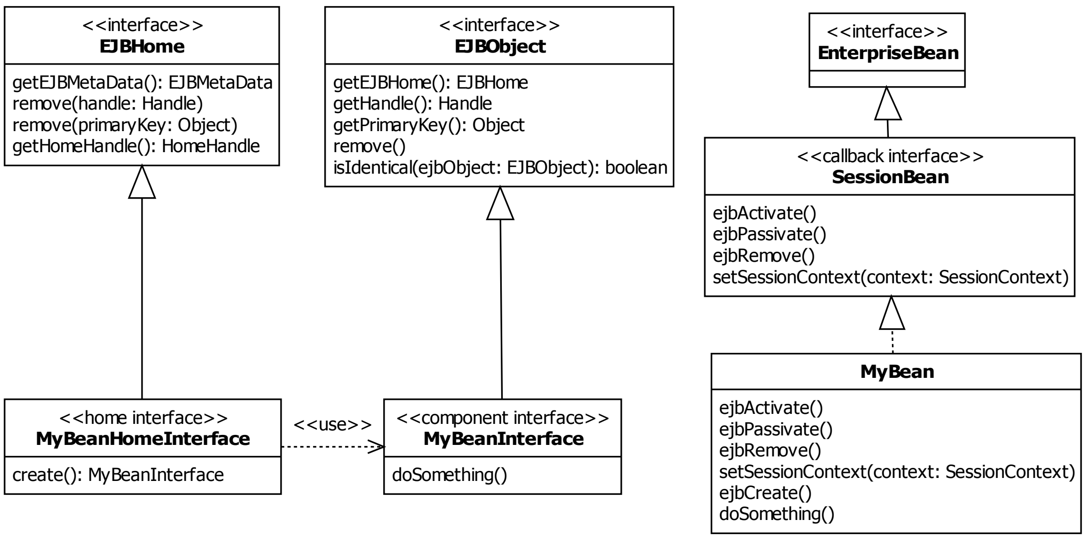

Figure 4.1: EJB 2.1 remote session bean contract

Although it may seem that the MyBean class implements only one interface (`SessionBean`), in fact, every session bean class has to provide an implementation of methods defined by several interfaces:

- It has to implement all business methods declared on the component interface (remote or local). Nevertheless, it should not implement the component interface by means of the Java programming language, i.e., by using the `implements` Java key word (there is no UML realization relationship between the `MyBeanInterface` interface and the `MyBean` class in figure 4.1). It is not a good idea because the bean class would have to provide empty implementations of methods from either the `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject` interface. Fortunately, there is a straightforward solution to avoid manual error-prone synchronization, the *Business Methods Interface* pattern [1] (see figure 4.2): both the component interface and the bean class have to implement a common interface that defines the business contract of the bean.

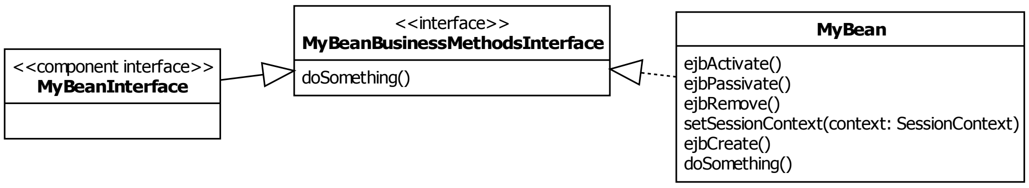

Figure 4.2: The Business Methods Interface pattern

- It has to implement some "constructor" methods declared on the home interface. That is, for each method with the "create" name-prefix declared on the home interface, the bean class has to implement the corresponding method with the "ejbCreate" name-prefix and the same parameters. Clients call these "constructor" methods on the home interface to obtain EJB objects; the corresponding methods implemented by the bean class are called internally by the container after bean instantiation via the default constructor.

- It has to implement the `javax.ejb.SessionBean` callback interface (for other types of beans, the EJB 2.1 specification defines similar callback interfaces). The methods declared on this interface serves to manage the bean and its life cycle by the EJB container, and alert the bean to significant events. A downside of this contract is that the bean class has to provide an implementation of each callback method although it may be only interested in particular events, if any.

To manage and keep consistent such amount of classes and interfaces for every bean does not often dispense with the support of tools like IDEs or XDoclet [54]. Using such tools, developers usually have to implement only the bean class and let the tools generate other necessary Java files on the basis of special comment-style metadata provided by the bean class.

In contrast to EJB 2.1, the Spring Framework does not pose any special requirements on managed objects and their structure; they are POJOs. The idea behind this approach is that the benefits of object orientation are greater and better proven than those of component-based software, and most of the valuable features of components can also be realized by means of objects [2]. In terms of OO design, a component is "a black box that defines a cohesive set of operations, which can be reused based solely upon knowledge of the syntax and semantics of its interface" [10]. A well-designed object with the contract specified by a regular Java interface can conform to this definition as well.

The Spring Framework aims to be as less invasive as possible. It does not force managed objects to have any dependencies on framework-specific interfaces or classes unless needed. Each lifecycle callback method is defined by a single interface: `org.springframework.beans.factory.InitializingBean` to be implemented by managed objects that need to react once all their properties have been set by the Spring container and `org.springframework.beans.factory.DisposableBean` to be implemented by those that want to release resources on destruction. This allows managed objects to implement only the callback interface they are interested in. Moreover, for every managed object selected methods can be marked as callback methods declaratively in a configuration file, or in the source code via annotations introduced by the EJB 3.0 specification (see further in this section), without the need for implementing any callback interface at all. An example of a Spring bean is shown in figure 4.3.

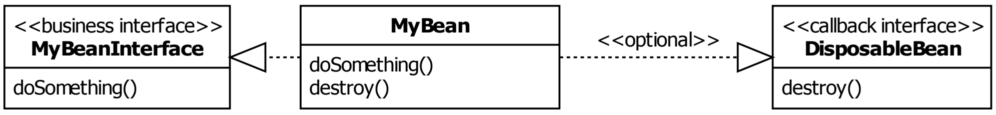

Figure 4.3: Spring bean contract

Following the concept of POJOs, the EJB 3.0 specification has simplified as the client-bean as the container-bean contract. To define the business contract of session beans, it has introduced a new type of interface, a *business interface*. This interface is a regular Java interface without any dependency on other EJB-specific interfaces, such as `EJBObject` or `EJBLocalObject`. Every session bean can define several business interfaces annotated as `@Remote` or as `@Local`, depending on the type of target clients.

EJB 3.0 beans are also no more tightly coupled with the callback interface. Provided with the appropriate callback-method annotation, `@PostConstruct` or `@PostDestroy`, lifecycle callback methods can be specified directly in the bean class. Furthermore, it is possible to define callback methods on a separate interceptor class associated with the bean (see section 5.1). As a result of these changes, an EJB 3.0 session bean class has to implement only one or more business interfaces (see code snippet 4.1); other interfaces specified by EJB 2.1 are optional and remain for compatibility reasons.

```
// remote business interface
@Remote
public interface MyBeanInterface {

  public void doSomething();

}

// stateless session bean
@Stateless
public class MyBean implements MyBeanInterface {

  public void doSomething() {...}

  // called before the bean instance is removed by EJB container
  @PreDestroy
  public void destroy() {
    // some clean-up code
  }
}
```

Code Snippet 4.1: EJB 3.0 remote session bean contract

Furthermore, the upcoming EJB 3.1 specification has dropped the business interface requirement. Since EJB 3.1, session beans do not need to implement any interfaces at all; they can be just simple Java classes.

4.1.2 Restrictions

EJB and Spring also very differ in the amount of programming restrictions, which the developer has to follow when implementing a managed object.

Although from the developer's point of view EJB 3.0 beans are POJO-style objects, which do not need to implement any specific interfaces, basically, they are the same components as they were in EJB 2.1 with the same programming restrictions. These restrictions may be divided into those relating to the implementation of the contract and those relating to the implementation of business logic.

At deployment time, for each bean, deployment tools provided by the EJB container generate additional classes depending on the structure of the bean, such as a class that implements the home interface, a class that implements the component interface, a class that implements the business interface, and a class that mixes some container-specific code with the bean class, to help the container to manage bean instances at runtime. The tools can use subclassing, delegation, and code generation. The developer does not have to prevent the EJB container from accomplishing this task, for example by declaring the bean class as `final` (it is not possible to subclass such a class).

When implementing business logic, the developer does not have to use programming practices that would interfere with the runtime management of bean instances, especially those that would conflict with services provided by the container to beans. For example, a bean does not have to use thread synchronization primitives to synchronize execution of multiple instances because it would not work if the EJB container distributed instances of the bean across multiple JVMs, a bean does not have to load a native library because it would compromise security, and more (see [13]). Moreover, the programmatic restrictions make it problematic to implement some common design patterns, such as the *Singleton* design pattern [6] which is hard to achieve due to the restrictions around read-write static variables.

While the EJB specification strictly defines a lot of programming restrictions that should not be violated to ensure that beans are portable and can be deployed in any compliant EJB container, Spring is less restrictive and it does not actually limit the use of any programming practices or Java language features. The developer does not usually need to take care of any programming restrictions unless it is required by a specific situation, as in the case of proxying a class that does not implement any interface via subclassing (see section 5.1).

4.1.3 Summary

As we have seen, the contract that application objects have to abide by in order to be managed by the Spring or EJB container is very similar; both frameworks allow for management of arbitrary POJOs. Speaking of programming restrictions, the situations is completely different. EJB strictly specifies which Java language constructs and facilities are forbidden. By contrast, the way taken by Spring is less restrictive, allowing developers to utilize any Java language features as needed. Nevertheless, this is not surprising since EJB is a specification whereas Spring is not.

Feature	Spring 2.5	EJB 3.0
Type of managed object	POJO	POJO
Business contract (client-side contract)	regular Java interface recommended	regular Java interface - bean business interface
Lifecycle management contract (container-side contract)	optional, callback interfaces, metadata configuration	optional, interceptors, metadata configuration
Structure restrictions (constructor, final, etc.)	no limitations (as required by the situation)	strictly defined
Business logic implementation restrictions (usage of Java language features)	no limitations (as required by the situation)	strictly defined

Table 4.1: Managed object contract overview

4.2 State Management

Essentially, every application consists of two types of business objects: *stateless* and *stateful*. While stateless business objects can be used by any client because they do not retain any information between two separate method calls, stateful business objects are intended for maintaining a conversational state and client-specific data. An application framework should support both types of business objects. In addition to what has just been said, a stateful object is often associated with a context, for example a long-running business process, an HTTP session, or a single web request, which determines its lifetime. JEE applications often implement contextual state management manually, nevertheless, application frameworks can help with this effort as well.

4.2.1 Stateless and Stateful Objects

As already mentioned in section 3.2, the EJB specification defines two types of session beans for modelling stateless and stateful business objects: stateless and stateful session beans. The names of session beans suggest their purpose. While stateful session beans are created by the EJB container per client and are tightly coupled with them, stateless session beans are shared among all clients and can be invoked by any of them. The developer specifies the desired type of a session bean in an EJB deployment descriptor or can use class-level annotations, `@Stateless` and `@Stateful`.

In Spring, each bean has specified its *scope*. There are two basic bean scopes: *singleton* and *prototype*. When a bean is singleton-scoped, it means that only one shared instance of the bean will be created and managed by the container. By contrast, a new instance of a prototype-scoped bean is created every time a request for this bean is made. Hence, the prototype scope should be used for all beans that are stateful while the singleton scope should be used for stateless beans. The following code snippet illustrates the use of bean scopes, showing a piece of configuration which contains definitions of two beans, one singleton-scoped and the other prototype-scoped.

```
...
<bean id="shoppingService" class="spring.service.ShoppingService"
scope="singleton"/>
<bean id="shoppingCart" class="spring.model.ShoppingCart"
scope="prototype"/>
...
```

Code Snippet 4.2: Bean scopes

4.2.2 Holding State in a Context

Since Spring 2.0, its scoping mechanism is extensible, allowing for custom scopes. The developer can implement its own scope (context) determining the lifetime of beans that are bound to it. This mechanism is heavily utilized, for example, by the Apache MyFaces Orchestra project [53].

Besides two basic scopes, singleton and prototype, Spring provides out of the box three other scopes, namely *request*, *session*, and *global session*. Nevertheless, these additional scopes can be used only in web-based applications because they are an integration of the contexts defined by the Servlet and Portlet specifications [33], [34]: an HTTP request, an HTTP session, and a global HTTP session.

The EJB specification does not concern with the issue of contextual state management at all.

4.2.3 Summary

Both Spring and EJB define a model for creating stateless and stateful application objects. Nevertheless, as we will see further in sections 4.3 and 4.5, the models differ significantly in the way the containers manage the lifecycle of objects and deal with concurrency. Regarding scoping mechanisms, Spring provides great support for handling objects in the traditional JEE web contexts and allows for defining custom object scopes, whereas the EJB specification lacks any support.

Feature	Spring 2.5	EJB 3.0
Stateless objects	singleton-scoped beans	stateless session beans
Stateful objects	prototype-scoped beans	stateful session beans
Other object scopes	request, session, global session, custom scopes	none

Table 4.2: State management overview

4.3 Lifecycle Management

The main responsibility of any container is to control the life cycle of application objects running within it. At a minimum, it should abstract the creation of new objects from code using it. More sophisticated lifecycle management also includes callbacks to alert managed objects of their instantiation or destruction within the container. In this section we discuss the lifecycle management fundamentals of both the EJB and the Spring container, and take a closer look at important lifecycle phases of managed objects.

4.3.1 Fundamentals

Although the core design of an EJB container usually varies between server vendors, the main principles of lifecycle management of residing beans do not differ. What differs is the way how EJB containers manage the lifecycle of stateless and stateful session beans.

Because stateless session beans hold no conversational state, all instances of the same stateless session bean are equivalent, and therefore, any of the instances can serve any client request. This fact allows for pooling of stateless session bean instances by the EJB container. Every time a request for a stateless session bean is made, one of the pre-created bean instances is assigned to the client. If the client relinquishes control, the container returns the bean to the pool of available beans. Actually, stateless session bean instances can be swapped from one client to another client on each method call.

In contrast to stateless session beans, stateful session beans are created per client, thus, they are usually destroyed after the reference to the corresponding proxy object (EJB object) is released by the client. Nevertheless, some EJB containers may utilize the pooling mechanism for stateful session beans as well [11]. In place of creating a new instance every time needed, the container returns a pre-created stateful session bean from the pool. Similarly, deprived of the client-specific state information, unnecessary instances are put back to the pool.

The core architecture of Spring is based on so-called *bean factories* which are responsible for creating and configuring managed objects. Actually, a bean factory is the representation of the Spring container. The business model of a simpler application may be managed by one bean factory. More complex applications can use several bean factories, usually organized in a tree structure. If a bean is not found in the current factory, the parent factory is asked, up until the root factory.

A bean factory also acts as a repository for singleton-scoped beans. Such beans are instantiated when the bean factory is created or at the time the first request for the bean is made (lazy instantiation), and exist till the bean factory is destroyed. While bean factories manage the complete lifecycle of singleton-scoped beans, in the case of prototype-scoped beans the situation is slightly different. The bean factory instantiates, configures, decorates, and otherwise assembles a prototype-scoped bean, hands it to the client, and then has no further knowledge of that prototype instance. Any other lifecycle aspects have to be handled by the client itself.

4.3.2 Creating Managed Objects

On the basis of information obtained from the reference documentation of both frameworks [13], [15], especially from the lifecycle diagrams of managed objects, we can extrapolate that managed objects are created in three general steps, regardless of which framework we use (see figure 4.4). First, the container instantiates an object via a constructor, then performs unnecessary configuration (see section 4.4.2), and finally invokes a callback method (see section 4.1) defined by the object, allowing for custom initialization. Let us look at the instantiation phase in more detail.

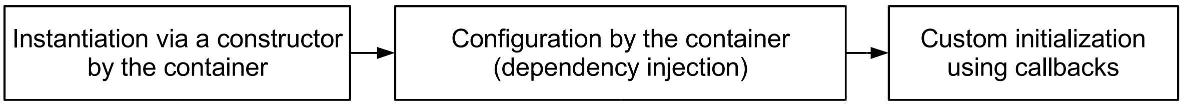
Instantiation via a constructor by the container
Configuration by the container (dependency injection)
Custom initialization using callbacks

Figure 4.4: Process of creating managed objects

While the EJB container instantiates managed objects via the required default constructor, the approach of Spring is more superior. In Spring, a bean can be instantiated in various ways:

- **Custom constructor:** The bean factory can instantiate a bean via a common constructor. This is achieved by means of the Java Reflection API [35] and constructor dependency injection discussed further in this chapter (see section 4.4.2).

- **Factory method:** A bean can be instantiated via a *factory method* defined on another existing bean or the bean itself. Calling this method, the bean factory expects that a live instance of the bean will be returned. This approach has been formalized in GoF [6] as the *Factory Method* pattern.

- **Factory bean:** Implementing the `org.springframework.beans.factory.FactoryBean` interface, a bean can act as a factory for another bean. A *factory bean* completely conceals the way how beans are created; it may instantiate the bean via a constructor, obtain the bean by a JNDI lookup, return a proxy object in place of the bean instance, etc. This concept is heavily used within Spring itself, for example by its AOP framework for creating advised beans. This way of instantiating objects is known as the *Abstract Factory* pattern [6].

In addition to initialization callback methods discussed in section 4.1, Spring provides an advanced *bean post-processor* mechanism for custom modification of new bean instances created by a bean factory. Central in this concept is the `org.springframework.beans.factory.config.BeanPostProcessor` callback interface, mandatory for all bean post-processor implementations. When a bean post-processor is registered with a bean factory, for each bean instance that is created by the factory, the post-processor will get a callback from the factory before and after the bean is being initialized. This allows the post-processor to perform a custom initialization and/or instantiation logic, for example, it can populate the bean via marker interfaces or wrap it with a proxy.

4.3.3 Activation and Passivation of Stateful Managed Objects

The issue with stateful objects is that they consume a lot of system resources, especially physical memory. Thus, to prevent an application, or an application server, from being run out of fixed amount of system resources, its necessary to limit the number of stateful objects in memory.

To achieve this goal, the EJB container can save the state of idle stateful session beans to a hard disk or other secondary storage, allowing resources like memory to be reclaimed; the "empty" bean is destroyed or returned into the pool to be reused. This process is known as *passivation*. When the original client invokes a method, the EJB container loads the passivated conversational state into a bean, newly created or obtained from the pool, which resumes the conversation with the client. This is called *activation*. The EJB specification strictly defines which member variables are considered to be part of the bean's conversational state.

As mentioned above, Spring does not manage the complete lifecycle of prototype-scoped beans, hence, it does not address this issue. In the case of custom-scoped beans, the responsibility is delegated to the owner of the scope context which the beans are bound to, for example, a web server takes care of the beans scoped to an HTTP session object.

4.3.4 Destroying Managed Objects

Considering both containers, the process of destroying fully-managed objects (stateless and stateful session beans, singleton-scoped beans) is similar and can be generalized into two steps (see figure 4.5). First, the container notifies the object of being removed by invoking a callback method. This method is usually responsible for freeing all resources that the object might have allocated earlier in the lifecycle and need to be cleaned up. After the callback method ends, the container releases all held references to the object and let the Java garbage collector do the rest.

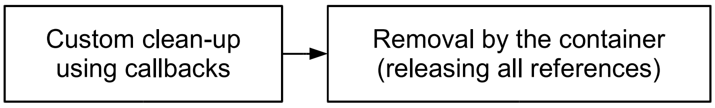
Custom clean-up using callbacks
Removal by the container (releasing all references)

Figure 4.5: Process of destroying managed objects

4.3.5 Summary

To sum up, the key difference between the EJB and Spring container is that the EJB container manages the complete lifecycle of managed objects while the Spring container rather concentrates on their creation and configuration, and let clients manage other lifecycle phases. From that point of view, it is not surprising that the ways a managed object can be instantiated and initialized within the Spring container are really immense in contrast to EJB.

Feature	Spring 2.5	EJB 3.0
Objects being completely managed by container	singleton-scoped beans	all beans
The ways of creation	default and custom constructor, static factory method, factory bean	default constructor
The ways of post-creation (initialization, configuration)	callback method, bean post-processors	callback method
The ways of activation and passivation of stateful objects	not supported	callback methods
The ways of pre-destruction (clean-up)	callback method (pre-destruction of the bean factory/container)	callback method (pre-destruction of the bean)

Table 4.3: Lifecycle management overview

4.4 Dependency Management

Applications are typically composed of various objects that collaborate with each other to form the application proper. We say that the objects have dependencies between themselves. Moreover, an application object may also be dependent on disparate environment resources like data sources, or even simple configuration parameters, such as number and string values, used to specify a particular behaviour of the object at runtime. The containers address this issue by providing a mechanism based on the *Inversion of Control* (IoC) principle [29].

Inversion of Control is a broad concept that can be implemented in different ways. There are two primary forms [2]: *dependency lookup* and *dependency injection*. Dependency lookup is a pattern where a caller use a container-specific API to asks for an object with a specific name or of a specific type. Dependency injection is a pattern where the container passes objects by name or by type to other objects, either via constructors, fields, or setter methods. The basic overview of IoC is shown in the following figure.

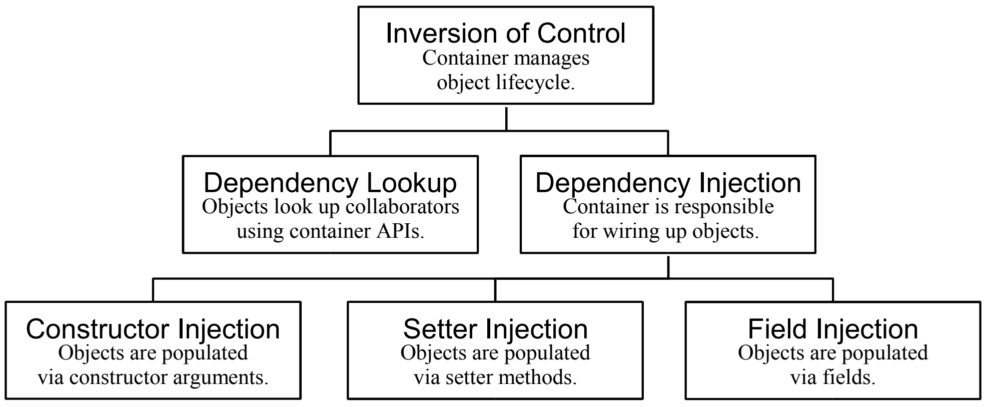
Inversion of Control
Container manages object lifecycle.
Dependency Lookup
Objects look up collaborators using container APIs.
Dependency Injection
Container is responsible for wiring up objects.
Constructor Injection
Objects are populated via constructor arguments.
Setter Injection
Objects are populated via setter methods.
Field Injection
Objects are populated via fields.

Figure 4.6: Different types of Inversion of Control

The dependency lookup paradigm is useful in certain circumstances, especially in distributed applications to locate objects spanned over the network, nevertheless, it is not suitable for dependency resolution and configuration of fine-grained objects in collocated applications; in such a situation, the dependency injection approach has been considered to be more appropriate [1], [2]. The container takes responsibility for wiring application objects together while the objects can concentrate more on business logic. As a result, the source code of application objects is not burdened with a container-specific lookup API which makes the objects more reusable. Moreover, because dependencies between application objects are usually specified declaratively, the application can be reconfigured without the need for source code modification.

4.4.1 Dependency Lookup

Every component-based architecture has to concern with the issue how components are published and further located by clients. We have already mentioned that for this purpose the EJB architecture harness the JNDI service (see section 3.2.3), which is essentially an implementation of the dependency lookup concept; published beans are looked up by clients via the JNDI API. In fact, the JNDI service serves for publishing any resources and configuration parameters needed by beans, not only beans themselves.

Beans, external resources, and configuration parameters a bean depends on make up the *bean's environment*. All these environment entries that the bean expects to be provided in its environment at runtime have to be declared in a deployment descriptor or, since EJB 3.0, in Java language metadata (annotations). Code snippet 4.3 illustrates the configuration of the `ShoppingService` bean which collaborates with the `StockService` bean through its remote business interface, the `IStockService` interface.

```
...
<session>
  <ejb-name>ShoppingService</ejb-name>
  ...
  <!-- dependency on the StockService bean -->
  <ejb-ref>
    <!-- JNDI name of the collaborator -->
    <ejb-ref-name>ejb/StockService</ejb-ref-name>
    <!-- business interface of the collaborator -->
    <remote>ejb.service.IStockService</remote>
  </ejb-ref>
</session>
...
```

Code Snippet 4.3: Configuration of bean's environment in deployment descriptor

An important concept of the JNDI architecture is a *JNDI naming context* which groups together logically related *bindings*, i.e., associations of user-friendly names with objects. For each deployed bean, the EJB container has to provide an implementation of the JNDI naming context that stores the bean's environment, and make the context available to the bean at runtime. Then, the bean's methods are able to access any environment entry by using the JNDI API; this process is illustrated in code snippet 4.4 (assuming the configuration in code snippet 4.3).

```
...
// obtain the initial JNDI context
Context initCtx = new InitialContext();
// obtain the bean's environment context
Context envCtx = (Context) initCtx.lookup("java:comp/env");
// lookup the remote business interface of the StockService bean
// in the bean's environment
IStockService stockService =
  (IStockService) envCtx.lookup("ejb/StockService");
...
```

Code Snippet 4.4: Dependency lookup in EJB

In Spring, all beans that collaborate with each other are usually configured to be managed by a single bean factory or another bean factory positioned higher in the hierarchy. A bean can be not only a regular application business object, but also any resource object, such as a database connection pool, or simple Java type; from Spring's perspective, everything is a bean. At runtime, a bean can ask the parent bean factory for any managed bean by name (id) through its client-view interface, the `org.springframework.beans.factory .BeanFactory` interface, as shown in code snippet 4.5.

```
...
<!-- configuration of the StockService bean -->
<bean id="stockService" class="spring.service.StockService"
scope="singleton"/>
...

...
// obtain the parent bean factory
BeanFactory beanFactory = ...
// lookup the StockService bean
// which implements the IStockService interface
IStockService stockService =
  (IStockService) beanFactory.getBean("stockService");
...
```

Code Snippet 4.5: Dependency lookup in Spring

Another issue is how a bean acquires a reference to its bean factory. There are many ways how this can be achieved. For example, beans can implement the `org.springframework.beans.factory.BeanFactoryAware` interface, enabling Spring to inject the parent bean factory to the bean at the time of initialization.

4.4.2 Dependency Injection

The dependency injection capabilities of Spring have belonged to its most significant features since its first versions. The dependency injection mechanism is tightly coupled with bean factories. Using the Java Reflection API, a bean factory is able to inspect any managed bean and inject the bean's dependencies via constructor arguments at the instantiation phase or via appropriate setter methods at the initialization phase. The field injection concept has not been supported up to Spring 2.5. This was partly because field injection violates the object encapsulation principle of OOP; on the other hand, it does not pose any requirements on application objects as the other types of dependency injection do (setters methods, constructors).

To shield beans from cumbersome JNDI lookups, the EJB 3.0 specification has introduced its own dependency injection mechanism. In contrast to Spring, EJB provides support only for field and setter injection; the EJB container can be configured to initialize fields or call setter methods when it creates the bean. This way any bean's environment entry can be injected into the bean transparently without using the JNDI API.

In fact, the dependency injection concept has been incorporated into the entire JEE platform since version 5.0. JEE 5.0 defines a set of managed classes like enterprise bean classes, application client classes, servlets, and JSF managed beans which can take advantage of dependency injection. Because of this limitation, dependency injection can not be used by arbitrary POJOs, for example helper classes. This means that such objects have to be implemented as EJB beans to be configured via dependency injection by the EJB container.

In EJB, application dependencies are configured via built-in Java annotations (see code snippet 4.6). By contrast, Spring supports as source code metadata as XML-based configuration (see code snippet 4.7).

```
@Stateless
public class ShoppingService implements IShoppingService {

  // the @EJB annotation tells the EJB container to initialize
  // those fields with references to collaborators
  @EJB
  IStockService stockService;

  ...
}
```

Code Snippet 4.6: EJB field injection

```
public class ShoppingService implements IShoppingService {

  IStockService stockService;

  // constructor
  public ShoppingService(IStockService stockService) {...}

  ...
}

...
<!-- configuration -->
<bean id=”stockService” class=”spring.service.StockService”
scope=”singleton”/>

<bean id=”shoppingService” class=”spring.service.ShoppingService”
scope=”singleton”>
  <!-- constructor injection -->
  <constructor-arg ref="stockService"/>
</bean>
...
```

Code Snippet 4.7: Spring constructor injection

4.4.2.1 Dependency Injection of Stateful Objects into Stateless Objects

An important issue with the dependency injection concept is how to resolve dependency between objects with different lifecycles. Let us consider a stateless object A which needs to use a stateful object B, perhaps on each method invocation on A. The container (no matter whether Spring's bean factory or the EJB container) will only create the stateless object A once, and thus only get the opportunity to set the properties once. There is no opportunity for the container to provide object A with a new instance of object B every time is needed.

A common solution to this issue is to use the dependency lookup approach as described in the previous section. Although this is not desirable solution since the business code is coupled with the container-specific lookup API, EJB does not offer another one. Spring addresses this issue in the following ways:

- **Lookup method injection:** Using bytecode generation via the CGLIB library [55], Spring can override methods on managed beans to return the result of looking up another named bean in the container, typically a prototype-scoped bean. The signature of such a method has to be of the following form: `<public|protected> [abstract] <return-type> theMethodName(no-arguments)`.

- **`ScopedProxyFactoryBean`:** Clients do not work with custom-scoped beans directly. Instead, they are injected with their proxy objects created by the appropriate `org.springframework.aop.scope.ScopedProxyFactoryBean` (for more information about factory beans see section 4.3.2). If a method is invoked by the client, the proxy will fetch the real custom-scoped bean from the underlying scope context that is coupled with the client, and delegate the method invocation onto the retrieved bean.

4.4.2.2 Autowiring

In a lot of cases, explicit specification of dependencies between managed objects is the recommended way. However, Spring also offers a different way of dependency resolution called *autowiring*. If a bean is marked as autowired, Spring will resolve collaborators for the bean automatically via matching beans in the bean factory, without the need to define the bean's dependencies explicitly. Spring also allows for fine-tuning autowirng on field, method, and constructor level via annotations `@Autowired` and `@Qualifier`. There are several autowiring modes:

- **By name:** Autowiring by property name. This option will inspect the bean factory and look for a bean named exactly the same as the property which needs to be autowired. For example, if a bean contains a `stockService` property, Spring will look for a bean named `stockService`, and use it to set the property.

- **By type:** Autowiring by property type. This option allows a property to be autowired if there is exactly one bean of the property type in the bean factory. If there is more than one, a fatal exception is thrown, indicating that this autowiring mode for that bean may not be used. This problem can be avoided, for example, by configuring a single bean definition as the primary candidate for autowiring.

- **Constructor:** This is analogous to autowiring by type, but applies to constructor arguments.

Although autowiring can significantly reduce the amount of configuration required, the developer should be aware of the implications of automatic wiring, particularly when defining a new bean which implements the same interface as an existing one.

Furthermore, Spring also has the ability to check that all JavaBean properties (dependencies for which the corresponding setter method are defined) of a bean have been resolved. This feature, called *dependency checking*, is particularly useful in combination with autowiring.

4.4.3 Summary

In this section, we have familiarized ourselves with the dependency management mechanisms of the frameworks. As we have seen, both Spring and EJB support the dependency injection as well as dependency lookup concept. However, it is evident that Spring's DI capabilities are superior to those provided by EJB. To some degree it is a result of the fact that EJB has originally been aimed at building distributed applications, therefore JNDI is well-suited for locating coarse-grained components elsewhere on the network, rather than for configuring fine-grained business objects within applications.

Feature	Spring 2.5	EJB 3.0
Registry/repository of managed objects and their dependencies	bean factory	JNDI
Lookup API	bean factory lookup interface	JNDI API
Supported DI types	field, setter, constructor	field, setter
DI of stateful objects into stateless objects	lookup method injection, scoped proxy factory beans	not supported
Autowiring	by name, by type, constructor	not supported
Dependency checking	supported	not supported

Table 4.4: Dependency management overview

4.5 Concurrency Management

Being accessed by multiple clients, application objects like business service objects cannot hold state on behalf of an individual client. However, they can hold an internal state; for example, a business service object can maintain a cache of resources to avoid repeated lookups as it services incoming requests. Since a lot of threads can operate on a business service object simultaneously, it is necessary to prevent the object's state from being corrupted. There exist several concurrency strategies, based on different *threading models* and object *pooling*, addressing this issue. Let us see which ones are used by EJB and Spring.

4.5.1 Threading Models

EJB aims to allow developers to write beans without concerning concurrency control; the EJB component model has been designed as *single-threaded*. A bean instance is locked if there is a thread of execution in it. In other words, each bean instance is dedicated to one client, and hence to one thread at a time (unless it is idle). Since a bean instance can execute only one method at a time, its internal state cannot be handled by multiple threads simultaneously, and thus be corrupted by concurrent access.

However, business service objects do not usually need to use a single-threaded model because they do not run into complex concurrency issues [2]. Having no read-write variables, most of them do not encounter concurrency issues at all. For this reason Spring does not lock managed objects into a specific threading model. If there is a need to protect an internal variable of an object from concurrent access, the developer has to implement the object to be *thread-safe* using Java synchronization primitives. The architects of Spring claim that a shared, thread-safe single object is the best threading model for most service objects.

4.5.2 Pooling

Pooling instances of objects is essentially an implementation strategy used to provide a single-threaded programming model to developers. We have already discussed (see section 4.3.1) that the EJB container maintains for each stateless session bean a pool of instances to which client requests are delegated. While some bean instances are locked on behalf of a particular client, the other in the pool remain free to serve incoming requests. Hence, an efficient pooling of single-threaded beans is necessary to achieve good performance results.

The big advantage of pooling is that a small number of object instances can serve a large number of clients. There are available several good pooling libraries, including Jakarta Commons Pool [56], which allow for pooling of arbitrary Java objects. Spring builds on such libraries and provides transparent instance pooling services to any managed bean.

The issue with pooling is that clients may not return objects to the pool, usually because of an programmer error in the code. Hence, the pool manager has to be concerned with the restoration of abandoned pooled objects. To avoid this issue, Spring take advantage of its AOP capabilities and conceals the process of acquiring and releasing pooled objects into AOP proxies. When a client invokes a method on an AOP proxy, the proxy obtains an instance of the target object from the pool, and delegates the method invocation to this object. After the object completes its work, the proxy releases it and returns control to the client. Actually, holding AOP proxies, clients do not need to be aware of pooling at all, although they can take for granted that their objects will never have more than one thread executing them.

In its spirit of openness and pluggability, Spring allows for integration of any pooling library and provides the `org.springframework.aop.target.AbstractPoolingTargetSource` superclass for pool managers that can maintain a pool of target object instances. Out of the box, Spring offers the implementation for the Jakarta Commons Pool library. Further, it is possible to examine the behaviour of the pool at runtime via the `org.springframework.aop.target.PoolingConfig` interface implemented by `AbstractPoolingTargetSource`, and set the maximum size of the pool as needed. By contrast, EJB containers do not usually offer any means to modify the bean pool size at runtime.

Other special pooling strategy supported by Spring is *thread-local pooling* based on thread-local variables. These variables differ from their normal counterparts in that each thread has its own, independently initialized copy of the variable. Thus, when the thread-local pooling strategy is used, each thread of execution have its own copy of the “pooled” object. If a client makes repeated calls on an AOP proxy of a thread-local pooled object, the proxy will operate on the same instance of the object bound to the thread all the time. Obviously, such a pooled object can retain its internal state during the operations of a single thread. In the case of EJB, this pooling model is not possible to support since a single request can span multiple beans running in various threads.

4.5.3 Summary

As can be seen, Spring and EJB address concurrency management in completely different ways. While EJB aims to shield developers from concurrency issues providing the single-threaded component model, Spring advocates the concept of shared, thread-safe business objects allowing developers to use Java synchronization facilities if needed. Concerning object pooling, the EJB container pools beans by default, without the possibility of changing this behaviour, whereas Spring supports a wide range of pooling strategies to choose from when pooling of managed objects is required.

Feature	Spring 2.5	EJB 3.0
Threading model	shared, thread-safe objects	single-threaded objects
“Standard” object pooling	supported	supported by default
Switching between various pooling providers	supported	not supported
Other pooling strategies	thread-local pooling	none

Table 4.5: Concurrency management overview

5 Services

The middle tier of an application usually has a lot of responsibilities, thus, it is often advantageous to divide it into several *layers*, as shown in figure 5.1. Each layer provides functionality for a particular area, such as business services, data access, and remoting. *Aspects* like security, transactions, and logging weave through the layers.

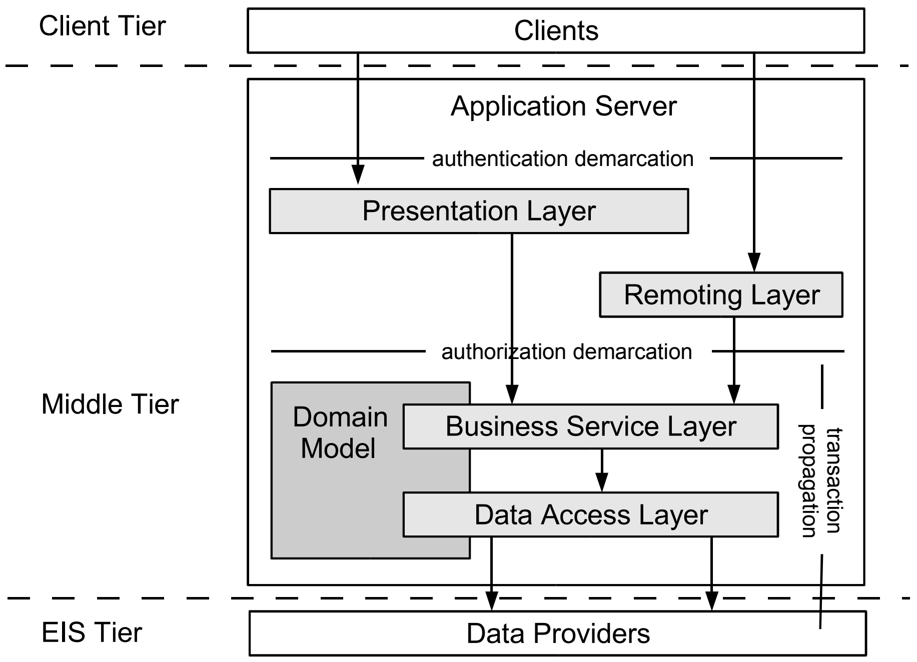

Figure 5.1: Middle tier

The previous chapter deals with the core functionality of both frameworks related to the management of business service objects which form the *business service layer*. The goal of this chapter is to analyse how Spring and EJB allow for building other middle-tier layers and how they facilitate the usage of standard JEE services such as JTA, JDBC, RMI, JMS, etc. We examine their support for data access, transaction management, remoting, messaging, security, and scheduling. Since the provision of middleware services is often implemented via AOP, we start this chapter with a description of their AOP capabilities.

5.1 AOP

Spanning across various application objects, middleware services are typical cross-cutting concerns which may be modularized into AOP aspects. Regarding this fact, application frameworks usually adopt the AOP concept to some degree. They often provide a basic set of services, such as transaction and security service, already modularized, and further allow for creating custom, application-specific ones. Let us discuss how AOP is utilized by EJB and Spring.

5.1.1 Fundamentals

While EJB tightly integrates the provision of middleware services into the core component model, Spring has adopted the AOP concept and separates middleware services from application objects with the same benefits of declarative middleware as EJB. Moreover, the AOP approach does not limit developers to use only common middleware services provided by Spring, but they are free to implement custom, reusable services according to the special requirements of an application. Since version 3.0, the EJB specification has introduced its own AOP mechanism based on *interceptors*, and thus made possible to design custom services as well.

In Spring, aspects are implemented via regular Java classes. An aspect class defines an advice that has to be taken at a particular join point during the execution of an application. A pointcut expression that determines matching join points can be assigned to an advice via an XML configuration file or annotations. A disadvantage of the XML-style configuration is that the knowledge of how an aspect is implemented is split across the backing class and the configuration file. By contrast, when annotations are used, the implementation of an aspect is fully encapsulated in a single module, in compliance with the *DRY* (Do not Repeat Yourself) *principle* [21]: “Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system.” As an example, a `LoggingAspect` is shown in the following code snippet.

```
@Aspect
public class LoggingAspect {

  // around advice
  // log the execution of any method that matches to the specified
  // pointcut expression - "execution(* spring.service..*.*(..))",
  // i.e., all methods in the spring.service package or any sub-package
  @Around("execution(* spring.service..*.*(..))")
  public Object log(ProceedingJoinPoint pjp) throws Throwable {
    // add code here to log the start of the method execution
    Object returnValue = pjp.proceed(); // proceed to the join point
    // add code here to log the end of the method execution
    return returnValue;
  }

}
```

Code Snippet 5.1: Aspect definition in Spring

EJB interceptors are also regular Java classes, however, the programming restrictions that apply to enterprise beans apply to interceptors as well (see section 4.1.2). In contrast to Spring, an aspect represented by an interceptor class does not make up a self-contained module; an interceptor class contains an advice, but does not hold the specification of join points that the advice has to be applied to. Interceptor classes are defined for a bean by means of annotations or in the deployment descriptor. For a comparison with the Spring-based `LoggingAspect`, code snippet 5.2 illustrates the `LoggingInterceptor` definition and its application to the `doSomething()` method of the `ShoppingService` bean.

```
// EJB interceptor
public class LoggingInterceptor {

  // around advice
  @AroundInvoke
  public Object log(InvocationContext ctx) throws Exception {
    // add code here to log the start of the method execution
    Object returnValue = ctx.proceed(); // proceed to the join point
    // add code here to log the end of the method execution
    return returnValue;
  }

}

@Stateless
public class ShoppingService implements IShoppingService {

  // interceptor declaration for the doSomething() method
  @Interceptors(ejb.interceptor.LoggingInterceptor.class)
  public void doSomething() {...}

}
```

Code Snippet 5.2: Interceptor definition in EJB

After a brief introduction, let us take a closer look at three selected topics: joint point models, advice types, and weaving implementation.

5.1.2 Join Point Models

Neither the Spring AOP framework nor the EJB interceptors aim to be the most complete AOP implementation like, for example, AspectJ [44]. Thus, they both only support *method execution join points*; field interception is not implemented. At runtime, join point information, such as the method name and arguments, is available in advice bodies via an appropriate join point object representation (`org.aspectj.lang.ProceedingJoinPoint`, `javax.interceptor.InvocationContext`) that is passed in as a method parameter (see code snippets 5.1 and 5.2).

To specify a set of matching join points for a given advice, AOP frameworks are usually equipped with a pointcut language. Spring provides a partial integration of the AspectJ pointcut language and allows to write pointcut expressions by using a subset of AspectJ *pointcut designators*. In code snippet 5.1, we have already seen the *execution* designator for matching method execution join points. Beside this one, there are several other useful designators, such as *within* (limits matching to join points within certain types), *args* (limits matching to join points where the arguments are instances of the given types), etc. The basic pointcut expressions can be combined into more complex ones by using standard logic operators, “AND”, “OR”, and “NOT”. Detailed information about the language syntax and semantics can be found in the Spring reference documentation and AspectJ programming guide.

The EJB interceptor concept does not offer any pointcut language. The only way how to specify where an interceptor will be invoked is to mark all affected methods and classes (in order to intercept all methods defined within the class) with the `@Interceptors` annotation, or provide the equivalent XML configuration. Such an explicit interceptor declaration for each method is at least cumbersome and it is easy to forget to annotate some methods. Further, the `@Interceptors` annotation is a simple tag, having no advanced semantics. For example, it is not possible to check the parameters of an advised method if they are of expected types; the need for an explicit type control makes the advice code more error-prone.

5.1.3 Advice Types

There exist a wide range of advice types to run the advice code before, after, and around join points. However, all of them are special kinds of the most general one, *around advice*, which can perform custom behaviour as before as after the join point, having control over the execution flow proceeding to the join point (see code snippets 5.1 and 5.2). While EJB offers only this type of advice, Spring provides several specific advice types, such as *before advice*, *after returning advice*, *after throwing advice*, and *after (finally) advice* (see section 3.3.3), to allow developers to use the least powerful advice type that can implement the required behaviour. To conform to the central around advice type based on the AOP Alliance [42] `org.aopalliance.intercept.MethodInterceptor` interface, all the specific advice types are wrapped by Spring via corresponding adapters.

Another special advice type supported by Spring is an *introduction*, also known as an *inter-type declaration*. Any managed objects can be enriched with additional methods or fields by introducing new interfaces and a corresponding implementation. This style of object composition where units of functionality are created in a class and then mixed in with other classes is called *mixin programming* [59]. As shown in figure 5.2, Spring treats introduction advice by means of interception.

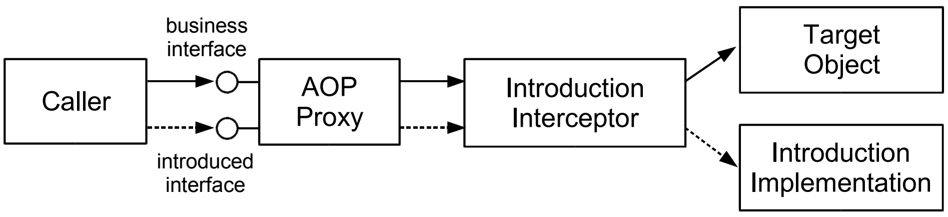

Figure 5.2: Introduction advice

When creating an AOP proxy for a given managed object, Spring inspects the implementation of the required introduction for all implemented interfaces and exposes them on the AOP proxy. Every time a client calls a method on the AOP proxy, the call is delegated to the appropriate `org.springframework.aop.IntroductionInterceptor`, which delegates the call to the target object, or more precisely to the introduction implementation, depending on whether the method call has been made through the business or introduced interface; since the `IntroductionInterceptor` extends the `MethodInterceptor` around advice, it is capable to perform such a task. Obviously, an introduced interface will conceal any implementation of the same interface by the target object.

5.1.4 Weaving Implementation

The same AOP language can be implemented through a variety of weaving techniques, thus, the semantics of an AOP language should never be understood in terms of the weaving implementation. However, the weaving approach ultimately affects the usability of an implementation; how fast it is and how aspects are deployed. Basically, the process of weaving can be done *at compile time* (requiring an additional preprocessing step, for example), *at load/deployment time* (for instance, by utilizing Java class-loading feature), or *at runtime*.

Many AOP frameworks, including Spring, model an advice as an interceptor, implementing a runtime weaving. For each join point (i.e., for each method) declared on the target object, the corresponding AOP proxy maintains a chain of matching interceptors. When a client calls an advised method on the AOP proxy, first, all the appropriate interceptors in the chain are executed, and then an implementation of the `org.springframework.aop.TargetSource` interface is used to obtain the target object containing the join point, which is invoked via reflection. The default `TargetSource` implementation holds a reference to a single target instance that is invoked with every method call. For example, more advanced `TargetSource` implementations allow for "standard" pooling of target object instances or thread local pooling by means of thread local variables (see section 4.5.2). Best of all, an AOP proxy can be configured to use another `TargetSource` implementation without changing a line of client code.

5.1.4.1 Proxying Mechanisms

AOP proxies are created by `org.springframework.aop.framework.Proxy FactoryBean`, which is an implementation of the `FactoryBean` interface discussed in section 4.3.2. Spring can use two different techniques for creating AOP proxies at runtime: *CGLIB* or *JDK dynamic proxies*.

JDK dynamic proxies are a Java language construct that allow for creating a type-safe proxy object for a list of interfaces without requiring pre-generation of the proxy class, such as with compile-time tools. A method invocation through one of the interfaces on an instance of the dynamic proxy class is dispatched to the registered invocation handler by using the Java Reflection API [35]. Consequently, the invocation handler can delegate the method call to another object, in the case of AOP proxies, to an interceptor or the target object.

If the target object does not implement any interfaces, Spring will create a CGLIB proxy. CGLIB [55] is a powerful code generation library, which uses ASM bytecode manipulation framework under the hood. Essentially, CGLIB proxying works by generating a subclass of the target object. Spring configures this generated subclass to delegate method calls to the original target weaving in advice (interceptors); the subclass is used to implement the *Decorator* pattern [6]. Since this technique uses the object inheritance principle, final methods cannot be advised because they cannot be overridden.

5.1.4.2 Self-Invocation Problem

An important implication of the proxy-based weaving technique is a *self-invocation problem*. Once a method call has finally reached the target object, any method calls that the target object may make on itself will be invoked against the `this` reference, and not the AOP proxy. Therefore, the advice associated with a method invoked by the target object itself will not be given a chance to execute. To make things clear, let us consider an excerpt of the code for the `TargetObject` class in code snippet 5.3.

```
public class TargetObject {

  public void methodA() {...}

  public void methodB() {
    ...
    this.methodA(); // self-invocation
    ...
  }

}
```

Code Snippet 5.3: Self-invocation problem of the proxy-based weaving technique

Obviously, if a client calls `methodB()`, `methodA()` will also be invoked as a part of business logic of `methodB()`. However, because of self-invocation, no matching advice for `methodA()` will be executed. When implementing application objects, developers should keep in mind this issue and design them in such a way that the self-invocation does not happen.

Since EJB is a specification, application server vendors are free to use an arbitrary weaving technique to implement the EJB interceptor concept. Probably, considering the fact that clients also call beans via proxies, a runtime weaving similar to the Spring approach will be the most common implementation.

5.1.5 Summary

In this section we have covered AOP capabilities of the frameworks. Regardless of Spring's support for mixins, both frameworks offer the basic and most powerful advice type, around advice. In order to make the use of AOP as easy as possible, Spring goes even further providing a set of more specialized advice types. The main difference between Spring and EJB is in the way of specifying join points for a given advice. Spring's support for defining join points by using the AspectJ pointcut language is more superior to EJB's approach via annotations or deployment descriptor, allowing fine-tuned configuration. In addition, if the needs go beyond the facilities offered by its own AOP framework, Spring allows to utilize the AspectJ compiler and weaver to enable the use of the full AspectJ language.

Feature	Spring 2.5	EJB 3.0
Join point model	method interception	method interception
Pointcut language	AspectJ pointcut language integration	none
Supported advice types	around, before, after returning, after throwing, after (finally)	around
Mixins support	introductions	none
Weaving technique	runtime weaving	vendor-specific
Weaving implementation	proxy-based, interceptor-based	vendor-specific

Table 5.1: AOP overview

5.2 Data Access

Applications very often access some kind of persistent data, usually stored in a database or another persistent storage. In a modular, layered architecture, business logic is decoupled from persistence logic by defining an abstraction layer of persistence interfaces which business service objects can use. This approach is known as the *Data Access Object* (DAO) pattern [1], which is in fact a special case of the *Strategy* pattern [6].

Data access objects work with the underlying persistence technology, such as JDBC, Hibernate, JPA, etc., concealing any implementation details from clients. Thus, these interfaces can be reimplemented to use different data access strategies, without any changes cascading into the business service layer as illustrated in figure 5.3.

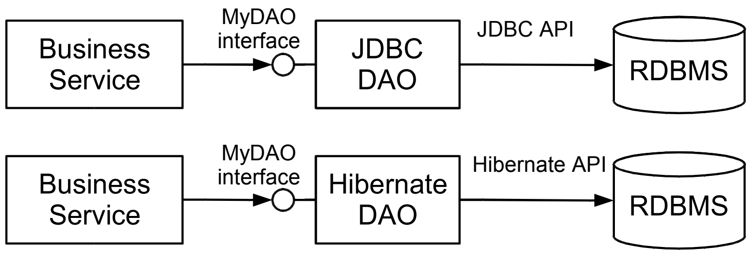

Figure 5.3: The Data Access Object pattern

Now let us turn our attention to Spring and EJB's support for building the data access layer.

5.2.1 Generic Data Access Exception Hierarchy

In comparison to EJB, Spring provides a convenient translation from technology-specific exceptions like `java.sql.SQLException` to its own technology-agnostic exception hierarchy with `org.springframework.dao.DataAccessException` as the root exception. This generic data access exception hierarchy is well-suited for designing abstract DAO interfaces, leaving business service objects without the knowledge of the particular data-access API in use [1]; for example, it is possible to react to an optimistic locking failure without knowing that JDBC is being used. Nevertheless, these exceptions wrap the original exceptions, thus, any information about the encountered error is not lost. A subset of the generic data access exception hierarchy provided by Spring can be seen below.

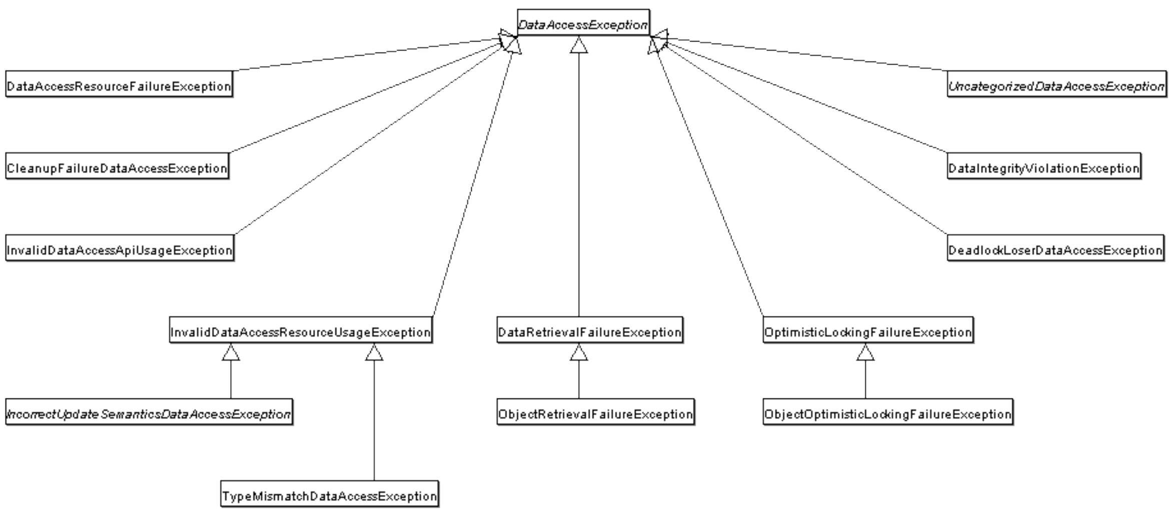
DataAccessException
DataAccessResourceFailureException
CleanupFailureDataAccessException
InvalidDataAccessApiUsageException
InvalidDataAccessResourceUsageException
IncorrectUpdateSemanticsDataAccessException
TypeMismatchDataAccessException
DataRetrievalFailureException
ObjectRetrievalFailureException
OptimisticLockingFailureException
ObjectOptimisticLockingFailureException
UncategorizedDataAccessException
DataIntegrityViolationException
DeadlockLoserDataAccessException

Figure 5.4: Generic data access exception hierarchy

Unlike `SQLExceptions`, which are basically checked exceptions, all generic data access exceptions defined by Spring are runtime (unchecked) exceptions. This allows developers to handle most persistence exceptions only in the appropriate layers, without polluting business logic code with `try/catch/finally` blocks to catch any error although it is to be considered fatal, i.e., non-recoverable.

Classes that can translate between JDBC exceptions and Spring's generic data access exceptions are supposed to implement the `org.springframework.jdbc.support.SQLExceptionTranslator` interface. Depending whether they are using portable SQL-state codes or vendor-specific (Oracle, MySQL, etc.) error codes provided by every `SQLException` to resolve matching exceptions, they can be generic or database-specific.

5.2.2 Data Access Using JDBC

Working with low-level APIs of a specific persistence technology is usually unpleasant. A good example is the JDBC API. There are a lot of common repetitious steps that have to be undertaken to accomplish even the simplest possible JDBC operation; this may cover: acquiring a connection, specifying the SQL query, creating/executing the statement, iterating over the result set, parsing the records returned, handling exceptions, and releasing the connection. As an illustration, a DAO method for querying some products of a given price is shown in the following code snippet.

```
private DataSource dataSource;

public List<Product> getProductsByPrice(int price)
  throws ApplicationException {

  final String query = "SELECT id, name FROM product WHERE price = ?";
  List<Product> products = new ArrayList<Product>();
  Connection conn = null;

  try {
    conn = dataSource.getConnection();
    PreparedStatement ps = conn.prepareStatement(query);
    ps.setInt(1, price);
    ResultSet rs = ps.executeQuery();
    while(rs.next()) {
      Product product = new Product();
      product.setId(rs.getLong("id"));
      product.setName(rs.getString("name"));
      products.add(product);
    }
    rs.close();
    ps.close();
  } catch(SQLException e) {
    throw new ApplicationException(
      "Could not run query: " + query, e);
  } finally {
    try {
      if(conn != null) {
        conn.close();
      }
    } catch(SQLException e) {
      throw new ApplicationException(
        "Failed to close connection.", e);
    }
  }

  return products;

}
```

Code Snippet 5.4: JDBC data access

It is evident that the resulting code is relatively complex and error-prone. A better solution than using the JDBC API is desirable. To simplify its usage and minimize the probability of making errors, Spring provides a JDBC abstraction framework.

Central in the JDBC abstraction framework is the `org.springframework.jdbc.core.JdbcTemplate` class. It encapsulates core JDBC workflow, taking responsibility for executing SQL queries, initiating iteration over the result set, catching JDBC exceptions, and translating them to Spring's generic data access exception hierarchy. Application code using this class only needs to provide SQL and implement necessary callback interfaces, giving them a clearly defined contract. The use of the `org.springframework.jdbc.core.simple.SimpleJdbcTemplate`, a Java-5-based convenience wrapper for the classic `JdbcTemplate`, is shown in code snippet 5.5.

```
private DataSource dataSource;

public List<Product> getProductsByPrice(int price) {

  final String query = "SELECT id, name FROM product WHERE price = ?";

  // create a template instance for the given DataSource to be accessed
  SimpleJdbcTemplate simpleJdbcTemplate =
    new SimpleJdbcTemplate(dataSource);

  // provide a row mapper implementation for mapping each row
  // of ResultSet to the result object
  ParameterizedRowMapper<Product> mapper =
    new ParameterizedRowMapper<Product>() {
      public Product mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        Product product = new Product();
        product.setId(rs.getLong("id"));
        product.setName(rs.getString("name"));
        return product;
      }
  };

  // perform the SQL query using the supplied row mapper
  // and the arguments for the query
  return simpleJdbcTemplate.query(query, mapper, price);

}
```

Code Snippet 5.5: JDBC data access using SimpleJdbcTemplate

As we can see, the template approach adds a lot of consistency to application code, especially when dealing with an API that uses checked exceptions. Beside common SQL CRUD queries, `JdbcTemplate` helps to perform tasks like batch processing, CLOB and BLOB data handling, and more. In addition to `JdbcTemplate`, Spring allows developers to model JDBC operations as Java objects, or access the database using SimpleJdbc classes taking advantage of database metadata that can be retrieved via the JDBC driver. More information about these approaches can be found in the Spring reference documentation.

EJB does not offer any similar facilities to simplify the use of JDBC at all.

5.2.3 Data Access Using ORM

As mentioned in section 3.4, the primary data access technology supported by EJB is the Java Persistence API (JPA) [14]. JPA follows the ORM approach providing many facilities taken from other popular ORM frameworks, particularly from Hibernate [16]. The architecture of JPA has been designed so that it makes possible to switch easy to another *persistence provider*, i.e., to another JPA implementation. Beside the TopLink Essentials project, the JPA reference implementation, there exist many other third-party implementations, for example Hibernate, TopLink, OpenJPA, and Kodo.

JPA allows for application domain modelling by lightweight persistent objects called *entities*. An entity is a POJO-style object that has to meet only a few requirements in comparison with the contract of EJB 2.1 entity bean. The basic deployment unit of entities is called a *persistence unit*. As the name suggests, the persistence unit is a set of entities that are managed together and have to be collocated in their mapping to a single database. At runtime, entity instances and their lifecycle are managed by `javax.persistence`

`.EntityManager` within a *persistence context*, a set of entity instances in which for any persistent entity identity there is a unique entity instance. The `EntityManager` provides methods to perform CRUD operations on entities, to retrieve and query entities, and to handle transactions. The following code snippet illustrates how an EJB bean utilizes JPA to persist domain objects.

```
@Stateless
public class StockDao implements IStockDao {

  @PersistenceContext
  private EntityManager entityManager;

  public void addProduct(Product product) {
    entityManager.persist(product);
  }

}
```

Code Snippet 5.6: The use of JPA in EJB

Unlike EJB, Spring provides integration with a wide range of ORM frameworks, namely Hibernate, JDO, TopLink, and JPA. It adds significant support when using these ORM tools to create the data access layer in terms of resource management, DAO implementation, and transaction strategies. For each ORM framework, there is available a template class similar to `JdbcTemplate` (see 5.2.2) to make the use of ORM-specific API less complicated, a translator for converting from proprietary exceptions to the generic data access exception hierarchy (see 5.2.1), an integration with Spring's transaction infrastructure (see 5.3.1), and more. Basically, Spring aims at making it easier to work with various data access technologies in a consistent way. On the other hand, it does not limit developers to use ORM-specific API, such as Hibernate's "criteria" query API, if needed. To compare the support of JPA between both frameworks, a Spring-managed bean utilizing JPA is shown in code snippet 5.7. As can be seen, both EJB and Spring allow for using JPA in the same way.

```
public class StockDao implements IStockDao {

  @PersistenceContext
  private EntityManager entityManager;

  public void addProduct(Product product) {
    entityManager.persist(product);
  }

}
```

Code Snippet 5.7: The use of JPA in Spring

5.2.4 Summary

From what has been discussed we may draw the following conclusions. EJB is tightly integrated with JPA, which has been introduced to replace old-fashioned EJB 2.1 entity beans. By contrast, Spring provides extensive support for building the data access layer of applications, offering several worthwhile facilities aimed at making it easier to work with a variety of data access technologies.

Feature	Spring 2.5	EJB 3.0
Support for JDBC data access	`JdbcTemplate`, `SimpleJdbcTemplate`	none
Supported ORM frameworks	Hibernate, JPA, TopLink, JDO	JPA

Table 5.2: Data access overview

5.3 Transaction Management

A key service for development of robust enterprise applications is transactions. In general, there are two choices for transaction management: *local* and *distributed transactions*.

Local transactions are resource-specific. They are used by applications that work only with a single transactional resource, usually a relational database. The most common example would be a transaction associated with a JDBC connection.

Distributed transactions can span multiple transactional resources, typically various databases and message queues. This transaction strategy requires a *global transaction manager*, which controls transaction processing, using the *two-phase commit protocol* (2PC) for coordinating commits of distributed transactions. To allow the global transaction manager to enlist resources in a distributed transaction, resource managers have to adhere to the *XA specification* [62]. Such resource managers are said to be XA-compliant. In JEE, the contract between the global transaction manager and the other parties involved in a distributed transaction system is specified by the Java Transaction API [61].

This section discusses the approach taken by the examined frameworks and the way they facilitate handling transactions in a programmatic as well as declarative way.

5.3.1 Transaction Infrastructure

The core transaction support in EJB is based on distributed transactions, i.e., it is tied to JTA. Transactions that span a single resource are viewed as a special case of distributed transactions. In such cases EJB containers may use a local transaction optimization, nevertheless, this behaviour is not required by the EJB specification. By contrast, Spring provides an abstraction layer over different transaction strategies, allowing developers to use the most appropriate one for the given requirements. Since application code uses a generic API, it is easy to switch between various transaction strategies with configuration changes only.

Similar to the JDBC API concealing the database driver beneath, JTA specifies several interfaces that the underlying transaction service is supposed to implement. The most important interfaces are:

- **`TransactionManager`**: This interface defines the methods that allow an application server (an EJB container) to manage transaction boundaries.

- `UserTransaction`: This interface defines the methods that allow an application (enterprise beans) to explicitly manage transaction boundaries.
- `XAResource`: A Java mapping of the XA standard interface. Any resource manager participating in a distributed transaction has to support this interface.

The most common JTA implementation provided by application server vendors is the Java Transaction Service (JTS) [63]. JTS is a Java mapping of the OMG Object Transaction Service (OTS) [46]. It uses the standard CORBA ORB/TS interfaces and *Internet Inter-ORB Protocol* (IIOP) for transaction context propagation between multiple JTS transaction managers, located elsewhere on the network. A bean residing in an application server can call a remote bean in a different server with implicit propagation of the transaction context. Figure 5.5 further clarifies the JTA/JTS-based transaction infrastructure.

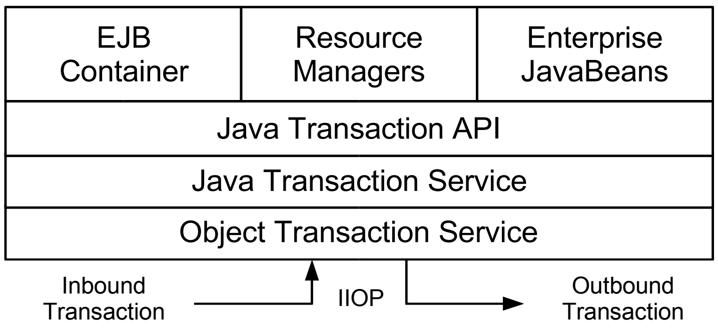

Figure 5.5: JTA/JTS-based transaction infrastructure

Obviously, remote transaction propagation is important to distributed applications. However, since transaction interoperability is not mandated by the EJB specification, vendors may choose not to provide their JTA implementations with this feature.

As mentioned above, Spring's transaction infrastructure is pluggable. The central interfaces that any developer should be familiar with are shown below.

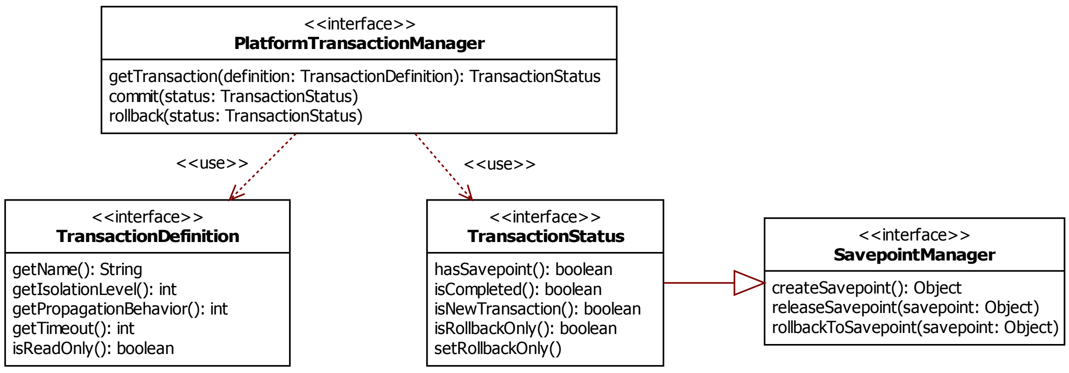
<<interface>>
PlatformTransactionManager
getTransaction(definition: TransactionDefinition): TransactionStatus
commit(status: TransactionStatus)
rollback(status: TransactionStatus)
<<use>>
<<use>>
<<interface>>
TransactionDefinition
getName(): String
getIsolationLevel(): int
getPropagationBehavior(): int
getTimeout(): int
isReadOnly(): boolean
<<interface>>
TransactionStatus
hasSavepoint(): boolean
isCompleted(): boolean
isNewTransaction(): boolean
isRollbackOnly(): boolean
setRollbackOnly()
<<interface>>
SavepointManager
createSavepoint(): Object
releaseSavepoint(savepoint: Object)
rollbackToSavepoint(savepoint: Object)

Figure 5.6: Central interfaces of Spring's transaction infrastructure

The `org.springframework.transaction.TransactionDefinition` interface allows for various ways to specify the characteristics of a transaction. The transaction parameters are:

isolation level (the database transaction isolation to apply), *propagation behaviour* (how to deal with creation of new transactions and propagation of existing transactions), *timeout* (the amount of time after which the transaction should be cancelled), and *read-only status* (a hint whether to optimize for a read-only transaction). Let us note that not all transaction managers support all these settings. For example, the read-only status is useful for ORM frameworks in order not to try to detect and flush changes in a read-only transaction; on the other hand, this level of optimization is ignored by most JDBC drivers.

The key to the Spring transaction abstraction is the notion of a transaction strategy defined by the `org.springframework.transaction.PlatformTransactionManager` interface. This interface abstracts the actual transaction strategy, being able to return a `org.springframework.transaction.TransactionStatus` object for a given `TransactionDefinition` and to trigger the commit or rollback for this status object. Spring offers a `PlatformTransactionManager` for a number of transaction management strategies: transactions managed on a JDBC connection, transactions managed on ORM Units of Work (Hibernate, JPA, etc.), and transactions managed via the JTA `TransactionManager` and `UserTransaction`.

The `TransactionStatus` object is a representation of the current transaction. It holds information about the transaction status and a reference to the underlying transaction object for identifying the transaction on commit or rollback calls. Further, it extends the `org.springframework.transaction.SavepointManager` interface, providing access to *savepoint* management facilities if supported by the transaction manager. By contrast, the EJB architecture supports only *flat transactions*, support for *nested transactions* is optional.

Since transaction failures cannot typically be handled by client code, Spring's transaction infrastructure classes use a generic hierarchy of unchecked transaction exceptions. Moreover, unchecked exceptions enable easy use with AOP interceptors (AOP interceptors cannot throw checked exceptions, as they are executed in a call stack for which they cannot change method signatures by changing the exception contract).

5.3.2 Transaction Demarcation

In a layered architecture, transaction management belongs into the business service layer. Business service objects demarcate transaction boundaries without understanding involved resources while DAOs fetch transactional resources without worrying about participation in transactions. There are two ways how applications can harness transaction services, in a programmatic way, or declaratively, using some kind of metadata. In terms of EJB, this is known as *bean-managed* transaction demarcation (BMT) and *container-managed* transaction demarcation (CMT).

5.3.2.1 Programmatic Transaction Demarcation

With programmatic transaction demarcation, developers initiate, commit, or roll back transactions by-hand, working with a low-level transaction API provided by the frameworks, namely with the `UserTransaction` and `PlatformTransactionManager` interface, respectively. In addition, Spring provides the `org.springframework .transaction.support.TransactionTemplate` class, which adopts the same approach as other template classes, such as the `JdbcTemplate` (see section 5.2.2). The `TransactionTemplate` ensures proper initialization and closing of the transaction, as well as transaction exception handling, allowing for executing transactional code without the need to re-implement transactional workflow; only a callback implementation that performs the actual transactional code is required.

5.3.2.2 Declarative Transaction Demarcation

Although programmatic transaction demarcation is sometimes useful, for example, when more than one transaction is required in a single business method, declarative transaction demarcation is in most situations preferable. In this demarcation strategy, the framework or container is completely responsible for transaction management according to provided configuration.

As we have already discussed in section 3.2.2, the EJB container delivers its services implicitly by method interception. This way it also manages transactions for beans, using the underlying `TransactionManager` implementation. By contrast, Spring's declarative transaction support is enabled via AOP proxies that use `org.springframework .transaction.interceptor.TransactionInterceptor` in conjunction with an appropriate `PlatformTransactionManager` implementation to drive transactions around method invocations. The `TransactionInterceptor` creates transactions for transactional methods before being invoked, and commits or rolls them back when methods complete. For detailed information about Spring's AOP capabilities see sections 3.3.3 and 5.1.

Using declarative transaction management, developers have to tell the container how it should manage transactions for individual methods and propagate transaction context up the thread of execution. For example, when a transactional method is executed, should the container continue running in the existing transaction, or suspend it and create a new one? There are several transaction attribute settings to be used to specify method-level transaction propagation behaviour:

- **REQUIRED**: Support a current transaction, create a new one if none exists.
- **REQUIRES NEW**: Create a new transaction, suspend the current transaction if one exists.
- **SUPPORTS**: Support a current transaction, execute non-transactionally if none exists.
- **MANDATORY**: Support a current transaction, throw an exception if none exists.
- **NEVER**: Execute non-transactionally, throw an exception if a transaction exists.
- **NOT SUPPORTED**: Execute non-transactionally, suspend the current transaction if one exists.
- **NESTED**: Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else.

Except the last option (`NESTED`), which is specific to Spring, all of them are supported by both frameworks and can be configured via annotations or in the deployment descriptor.

The issue with the declarative transaction model is how to control the rollback of transactions in order not to couple application code to the framework's transaction infrastructure. In other words, how to avoid handling the rollback operation for declarative transactions in a programmatic manner.

The recommended way to indicate to Spring that a transaction is to be rolled back is to throw an exception from code within the transaction. These unhandled exceptions will be caught by the `TransactionInterceptor` and result in the transaction being rolled back. By default, the `TransactionInterceptor` only marks a transaction for rollback in the case of runtime, unchecked exceptions. All checked exceptions that should cause automatic rollback have to be specified for a transactional method declaratively. This concept is referred to as *rollback rules*. A snippet of XML configuration that demonstrates this approach is shown below.

```
...
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <!-- configuration for all methods starting with 'get' -->
    <tx:method name="get*"
    rollback-for="spring.exception.NoProductInStockException"/>
  </tx:attributes>
</tx:advice>
...
```

Code Snippet 5.8: Concept of rollback rules in Spring

The EJB specification distinguishes between *system exceptions*, usually unchecked exceptions used internally by the EJB container, and *application exceptions*, usually application-specific checked exceptions. The EJB container automatically rolls back the transaction on a system exception. However, if an application exception is thrown within a transaction, the EJB container does not roll back the transaction. In fact, developers have two choices in order to avoid handling the rollback for application exceptions in a programmatic way:

- To throw the `javax.ejb.EJBException` that wraps the original application exception. Since the `EJBException` is a system exception, the transaction will be rolled back.
- To apply the `@ApplicationException` annotation to the application exception class with the `rollback` parameter set to `true`, or provide the equivalent configuration in the deployment descriptor. This way annotated application exceptions are handled by the EJB container in the same way as system exceptions.

Particularly, the `@ApplicationException` approach tries to mimic Spring's rollback rule processing. Nevertheless, the difference is evident. While rollback rules can be fine-tuned per transactional method, the `@ApplicationException` specifies automatic rollback handling for an application exception per deployment unit.

5.3.3 Transaction Isolation

Choosing the right level of transaction isolation is crucial for robustness and scalability of applications. However, isolation levels are very specific to the behaviour and capabilities of the underlying resources. For example, not all relational databases support all four isolation levels specified by SQL92 (read uncommitted, read committed, repeatable read, serializable). For this reason the EJB specification does not offer component-level isolation setting, and suggest that isolation level should be set at individual resource level, using the respective resource APIs or some other means of configuration [11]. By contrast, Spring's transaction abstraction allows to specify isolation level declaratively per transaction, see the `TransactionDefinition` interface. Nevertheless, this requires support from the underlying `PlatformTransactionManager` implementation, or more precisely from the underlying transaction system.

5.3.4 Summary

Being originally designed as a distributed component architecture, EJB provides extensive support for distributed transactions, such as transaction context propagation, in the first place. By contrast, since Spring is particularly used for building web applications where a single data source is usually involved, it pays more attention to local transactions. As regards to declarative transaction management, it has been considered as one of the greatest benefits of EJB since its first versions. Nevertheless, Spring's transaction infrastructure offers equivalent functionality with additional features like rollback rules and isolation level configuration.

Feature	**Spring 2.5**	**EJB 3.0**
Transaction models	flat transactions, nested transactions	flat transactions
Transaction strategies	local transactions, distributed transactions	distributed transactions
Transaction infrastructure	Spring transaction API	JTA
Facilities to handle transactions in a programmatic way	`TransactionTemplate`	none
Supported propagation attributes	REQUIRED, REQUIRES NEW, SUPPORTS, MANDATORY, NEVER, NOT SUPPORTED, NESTED	REQUIRED, REQUIRES NEW, SUPPORTS, MANDATORY, NEVER, NOT SUPPORTED
Declarative rollback handling	rollback rules	`EJBException`, `@ApplicationException`
Isolation level configuration	supported	not supported

Table 5.3: Transaction management overview

5.4 Remoting

Running in a heterogeneous environment, applications often need to make available some of their functionality to remote clients and other systems. For this purpose, application architecture usually consists of another layer, the *remoting layer* [2] (see figure 5.1). For each business service object being exposed, a corresponding server-side proxy exists in this layer. External clients use similar client-side proxies to connect to their server-side counterparts in order to invoke methods on target business service objects. Both client-side and server-side proxies concern with all protocol-specific and network-related matters, such as data marshalling and unmarshalling, making remote calls, from a client perspective, as transparent as possible. Depending on whether we have both sides of communication under control, we can take advantage of a technology-specific protocol to achieve better performance, or use a web-services-oriented one to offer platform-independent remoting.

5.4.1 Java Remoting

Traditional Java remoting is based on the *Remote Method Invocation* (RMI) standard [30]. RMI builds on Java serialization, with pluggable wire protocols; the default protocol is Java Remote Method Protocol (JRMP). In terms of programming model, all RMI services have to implement a remote interface that extends the `java.rmi.Remote` marker interface. Further, each remote method has to throw the checked `java.rmi.RemoteException` in addition to any application-specific exceptions. Finally, to be available for clients, any exported service has to be registered with an *RMI registry*. Code snippet 5.9 provides an illustration of RMI remote interface.

```
public interface MyRemoteService extends Remote {
  public void doSomething() throws RemoteException;
}
```

Code Snippet 5.9: RMI interface

As we have seen in section 3.2.1, the EJB specification has adopted RMI as a central technology for remoting. A remote session bean has to provide either an old-style *remote interface* (see figure 4.1) or a *remote business interface* (see code snippet 4.1), or both. For each remote bean, the EJB container is responsible for generating all necessary proxy classes, registering the bean with an RMI registry maintained usually by the application server, and exposing it via JNDI environment.

In case that an application is distributed in its nature, EJB's component model is a good choice. Supporting advanced concepts like transaction and security context propagation upon remote method invocation (see 5.3.1, 5.6.1.1) , EJB is well-suited for building such kind of applications. Further, simplifying the contract for remote session beans, the EJB 3.0 specification has made it easier to use enterprise beans in collocated applications as well. In contrast to the EJB 2.1 remote interface which has to be a regular RMI interface, the remote business interface introduced in EJB 3.0 is a plain Java interface, hiding the RMI contract from clients.

Spring offers support for remoting within a lightweight architecture. Its remote access framework is an abstraction for working with various RPC-based technologies available on the Java platform, such as RMI, Spring's HTTP invoker, Caucho's Burlap and Hessian. To allow for exposing business service objects on servers, it provides a mechanism of so-called *exporters*. For example, Spring's RMI exporter is built on sending method invocations through an *RMI invoker* via Java reflection. From the RMI point of view, each exported RMI service is of the invoker type, with a common remote interface and stub/skeleton pair which define a single "invoke" method. In other words, each exported business service object is wrapped with an RMI invoker service. Obviously, transparent remoting via an RMI invoker allows any business service interface to be used as a remote service interface as long as serialization requirements are met, thus, business service objects do not have to be aware of their potential remote exposure at all. Since other exporters are implemented in a similar way, switching from one remoting protocol to another is just a matter of configuration as shown in the following code snippet.

```
...
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
  <!-- RMI service name -->
  <property name="serviceName" value="MyService"/>
  <!-- business service object to be exported via RMI -->
  <property name="service" ref="myService"/>
  <!-- business service interface to be used as remote interface -->
  <property name="serviceInterface" value="spring.service.IMyService"/>
  <!-- RMI registry port -->
  <property name="registryPort" value="1199"/>
</bean>
...
```

Code Snippet 5.10: Exporting business service object using `RmiServiceExporter`

From a client perspective, Spring provides several `FactoryBeans` (see section 4.3.2), for example `org.springframework.remoting.rmi.RmiProxyFactoryBean`, to make accessing remote services less complicated. In general, proxies created by these factories delegate method invocations to the client-side proxies of the technology beneath via Java reflection, keeping clients unaware of this fact, that is, clients have no knowledge whether they are calling local or remote business services. This also includes support for an automatic translation from technology-specific exceptions, such as `java.rmi .RemoteException`, to a generic unchecked `org.springframework.remoting .RemoteAccessException` and its subclasses (compare to Spring's generic data access exception hierarchy discussed in section 5.2.1). In that way, the concept of `FactoryBeans` resembles the *Business Delegate* pattern [25] whose purpose is to shield clients from underlying remoting technology as well.

5.4.2 Web Services

The most common approach to modelling web services is based on the *Web Services Description Language* (WSDL) [37] specified by the W3C consortium. Each WSDL-based web service consist of a service that defines one or more ports. Each port corresponds to a service endpoint on a server, i.e., to a Java service interface; in that respect, the port level is comparable to classic remote service interfaces. Finally, multiple endpoints can be gathered into a single WSDL-defined web service.

The WSDL-based web service programming model has been adopted by JEE in the JAX-WS standard [36], which supersedes the previous JAX-RPC [38], being more flexible in terms of bindings and being heavily annotation-driven. Since Java 1.6, JAX-WS has also been integrated with the JDK's built-in HTTP server.

The EJB-compliant container has to support both specifications, JAX-WS as well as JAX-RPC. For each stateless session bean representing an web service endpoint, the container has to generate the implementation class that handles requests to the endpoint, unmarshalls the request, invokes any business method interceptor methods, and finally invokes the bean method that matches the endpoint method that corresponds to the request. The EJB container is also responsible for generating and publishing WSDL contract for each web service endpoint. As an illustration, the following code snippet shows a stateless session bean exposed via JAX-WS using `@WebService` and `@WebMethod` annotations.

```
@Stateless
@WebService
public class MyWebServiceEndpoint {

  @WebMethod
  public void doSomething() { … };

}
```

Code Snippet 5.11: JAX-WS endpoint in EJB

Spring offers support for JAX-WS and JAX-RPC via its concept of exporters mentioned in the previous section. Speaking of the JAX-WS specification, it allows for using `@WebService` and `@WebMethod` annotations in a similar way as EJB does. Spring's `org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter` detects all `@WebService` annotated beans in the Spring application context and exports them through the default JAX-WS server (the JDK 1.6 HTTP server). A simple Spring JAX-WS endpoint is presented in code snippet 5.12. As can be seen, there is no difference to the EJB example.

```
@WebService
public class MyWebServiceEndpoint {

  @WebMethod
  public void doSomething() { ... };

}
```

Code Snippet 5.12: JAX-WS endpoint in Spring

Analogous to `RmiProxyFactoryBean`, `for each web service specification`, Spring provides two types of factory beans for creating client-side proxies; one type returns an entire JAX-WS/JAX-RPC service to work with, the other returns only a proxy that implements a particular endpoint interface the client is interested in.

In addition to support for JAX-RPC and JAX-WS, the Spring portfolio also features the Spring Web Services project [43] that focuses on creating contract-first, document-oriented web services. A detailed description of Spring Web Services is beyond the scope of this thesis, more information can be found at the project's homepage.

In the end, let us mention that upcoming Spring 3.0 and EJB 3.1 are likely to support another style of modelling web services, *RESTful web services* [39]. While EJB 3.1 seems to adopt the JAX-RS specification [40] Spring 3.0 is taking a different approach in order to tightly integrate REST functionality to its presentation framework, Spring MVC.

5.4.3 Summary

To conclude, both frameworks provide a great support for exporting business services as via Java-based remoting as via web services. The most notable benefit of Spring's remote access framework is the ability to expose any business service object over a variety of protocols just by means of configuration. By contrast, the strength of EJB lies in its support for distributed computing.

Feature	Spring 2.5	EJB 3.0
Supported remoting technologies	RMI, Spring's HTTP invoker, Hessian, Burlap	RMI
Support for web services	JAX-WS, JAX-RPC, Spring Web Services	JAX-WS, JAX-RPC
Distributed objects	not supported	supported

Table 5.4: Remoting and web services overview

5.5 Messaging

An alternative to remote method invocation is *messaging*, which allows asynchronous programming via sending and receiving messages. Message-oriented middleware (MOM), such as Apache ActiveMQ, BEA Tuxedo, IBM WebSphere MQ, and Sun Java Messaging Server, sits between *message producers* and *message consumers* and allows them to communicate in a non-blocking way without knowing each other. There are two basic messaging styles or *domains*: *publish/subscribe* and *point-to-point*.

A publish/subscribe messaging is based on message broadcasting. Subscribers register their interest in a particular event *topic*. Publishers create messages (events) that are distributed to all the subscribers.

In the case of a point-to-point messaging, message producers communicate directly to message consumers, or send messages to a centralized *queue*. Multiple consumers can grab messages of the queue, but any given message is consumed only once; for each message, there is a single consumer. From this point of view, point-to-point is a special case of publish/subscribe.

An issue with MOM products is that they have their own API, thus, switching between different MOMs usually means to learn another proprietary API. The *Java Message Service* (JMS) API [47] solves this problem. It has two parts: a client API to write code to send and receive messages, and a *Service Provider Interface* (SPI) to plug in various JMS providers that know how to communicate to a specific MOM implementation. Before delving in details of EJB and Spring's support for messaging, let us briefly describe the client API of JMS.

5.5.1 Introduction to the Java Message Service API

The JMS API involves many interfaces which it is necessary to get familiar with. The flow of steps a client has to undertake to send/receive a message is illustrated in figure 5.7, a detailed description of the process follows.

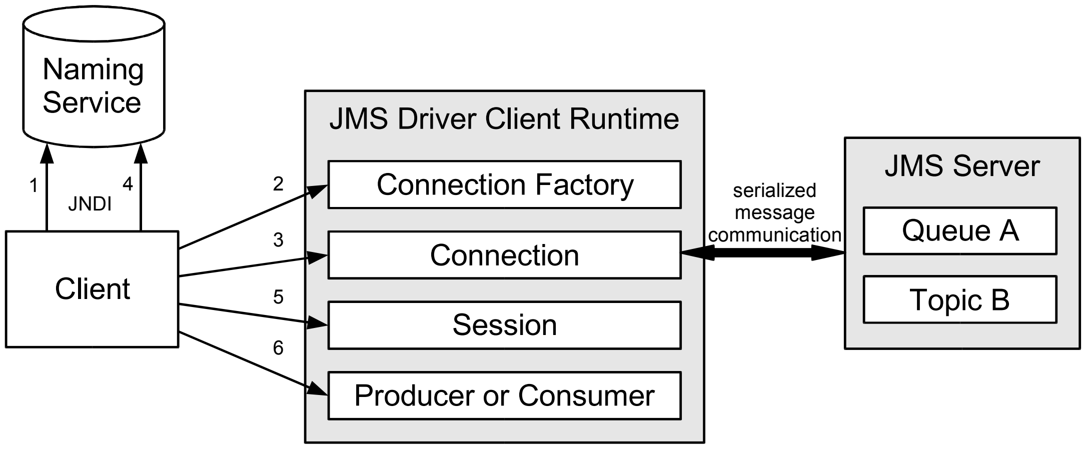

Figure 5.7: Client view of a JMS system

First, a client needs to locate the JMS provider's `javax.jms.ConnectionFactory` instance. It usually looks it up via JNDI (1). Using the `ConnectionFactory`, the client establishes a `javax.jms.Connection` to the JMS provider (2). Then, it creates a `javax.jms.Session` object that allows for encapsulating messages in transactions and setting the way of message acknowledgement (3). Next, it obtains a `javax.jms.Destination`, that is, the target `javax.jms.Queue` or `javax.jms.Topic`, again via JNDI (4). Using the `Session` and `Destination` objects, it creates a `javax.jms.MessageProducer` or `javax.jms.MessageConsumer` object, depending on whether it is interested in sending or receiving messages (5). Finally, it encapsulates business data into a `javax.jms.Message` and sends it via `MessageProducer`, or processes messages received via `MessageConsumer` (6).

As regards receiving messages, there are two possible ways. A client can synchronously receive messages invoking one of `MessageConsusmer's send(..)` methods, or register a `javax.jms.MessageListener` implementation via `MessageConsumer's setMessageListener(..)` method to be automatically notified when a message is delivered. Further in the text we concentrate on the latter approach, i.e., asynchronous delivery, because this is usually the preferred way.

5.5.2 Asynchronous Message Consumption

An application heavily utilizing messaging usually manages a pool of message listeners to process messages delivered from clients in an asynchronous way. To develop such a piece of code from scratch is challenging; it includes writing a multi-threading logic for registering objects as message listeners, starting and stopping them as the amount of messages to be consumed is changing, and more. The EJB specification makes this task easier. It defines a special type of component that can receive messages, a *message-driven bean* (MDB).

Holding no conversational state and having no client-visible identity, MDBs are in their nature stateless. To be registered by the EJB container as a JMS listener, an MDB has to implement the `MessageListener` interface. Further, it has to define a destination to be associated with, and can also specify other operational requirements, such as message acknowledgement mode, via the required `@MessageDriven` annotation.

Like stateless session beans, MDBs are pooled by the container to allow streams of messages to be processed concurrently; an MDB is invoked by the container upon arrival of a message at the destination that is serviced by the bean. Since MDBs are single-threaded, there is no need for synchronization code in the bean class; the container is responsible for serializing messages to a single MDB so that it processes only one message at a time. An example of an MDB is shown in code snippet 5.13.

```
@MessageDriven(mappedName = ”jms/LogQueue”)
public class Logger implements MessageListener {

  // method prescribed by MessageListener interface
  public void onMessage(Message message) {
    if (message instanceof TextMessage) {
      // process text message
    } else {
      // process messages of other types
    }

  }

}
```

Code Snippet 5.13: Message-driven bean

To address asynchronous messaging, Spring provides a concept of *message-driven POJO* (MDP) that acts as a receiver for JMS messages. An MDP is a plain Java object which can optionally implement the `MessageListener` interface, or, in contrast to an MDB, just declare an arbitrary method for handling messages. In the latter case, the MDP is wrapped by an `org.springframework.jms.listener.adapter.MessageListenerAdapter` that delegates the handling of messages to target listener methods via reflection. In case an MDP is receiving messages on multiple threads, it is also necessary to ensure that the implementation is thread-safe.

An important part of Spring's messaging infrastructure is a *message listener container* that plays a key role in management of MDPs. It is responsible for all threading of message reception and dispatching into MDPs for processing. A message listener container is an intermediary between an MDP and a messaging provider; it takes care of registering to receive messages, participating in transactions, acquiring and releasing resurces, and similar. The following code snippet illustrates an MDP which benefits from the `MessageListenerAdapter` approach and the configuration of the message listener container used to manage its instances.

```
// message-driven POJO
public class Logger {

  public void log(TextMessage textMessage) {
    // process text message
  }

}

...
<!-- message-driven POJO configuration -->
<bean id="logger" class="spring.services.Logger"/>
<!-- message listener container configuration -->
<jms:listener-container>
  <jms:listener destination="jms/LogQueue" ref="logger" method="log"/>
</jms:listener-container>
...
```

Code Snippet 5.14: Message-driven POJO

5.5.2.1 Handling Message Payload

An issue with the `MessageListener` interface is that the `onMessage(..)` method takes as its parameter a `Message` object. As there are a lot of subclasses of the `Message` class, such as `javax.jms.TextMessage` or `javax.jms.BytesMessage`, one has no choice but to use the `instanceof` operator to determine whether a received message is of the desired type. We could have seen this approach in code snippet 5.13 presenting an MDB.

Spring addresses this issue and allows for strongly typed message consumer methods. It introduces so-called *message converters*. Central in this mechanism is the `org.springframework.jms.support.converter.MessageConverter` interface that specifies a converter between Java objects and JMS messages. The default implementation supports conversion between `String` and `TextMessage`, `byte[]` and `BytesMesssage`, and `java.util.Map` and `MapMessage`. Utilizing an appropriate `MessageConverter` implementation, the `MessageListenerAdapter` provides automatic conversion from `Message objects` to objects of target type, and delegates to MDPs to handle them. Obviously, using custom `MessageConverters`, application code can focus on business objects being sent or received via JMS, rather than on the details of how they are represented as JMS messages.

5.5.2.2 Returning Results Back to Message Producers

In some situations it may be required to send a response back to the producer that originally generated the message. The authors of the Mastering Enterprise JavaBeans book [9] propose two possible solutions to this problem:

- The client creates a *temporary queue* that all response messages are send to.
- A *permanent topic* is configured. All response messages are delivered to this topic for all clients, which filter out the messages that belong to them.

In either case a message producer specifies the destination to send the response message to in the JMS message header field called `JMSReplyTo` of the original message. Extracting the value of this field, a message consumer is able to return some data back to the message producer as shown in code snippet below.

```
...
MessageProducer producer =
  session.createProducer(requestMsg.getJMSReplyTo());
TextMessage responseMsg = session.createTextMessage("Result: success");
// reference the original message
responseMsg.setJMSCorrelationID(requestMsg.getJMSMessageID());
producer.send(responseMsg);
...
```

Code Snippet 5.15: Using `JMSReplyTo` field to return results back to message producer

The EJB specification does not offer any mechanism that allows an MDB to propagate a response back to the message producer. It is necessary to use the approach described above. By contrast, if an MDP's method returns a non-null object (typically of a message content type such as `String` or byte array), the `MessageListenerAdapter` will wrap it in the JMS message and automatically replies to the destination specified by the incoming message's `JMSReplyTo` field.

5.5.3 Summary

Both message-driven bean and message-driven POJO concept are very helpful for creating robust messaging solutions. Providing the underlying infrastructure, both Spring and EJB allow developers to focus primarily on business logic of message consumers without spending a lot of time on low-level messaging issues. Nevertheless, comparing the concepts to each other, we can conclude that Spring's one is more advanced. It allows for using any managed object as a message consumer without forcing it to implement a message listener interface. In addition, Spring offers some useful features, such as payload conversion and auto-reply, which EJB does not support.

Feature	Spring 2.5	EJB 3.0
Asynchronous message consumers	message-driven POJOs	message-driven beans
Type of asynchronous message consumer	any object	JMS listener
Payload conversion	supported	not supported
Automatic auto-reply	supported	not supported

Table 5.5: Messaging overview

5.6 Security

Integrating diverse business resources, enterprise applications usually handle huge amounts of critical data which needs to be protected in some way. Thus, security is definitely one of the most important aspects of any application architecture. Security comprises two major operations. The first is known as *authorization*, the process of verifying that a user, device or other system, generally called a *principal*, is who they claim to be. When the identity of a principal has been established, the second process of deciding whether the principal is allowed to perform an action in the application, referred to as *authorization*, comes into play. In this section we analyse how the examined frameworks approach these basic security concepts.

5.6.1 Authentication

A common way to call authentication logic in Java is through the *Java Authentication and Authorization Service* (JAAS) [48] an integral part of Java since the JDK 1.4. JAAS is a set of portable interfaces which allow a user to log into a system without knowing about the underlying security system being used. Behind the scene, the JAAS implementation, such as an EJB container, or more precisely an application server, determines if the user's credentials are authentic. The power of JAAS lies in its ability to use almost any underlying system, just by providing a custom authentication module. A basic overview of the JAAS authentication procedure is shown in the following figure.

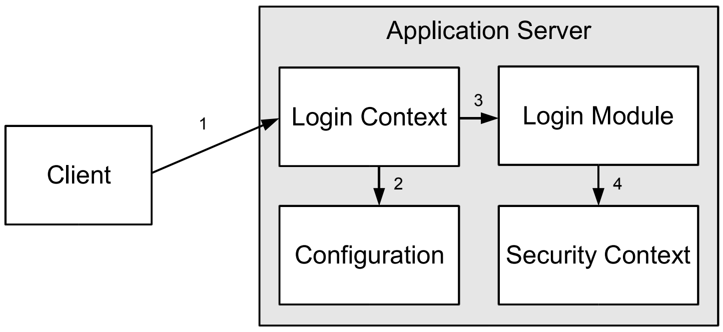

Figure 5.8: JAAS authentication procedure

Suppose a client is calling a bean in an application server. Fist of all, the client instantiates a new `javax.security.auth.login.LoginContext` (1), a class responsible for coordinating the authentication process provided by the application server. Next, the login context asks a configuration object for the authentication mechanism to use (2), such as password-based or certificate-based, and instantiates the corresponding `javax .security.auth.spi.LoginModule` (3). The login module authenticates the client using a proprietary means and stores authentication information in the user's security context (4). Once the client has been authenticated, the user can call EJB methods securely; the user's security context is propagated upon method invocations by the EJB container, being available for the programmatic use via the container-provided runtime EJB context, `javax.ejb.EJBContext`.

Spring's authentication features, as all its security features, are provided in a separate subproject, *Spring Security* [49]. The aim of this subproject is to offer a security framework with support for wide range of authentication and authorization models, which is easily customizable and extensible. Taking into account the amount of work which usually needs to be done when switching between different server-specific security implementations, Spring Security introduces its own abstract security architecture which allows for portability between various Java environments. Considering figure 5.9, let us discuss a basic authentication process and describe the main security components being used.

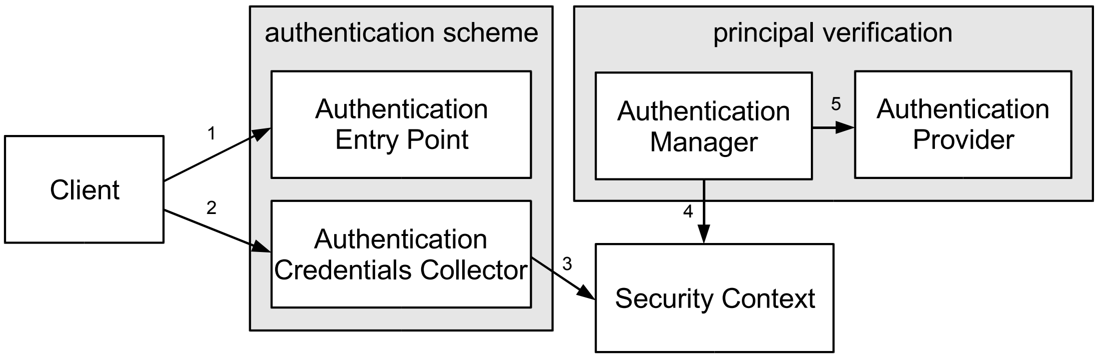

Figure 5.9: Spring Security authentication procedure

When an unauthenticated client asks for a protected resource, the authentication procedure is started by an `org.springframework.security.ui.AuthenticationEntryPoint` indicating the client that the user must authenticate (1), for example, it redirects the client to the login page. After the user provides its identity and sends the data back to the server, an authentication-scheme-specific object (for simplicity, termed as an authentication credentials collector in the figure) collects received authentication details into an `org.springframework.security.Authentication` object (2) and places it into the security context (3). Afterwards an `org.springframework.security.Authentication Manager` retrieves the `Authentication` object (4) and delegates user identity verification to an `org.springframework.security.providers.AuthenticationProvider` which decides whether or not the identity is valid (5), for example by consulting to an LDAP server.

Allowing to implement any standardized or custom authentication mechanism, both JAAS and Spring authentication architecture are very powerful. Spring Security supports authentication integration with a lot of technologies, such as form-based authentication, HTTP X.509 client certificate exchange, LDAP, and JAAS, out of the box. Speaking of authentication mechanisms available in an EJB environment, it often varies depending on the security implementation provided by a particular application server. In this sense Spring Security framework is more superior, overcoming the problems associated with security reconfiguration when switching between different server environments.

5.6.1.1 Security Propagation

Once a client is authenticated, all security checks are made against the security context propagated upon method invocations. In collocated applications security context propagation is quite straightforward; security contexts are usually stored as thread-local variables. In the case of distributed applications the situation is a little bit complicated. Regardless of remoting technology, it is highly desirable to enable security context propagation upon remote method invocations transparently at the protocol layer, without affecting existing application code. This is typically achieved via an interception mechanism [50], piggybacking security information on each invocation message between the client and the server. In addition, sometimes business services need to call remote objects which require different authentication and authorization credentials. This mechanism of temporary replacement of a user identity is known as a *run-as authentication replacement* [49].

To achieve interoperability between EJB containers from different vendors, the EJB specification requires *RMI-IIOP* support. Since the IIOP protocol is a part of CORBA portfolio, EJB container vendors often leverage functionality provided by CORBA's *object request broker* (ORB) runtime. Generally speaking, there are four interception points available through the ORB runtime allowing to add transparently any value-added functions via CORBA *invocation interceptors*. Among the many uses of interceptors, propagating security-context data is one of the most important, being supported at the IIOP level. Besides IIOP, the EJB specification leverages a *CSIv2* protocol for additional authentication capabilities, such as *identity assertions* as the means of principal propagation. An identity assertion is sent by the calling client to tell the receiver that it should not consider the client identity (which was established on the transport layer or by the authentication process) for making authorization decisions, but the asserted identity instead. More specific details about EJB and CORBA interoperability can be found in [51].

Although Spring lightweight remoting architecture is not a full-fledged CORBA-like stack for building distributed applications, it also allows for security propagation. As described in section 5.4.1, Spring's RMI support is based on *RMI invokers*, a concept similar to CORBA's *dynamic interface invocation* (DII) to a certain degree. Each remote invocation handled by an RMI invoker is encapsulated into an `org.springframework.remoting.support.RemoteInvocation` object which provides core method invocation properties, such as method name and method arguments, in a serializable fashion. In case that security context propagation is required, it is possible to switch to `org.springframework.security.context.rmi.ContextPropagatingRemoteInvocation` objects which additionally supply authentication and authorization information. Regarding run-as authentication replacement, this functionality is implemented via an `org.springframework.security.RunAsManager` which can replace the current `Authentication` object with another one that applies to the secure object invocation. We'll see how the `RunAsManager` component fits into Spring's security architecture in the next section.

To conclude, both EJB and Spring provide some means to support security propagation. Nevertheless, while Spring's capabilities may be sufficient for Spring-based applications, if complex secure interoperability requirements come into play, EJB's CORBA-based approach proves to be more adequate.

5.6.2 Authorization

Any object that can have security applied to it is termed as a *secure object*. Within any larger application there is usually defined a set of such objects to be protected according to application-specific security policies. On each secure object call the principal is tested whether or not it has the authority to invoke the object. All granted authorities to the principal are usually retrieved during the authentication process and stored within the security context. In general, there are two ways to perform authorization:

- **Programmatic authorization**: Security checks are hard-coded into application code.

- **Declarative authorization**: Authorization is delegated to the framework which performs all security checks behind the scene according to security policies defined declaratively.

Depending on the type of secure objects we can distinguish between business service object security and domain object security [9], [49]. Regarding business service objects, security checks are typically performed at object or method level. By contrast, domain object authorization is usually condition-based, taking into account properties of each object instance; for example, we might want to permit users to perform operations on a bank account according to its balance. Further in this section we discuss these types of authorization in more detail.

5.6.2.1 Business Service Object Security

The authorization mechanism defined by the EJB specification is based on security roles. For a client to be authorized to execute a method on a bean, its security identity must be in the correct security role for that method. Security checks can be specified either via JSR 250 security annotations or via the equivalent deployment descriptor. Figure 5.16 shows the former approach.

```
@Stateless
@DeclareRoles({“manager”,”employee”})
public class EmployeeManagementService {

  // only managers can perform this operation
  @RolesAllowed({“manager”})
  public void newEmployee(..) { … };

}
```

Code Snippet 5.16: EJB declarative authorization via JSR 250 annotations

First, it is necessary to define roles for security checking by using the `@DeclareRoles` annotation. Next, we need to declare permissions at bean or method level via the `@RolesAllowed` annotation. Finally, the deployer is responsible for mapping principals to the roles we have declared, using container-specific APIs and tools. Once defined, the container automatically performs the security checks for beans at runtime, intercepting method invocations.

In contrast to EJB, Spring does not lock in a particular authorization mechanism. It provides a general `org.springframework.security.GrantedAuthority` interface which represents an authority granted to an `Authentication` object. Hence, it allows for custom authorization mechanisms being based on various `GrantedAuthority` implementations. Providing an authority implementation for security roles, Spring supports the role-based authorization mechanism out of the box.

To add security to service layer methods, Spring supports JSR 250 annotations similar to EJB. Alternatively, its native `@Secured` annotation can be used. Moreover, Spring allows to apply security to multiple beans across the entire service layer via the AspectJ style pointcuts discussed in section 5.1.2; this is particularly powerful. For illustration, code snippet 5.17 depicts the configuration to protect all methods on beans whose classes are in the `spring.service` package and whose class names end in “Service”. Only users with the ROLE_EMPLOYEE role are allowed to invoke these methods.

```
<global-method-security>
  <protect-pointcut
  expression="execution(* spring.service.*Service.*(..))"
  access="ROLE_EMPLOYEE"/>
</global-method-security>
```

Code Snippet 5.17: Spring Security declarative authorization via AspectJ pointcuts

Within the Spring framework method-level security is enforced through its AOP capabilities, to be more specific via an around advice, which can elect whether or not to proceed with a method invocation. The basic class that implements security interception for secure objects is called an `org.springframework.security.intercept .AbstractSecurityInterceptor`; this is a superclass of all security interceptor implementations. The `AbstractSecurityInterceptor` defines the proper handling of secure object invocations which involves cooperation with several other components shown in the following figure.

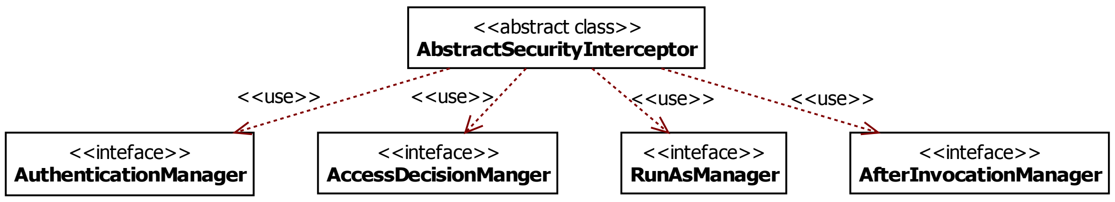

Figure 5.10: Spring Security authorization architecture

First, the `AbstractSecurityInterceptor` obtains the `Authentication` object and checks whether the principal has been authenticated, if necessary it calls the configured `AuthenticationManager` to authenticate it. Next, it authorizes the method invocation against an `org.springframework.security.AccessDecisionManager` which makes the final access control decision according to the principal's granted authorities. Then, any run-as replacement can be performed via the configured `RunAsManager`. Further, the target method on the secured object is invoked. Finally, after returning back to the interceptor, if an `org.springframework.security.AfterInvocationManager` which can replace the object to be returned to the caller is defined, it is processed. Apparently, such an authorization architecture is powerful, extensible and customizable, being more flexible than the EJB authorization mechanism.

5.6.2.2 Domain Object Security

As already mentioned, complex applications might need to define access permissions not simply at a method invocation level; when enforcing security, they might also need to consider the actual domain object instance subject of a method invocation. Generally speaking, such authorization decisions can be made:

- before a method is invoked to ensure a given principal is permitted to work with the objects to be passed as method arguments

- after a method is invoked to ensure a given principal is permitted to work with the object or set of objects returned by the method

Unfortunately, the EJB specification does not provide support for performing this kind of authorization, i.e., instance-level authorization or condition-based authorization. To achieve such a level of security, one has no choice but to implement its own solution and integrate it into the EJB environment. By contrast, as we have seen in the previous section, Spring's security architecture allows for performing security checks as before as after a method is invoked. One only needs to supply an appropriate `AccessDecisionManager` implementation and `AfterInvocationManager` implementation, respectively. Building on its flexible core architecture, the Spring Security subproject delivers an implementation of the well-known ACL-based security model out of the box.

5.6.3 Summary

EJB and Spring approach authentication in very different ways. In the EJB environment, authentication is achieved via a JAAS implementation provided by the application server vendor. By contrast, Spring draws on its proprietary, but customizable and extensible authentication mechanism, offering several implementations of widely-used authentication technologies out of the box. Both frameworks also allow for run-as authentication replacement and security context propagation for distributed applications.

Regarding authorization, as Spring as EJB support the role-based concept to declaratively secure business service method invocations. Nevertheless, allowing for authorization mechanisms based on custom authorities, Spring goes far beyond the role-based authorization model. Additionally, in contrast to EJB, Spring offers a solution for domain object authorization based on access control lists or a custom-made mechanism.

Feature	Spring 2.5	EJB 3.0
Authentication mechanism	proprietary, portable authentication mechanism	JAAS, application-server-specific
Run-as authentication replacement	supported, Spring-specific mechanism	supported, CORBA-based mechanism
RMI security context propagation	supported, Spring-specific mechanism	supported, CORBA-based mechanism
Business service authorization mechanism	role-based, custom mechanism	role-based
Domain object authorization mechanism	ACL-based, custom mechanism	not supported

Table 5.6: Security overview

5.7 Scheduling

Scheduling is another functionality required in many business applications. Various scenarios involving scheduling ensure that certain code, usually called *job*, is executed at a given point in time. For example, a system handling huge loads during the peak hours wants to run some maintenance jobs, such as cleaning the file system of temporary files or generating audit reports, during off hours. A workflow process comprising a set of activities which need to be scheduled to run at a specific time or when a conditional criteria is met is another example involving scheduling. Obviously, sometimes we know the exact time to run a job, as in the former example, on the other hand, there are situations when job scheduling depends on runtime information, as in the latter example. Hence, a scheduling service should allow scheduling jobs as in a programmatic way as in a declarative way.

5.7.1 Scheduling Service

In the EJB environment, scheduling is supported through the container-managed *Timer Service*. A central part of the Timer Service is the `javax.ejb.TimerService` interface whose container-specific implementation is available at runtime via the `EJBContext` object; an alternative way to get access to the Timer Service is to look up the `TimerService` object in the JNDI.

The Timer Service allows scheduling a given *timeout method* defined by a bean to run at a particular point in time so that the EJB container can call back once the scheduled time has elapsed. The bean can declare a method to be the timeout method via the `@Timeout` annotation or in the deployment descriptor; alternatively, it can directly implement the `javax.ejb.TimedObject` interface which defines a single method `ejbTimeout(..)`. Further, the bean must create a `javax.ejb.Timer` object by calling one of `createTimer(..)` methods provided by the `TimerSevice` interface. This way the bean registers itself to the Timer Service to be notified via the timeout callback method upon timer expiration. The following code snippet illustrates the session bean whose cleaning method needs to be executed every day at midnight.

```
@Stateless
public class MaintenanceService {

  @Resource
  private SessionContext ctx;

  // call to schedule the clean(..) method execution
  public void schedule() {
    int start = … // calculate time left till midnight
    int period = 24 * 60 * 60 * 1000; // every day
    // run  maintenance tasks every day at midnight
    ctx.getTimerService().createTimer(start, period, null);
  }

  @Timeout
  public void clean(Timer timer) {
    // put here the code to perform a maintenance task
  }

}
```

Code Snippet 5.18: Programmatic scheduling support in EJB

Although the use of the Timer Service might seem to be straightforward, there are some disadvantages. When creating timers, the only unit of time that the Timer Service API accepts is milliseconds, there is no way to define time intervals in terms of hours, days, or moths. Also, it is not possible to create timers that expire on given days of the week but not on other days, for example a timer that expires on every working day at midnight. Besides these drawbacks the Timer Service lacks any support for declaration of timer intervals in the deployment descriptor, in other words, it does not allow for providing time-related information declaratively at deployment time. All these limiting factors have been eliminated in the EJB 3.1 specification which has been enhanced of the ability to declaratively create cron-like schedules to trigger EJB methods.

Spring provides neither its own scheduler nor an abstract scheduling service API similar to the EJB Timer Service. Instead, it integrates with existing schedulers, supporting the `java.util.Timer`, part of the JDK since 1.3, and the *Quartz* scheduler [52]. Therefore, features provided highly depend on the underlying scheduler being used. For both technologies Spring supplies convenience classes, allowing to schedule a method invocation on an existing target object. For simplicity, only Quartz integration is discussed further in the text; integration with the JDK Timer API is achieved in a very similar manner.

The simplest way to schedule the execution of an existing service method on a target bean is via the `org.springframework.scheduling.quartz.MethodInvokingJobDetail FactoryBean` that exposes an `org.quartz.JobDetail` object which delegates job execution to the specified method. Further, it is necessary to associate the `JobDetail` object with an `org.quartz.Trigger` which is used to trigger the execution of the job at a specified point in time. There are two types of triggers available: `org.springframework.scheduling.quartz.SimpleTrigerBean`, the most simple trigger type, and `org.springframework.scheduling.quartz.CronTrigerBean`, the trigger with cron-style scheduling support. Finally, the trigger needs to be registered to an `org.springframework.scheduling.quartz.SchedulerFactoryBean`. This kind of `FactoryBean` creates and configures a Quartz scheduler and manages its lifecycle within the Spring framework. The next code snippet shows the configuration for the `spring.service.MaintenanceService` bean whose `clean(..)` method needs to be executed every working day at midnight

```
<!-- maintenance service configuration -->
<bean id="maintenanceService"
class="spring.service.MaintenanceService"/>

<!-- maintenance job configuration -->
<bean id="maintenanceJob"
class="org.springframework.scheduling.quartz
.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="maintenanceService"/>
  <property name="targetMethod" value="clean"/>
</bean>

<!-- maintenance job trigger configuration -->
<bean id="maintenanceJobTrigger"
class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail" ref="maintenanceJob"/>
  <!-- run every working day at midnight -->
  <property name="cronExpression" value="0 0 0 ? * MON-FRI *"/>
</bean>

<!-- scheduler configuration -->
<bean
class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <!-- schedule maintenance job -->
      <ref bean="maintenanceJobTrigger"/>
    </list>
  </property>
</bean>
```

Code Snippet 5.19: Declarative Quartz scheduling support in Spring

5.7.2 Summary

We have discussed the EJB Timer Service and Spring's convenience classes for integration with third party schedulers, in particular with Quartz. As we have seen, the EJB Timer Service is pretty simple to use in the programmatic way, on the other hand, it lacks the ability to specify time information declaratively in a more powerful, cron-style manner. This drawback is avoided in the EJB 3.1 specification. Speaking of Spring, the scheduling facilities available completely depend on the capabilities of the scheduler being used. For scheduling simple jobs the JDK `Timer` can be sufficient; in more complex situations the features provided by the Quartz scheduler may be needed.

Feature	Spring 2.5	EJB 3.0
Task scheduler service	Quartz scheduler, JDK `Timer`	EJB Timer Service
Programmatic task scheduling	via scheduler-specific API	via the Timer Service API
Declarative task scheduling	supported	not supported
Cron-style scheduling	supported by Quartz scheduler	not supported

Table 5.7: Scheduling overview

6 Application Development

In the previous two chapters we have covered main features and capabilities of the frameworks relevant to business object management and provision of middleware services. In this chapter we change the subject and focus on more practical issues related to several areas of application development efforts. We discuss characteristic architectures of Spring-based and EJB-based applications and related clustering topics. Further, we examine how the frameworks facilitate testing, particularly unit and integration testing. In the end, we familiarize ourselves with their supporting infrastructure for application configuration.

6.1 Architecture

When developing an enterprise application, the most important design decisions are those about its architecture. There are many definitions of what architecture is. According to [7], architecture can be viewed as "the highest-level breakdown of a system into its parts" and "decisions that are hard to change", i.e., architecture describes the major components of the application and how they interact.

As architecture immensely affects most of the characteristics of the application, such as those discussed in section 1.1, architectural decisions should be made very carefully. In fact, the architecture of an application is influenced to some extent by the underlying infrastructure, particularly by the application framework being used. In this section we focus on characteristic architecture of Spring-based and EJB-based applications. Further, as enterprise applications often run in a cluster of servers, we also discuss related scalability issues.

6.1.1 JEE Architectures

Basically, we distinguish two types of application architectures [7]: *collocated architecture* whose tiers are separated logically in a single server and *distributed architecture* whose tiers are scattered physically on dedicated servers. In other words, a collocated application runs in a single JVM instance while a distributed application spans several JVM instances. Speaking of JEE-based web applications, an application is considered to be collocated or distributed according to the separation of its presentation and business service layer. The basic types of JEE architectures are depicted in the following figure.

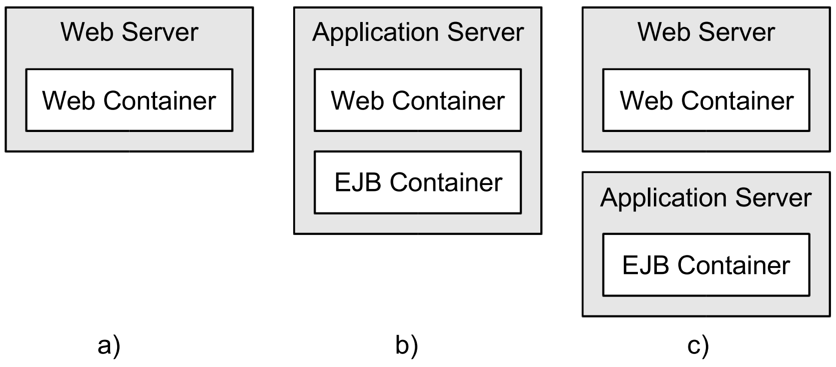

Figure 6.1: Collocated and distributed JEE architectures

JEE applications with the most simple structure are web applications packaged as web modules. That is, all application business objects are modelled as regular Java classes, which are assembled together with web components into a single WAR. These applications can be deployed into any JEE-compliant web server, or more precisely into any JEE-compliant web container. This kind of collocated architecture is particularly advocated by Spring, which allows for packaging business logic into ordinary Java archives. This architecture is shown in figure 6.1, part a).

The second type of JEE architecture illustrated in figure 6.1, part b), is characteristic of EJB-based collocated web applications. Similar to the previous scenario, web components of such applications are packaged as a WAR, but business services are implemented as EJBs being assembled into one or more EJB modules. Thus, an application server containing both the web and EJB container is necessary to deploy these applications into. In addition, the separation of presentation and business logic into different containers makes the interaction between web and EJB components a little bit complicated in terms of class loading [1].

As Java class loaders are hierarchical, when one class depends on other classes, they have to be loaded by the same class loader or any of its parent class loaders; otherwise `java.lang.ClassNotFoundException` is encountered. This may happen when a class loaded by the EJB class loader attempts to load classes loaded by the WAR class loader because the EJB class loader is often the parent of the WAR class loader, and thus it cannot see classes in the WAR. On the other hand, this approach is in compliance with the concept of structuring applications in layers; each layer should depend only on the layer beneath.

Finally, when deploying web and EJB components into physically separated servers, we obtain an EJB-based distributed web application, see figure 6.1, part c). The way clients, in this case web components, can access distributed EJBs has already been discussed in more detail in sections 3.2.1 and 5.3.1. As distributed computing always introduces a higher level of complexity and performance overhead, it is recommended [1], [7] to use this type of architecture as less as possible; there should be a good reason for choosing distributed architecture, for example, there may be an extreme disproportion between the load of web and EJB tier, or web and EJB components may be in different security levels and need to be separated physically.

6.1.2 Clustering

A highly desirable feature of any enterprise application is the ability to scale-up and serve large number of clients, given appropriate hardware. A common way to support increased load is *clustering*, often referred to as *horizontal scalability*. It allows a group of servers to share the heavy tasks and operate logically as a single server. There are two key technologies at the heart of clustering: *load balancing*, the way to obtain high availability and better performance by dispatching incoming requests to different servers, and *failover*, the process to achieve fault tolerance by choosing another server in a cluster when the original server fails, including *state replication* between servers to ensure that all client requests can be processed properly without losing any session state.

Speaking of web tier, no matter whether it is part of a Spring-based or EJB-based application, web load balancing is usually approached by using a *load balancer*, which intervenes between browsers and web servers. A load balancer is typically a hardware product dispatching HTTP requests to the back-end server instances according to a specific load balancing algorithm such as round-robin, random, and weight-based [57]. In addition, it often performs some other important tasks such as *session stickiness* to have a user session live entirely on one server and *health checking* to prevent dispatching requests to a failing server.

In the case of distributed architecture, EJB tier load balancing can happen on several places during the remote method invocation process, which has been described in section 3.2.1. Due to the lack of support from the EJB specification, load balancing implementations vary between EJB container vendors, being based on one of these three mechanisms [58]:

- **Smart stub:** Load balancing and failover logic is put in the stub code running on the client side. The smart stub contains the list of servers it can access, being able to detect any failure about the target server instance.

- **IIOP runtime library modification:** All the logics and algorithms to perform load balancing and failover are implemented in the IIOP runtime library, typically in a modified version of `ORBSocketFactory`.

- **Interceptor proxy:** In this approach, the client-side stub contains routing information to an interceptor proxy rather than to the application server which hosts EJBs. The proxy receives all incoming requests and determines the target server instance to send them to, according to the load balancing and failover policy.

All these techniques are illustrated in figure 6.2.

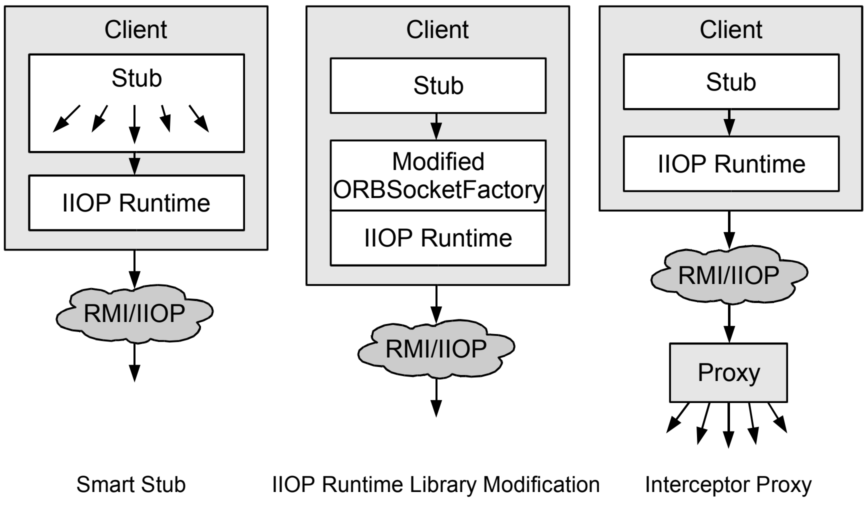

Figure 6.2: EJB load balancing and failover techniques

Concerning state replication, we already know that Spring and EJB deal with state management in different ways (see 4.2). While Spring prefers stateless business services storing client state into the underlying context, mostly into the HTTP session object, the EJB specification provides first-level support for creating both stateless and stateful business services. Thus, regarding Spring, the state replication issue relates typically to HTTP session clustering; in the case of EJB, we speak about stateful session bean clustering.

A well-established solution to HTTP session clustering for Spring-based web applications is the Terracotta platform [60], an infrastructure software for JVM-level clustering. Managing mission critical data using its Network-Attached Memory (NAM) technology, Terracotta extends the Java memory model of a single JVM to include a cluster of virtual machines such that threads on one virtual machine can interact with threads on another virtual machine as if they were all on the same virtual machine with an unlimited amount of heap, see figure 6.3. Thus, when a business service modifies some data in a client's HTTP session object, the change is made visible via NAM to other servers in the cluster. To achieve this functionality, Terracotta uses bytecode manipulation techniques similar to those used by many AOP frameworks. Because there is no developer API specific to Terracotta, it allows to cluster any application written for a single JVM.

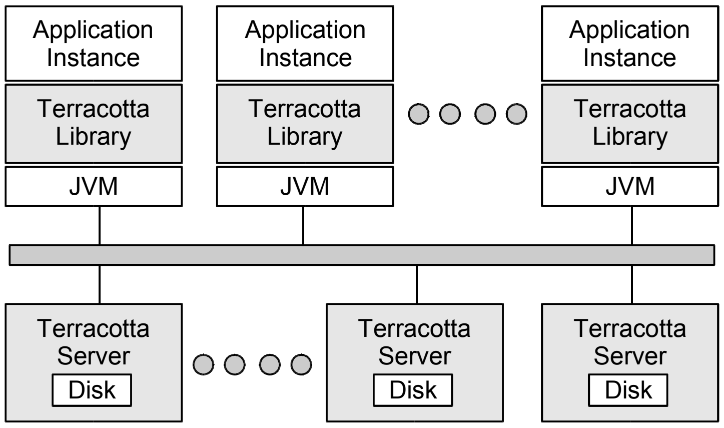

Figure 6.3: Terracotta architecture

Similar to EJB load balancing, the major EJB container vendors support stateful session bean replication in various ways. Basically, the process works as follows: When a stateful session bean is created, the state information is copied to one or more other servers in the cluster. The bean is routinely synchronized with its backups to ensure that all locations are current. If the primary server fails, another backup server will take over according to failover policy. Real implementations of state replication can be based on many techniques such as in-memory replication, the use of distributed shared cache or shared persistent storage [9]; some of them are shown in the following figure.

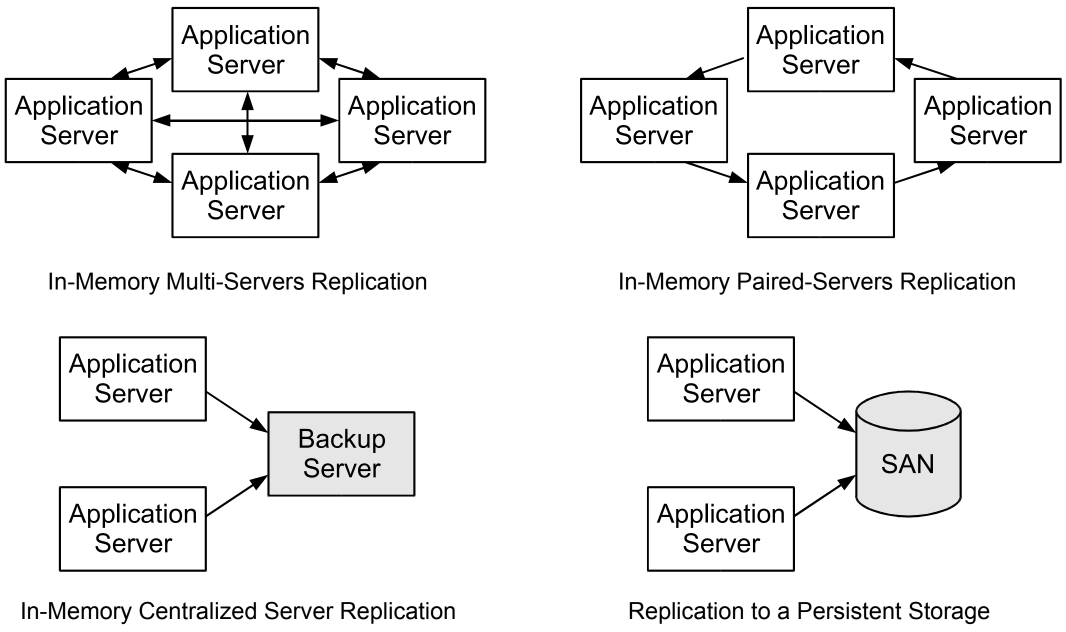

Figure 6.4: EJB state replication techniques

6.1.3 Summary

In this section we have familiarized ourselves with the basic types of JEE architectures. As we have seen, both Spring and EJB support building the most common web applications based on collocated architecture, with the difference that Spring-based applications need only a web server for operation while EJB-based ones require a full-fledged application server. In contrast to Spring, EJB also allows to separate web and business service tier physically to build more complex distributed web applications if such a requirement occurs.

Further, we have discussed fundamental clustering techniques to scale-up applications horizontally. Because JEE specification lacks any support for clustering, as web as EJB tier clustering is usually implemented in a vendor-specific way. While EJB can take advantage of proprietary clustering mechanisms provided by application servers, Spring draws on third party clustering solutions which can be used in various server runtime environments. In fact, the level of support for clustering is one of the main market-driven factors in the area of application servers.

Feature	Spring 2.5	EJB 3.0
Characteristic architecture	collocated	distributed, collocated
Deployment unit	JAR	EJB JAR
Load balancing	HTTP request level	method invocation level, HTTP request level
Failover	server level	component level, server level
State replication	JVM-level clustering	application-server-specific mechanisms (in-memory replication, distributed shared cache, shared persistent storage)

Table 6.1: Architecture overview

6.2 Testing

In current enterprise level applications it is more and more necessary to have a suite of tests to be executed automatically in order to ensure that the given code behaves the way a developer intended it to behave. Hence, integration testing into the entire development process is important, even if we are not practising the *test-driven development* (TDD) [64]. Beside many testing methods the most fundamental ones are *unit* and *integration testing*.

Unit testing is focused on testing a single unit of functionality, usually a single Java class, in isolation; all dependencies are replaced by their fake implementations by *stubbing* or *mocking* [1]. By contrast, integration testing is aimed at testing how application classes or components work together; in such a testing scenario a tested object is wired up to its collaborators and other resources like databases, which often need to be considered in any testing strategy. Sometimes resources may be tightly coupled to server infrastructure, then there is no choice but to run integration tests within the application sever. Requiring bootstrapping the server and deploying the application into it, this makes integration testing more complicated and time-consuming. Therefore, it is desirable to be able to execute most of integration tests out of the server, if possible.

Unit and integration testing are very similar in common; in practice we can use the same tool, such as JUnit [65], for writing both types of tests. Considering this fact, these testing methods are not discussed separately further in the text, rather more attention is paid to out-of-server and in-server testing.

6.2.1 Testing out of the Server

When preparing a unit or integration test, a lot of code needs to be usually written to setup the test, especially to wire up tested objects and their dependencies, no matter whether mocks or real implementations, together. Apparently, this issue might be easily addressed by dependency injection, or more precisely by IoC capabilities of both frameworks. Thus, the question how to setup the Spring IoC container or the EJB container in a testing environment, out of an application server, comes into mind. This kind of integration is even more desirable when we consider the fact that to properly wire up tested objects we only need to slightly modify their production configuration.

As for EJB, the specification does not define any support for running the EJB container standalone, in an embeddable mode; the EJB container is in most situations tightly coupled to the application server. Nevertheless, as unit and integration testing has become increasingly crucial to application development, some vendors has introduced embeddable versions of their EJB containers, namely JBoss [66].

The JBoss embedded EJB container provides a managed environment with support for the same basic services that exist within a JEE runtime: dependency injection, access to a bean environment, container-managed transactions, and more. The client uses a proprietary bootstrapping API to start the container and identify the set of EJBs for execution. The use of this container is illustrated in the following code snippet, which shows the JUnit test for testing the `addProduct(..)` method of the `StockDao` bean introduced in section 5.2.3.

```
public class StockDaoTest {

  private IStockDao stockDao;

  @BeforeClass // to be run before test method execution
  public void initialize() throws Exception {
    // bootstrapping embeddable EJB container
    EJB3StandaloneBootstrap.boot(null);
    EJB3StandaloneBootstrap.deployXmlResource("jboss-jms-beans.xml");
    EJB3StandaloneBootstrap.deployXmlResource("testjms.xml");

    // scanning classpath for EJBs
    EJB3StandaloneBootstrap.scanClasspath();

    // context initialization
    Hashtable props = new Hashtable();
    props.put("java.naming.factory.initial",
      "org.jnp.interfaces.LocalOnlyContextFactory");
    props.put("java.naming.factory.url.pkgs",
      "org.jboss.naming:org.jnp.interfaces");
    InitialContext context = new InitialContext(props);

    // setting test fixture
    stockDao = (IstockDao) context.lookup(IStockDao.class.getName());
  }

  @Test
  public void addProduct() {
    int numOfProductsBefore = stockDao.getNumberOfProductsInStock();
    Product newProduct = … // create new product
    stockDao.addProduct(newProduct);
    int numOfProductsAfter = stockDao.getNumberOfProductsInStock();
    assertEquals(numOfProductsAfter, numOfProductsBefore + 1);
    stockDao.deleteProduct(newProduct); // not to affect database
  }
}
```

Code Snippet 6.1: JUnit test using JBoss embedded EJB container

Since embedded EJB containers are not standardized, there exist a wide range of other testing tools, for example projects Ejb3Unit [67] or Pitchfork [68]. These tools try to mimic the EJB container's facilities, particularly dependency injection, supporting the EJB configuration annotations defined by the specification. To change the situation, the EJB 3.1 specification has defined the requirements for the execution of EJB applications within a Java SE environment, providing better support for testing.

Not being dependent on any server, Spring IoC container can be easily utilized for testing in a very similar way as the JBoss embedded EJB container. In addition, Spring offers support for unit and integration testing in the form of the *Spring TestContext Framework*, which is agnostic of the actual testing framework in use, allowing instrumentation of tests in various environments including JUnit. The Spring TestContext Framework addresses a lot of issues developers often face to when writing tests. Most notably, it facilitates testing in these areas:

- **Spring IoC container caching between test executions:** It supports the caching of loaded contexts among tests, avoiding the overhead associated with IoC container start-up process and object instantiation phase.
- **Dependency injection of test fixture instances:** It provides a convenient mechanism for injecting pre-configured objects directly into test classes, drawing on Spring's field and setter dependency injection support.
- **Transaction management appropriate to integration testing:** It allows to run tests modifying persistent data within a transaction, which is created and automatically rolled back by the framework at the end of each test in order not to affect the state of the persistence store for future tests.

All these features are demonstrated in the following code snippet, which presents the JUnit test similar to one we have seen above in the EJB scenario.

```
@RunWith(SpringJUnit4ClassRunner.class)
// loading Spring IoC container's configuration for this test
@ContextConfiguration(locations={"/dao-test.xml"})
// enabling support for transactional test method execution
@TransactionConfiguration(transactionManager="txMgr",
  defaultRollback=false)
public class StockDaoTest {

  @Autowire // this instance will be dependency injected by type
  private StockDao stockDao;

  @Test
  @Transactional // method will be executed within a transaction
  @Rollback(true) // transaction will be rolled back in the end
  public void addProduct() {
    int numOfProductsBefore = stockDao.getNumberOfProductsInStock();
    Product newProduct = … // create new product
    stockDao.addProduct(newProduct);
    int numOfProductsAfter = stockDao.getNumberOfProductsInStock();
    assertEquals(numOfProductsAfter, numOfProductsBefore + 1);
  }

}
```

Code Snippet 6.2: JUnit test using Spring TestContext Framework

6.2.2 Testing within the Server

As mentioned previously, in some circumstances there is no choice but to test the application deployed in the target server. In such testing scenarios, integration tests also run within the server, calling business layer services being tested. As these tests may be considered to be imitating the presentation layer, they are usually implemented as servlets. Many frameworks for in-server testing, such as Jakarta Cactus [69], are based on this concept, providing automated tasks to automatically start the server, deploy the application, run the tests and stop it, thus automating the entire test process. These types of frameworks are particularly used for EJB integration testing because of the difficulties to run the EJB container standalone, in a Java SE environment.

6.2.3 Summary

In this section we have discussed how Spring and EJB allow for the most fundamental testing techniques, unit and integration testing. As we have seen, Spring address this task by providing the Spring TestContext Framework, which allows to test applications in a consistent way, offering a lot of useful features to make testing as simple as possible. By contrast, due to the lack of any support from the EJB specification, EJB testing is more complicated despite the existence of a wide range of testing tools and frameworks.

Feature	Spring 2.5	EJB 3.0
Out-of-server testing	Spring TestContext Framework	embeddable EJB container, Ejb3Unit, Pitchfork
In-server testing	Jakarta Cactus	Jakarta Cactus

Table 6.2: Testing overview

6.3 Configuration

Beside many good practices that are advisable to follow when developing large-scale applications, it is vital to adopt a consistent way in their configuration. In common situations application configuration is partly scattered over source code by using annotations to achieve fine-tuned settings per application object, and partly externalized to allow application parametrization and reconfiguration, usually during the deployment process. Apparently, without taking a consistent approach, we can end up with holding configuration on several places and in a variety of files, which decreases maintainability of the application. Therefore, some kind of supporting infrastructure to centralize configuration management is desirable, ensuring consistency throughout the application. In this section we focus on how the examined frameworks address this issue. We discuss what kind of infrastructure they utilize to hold application settings in and the ways applications can be configured.

6.3.1 Infrastructure for Holding Configuration

In fact, when concerning dependency management in section 4.4, we have already familiarized ourselves with the central parts of both frameworks to hold application configuration in. The EJB container employs the JNDI service [31] while the bean factory keeps the configuration of managed beans on its own in regular JavaBeans [32], namely in `org.springframework.beans.factory.config.BeanDefinition` objects. The moment a new EJB instance is created, the container initializes it depending on its environment settings, which are made available to the bean via its private JNDI context. When the bean factory is asked for a new instance of a managed bean, it configures it using the standard JavaBean API according to the corresponding `BeanDefinition`, and, in case the bean wishes to be aware of its environment, it injects itself into the bean.

As can be seen, Spring and EJB manage application configuration in very different ways. Obviously, the design decisions about the underlying infrastructure have been made in respect of their original objectives. Since EJB in its first versions was primarily aimed at building distributed applications, the JNDI service allowing to locate distributed objects elsewhere on the network was a good choice at that time. On the other hand, as EJB has begun to operate more in the area of collocated applications, it has turned out that JavaBean-based configuration approach taken by Spring is superior to the JNDI for such types of applications [2]. This is particularly evident when configuring fine-grained objects. The JavaBean API provides abilities to convert a string parameter to any primitive type or a custom object if necessary using the appropriate `java.bean.PropertyEditor`, thus, one does not need to declare the Java type of the parameter explicitly in the configuration file. Another good example might be the bean factory's support for arrays and standard Java collections configuration. As for the JNDI service, such a fine-grained application configuration is not so straightforward, often requiring more effort.

6.3.2 Ways of Configuring Applications

While it is desirable to externalize some things from program source code, some important settings arguably belong to it. For example, it seldom makes sense to modify the transactional characteristics declared at method level, although parameters like transaction timeouts may change. Hence, a good application framework should provide capabilities to configure as *where* in the source code services need to be applied to as *how* these services are set-up at application level. The first type of configuration is particularly used during development when services are declared for each application object; the second type at deployment time when applications usually need to be reconfigured for the target runtime environment.

As EJB as Spring support this two-level configuration. Speaking of source code metadata, they both provide annotations in these main areas:

- **Auto-detection of managed objects:** @Stateless, @Stateful vs @Service annotation
- **Dependency injection of collaborators:** @EJB vs @Autowired annotation
- **Dependency injection of resources:** both frameworks support @Resource annotation
- **Configuration of provided services:** framework and service-specific annotations

If necessary, the source-level configuration of each of managed objects may be overridden in the EJB deployment descriptor and Spring XML configuration file, respectively. Further, in these configuration files global settings of all the frameworks' services may be defined according to deployment requirements.

In addition, Spring offers third-level configuration. It provides a mechanism, based on `org.springframework.beans.factory.config.BeanFactoryPostProcessor objects`, for post-processing of the bean definitions that have been read in by the underlying bean factory, allowing to override certain property values or to resolve place-holders in property values. This enables applications to keep some administration settings in external properties files. As a result an application administrator does not have to understand the Spring XML configuration file at all. The following code snippet illustrates this kind of configuration.

```
...
<!-- XML configuration -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
...

# properties file configuration
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost
jdbc.username=admin
jdbc.password=password
```

Code Snippet 6.3: Support for administration configuration files by Spring

6.3.3 Server Configuration

Beyond standard application configuration we need to set up the target server. This step involves creating the server-wide resources the application relies on at runtime, such as database connection pools and JMS destinations, and writing proprietary deployment descriptors, which are required to provide additional configuration that fills gaps left by the standard WAR and EJB deployment descriptors. Typically, they contain information about mapping the JNDI names of the resources used in application code onto their real representations, server authentication mechanisms being used, and more. Further, EJB settings may include pooling options, cluster-related options, transaction isolation levels, and other vendor-specific functionalities the EJB specification does not cover, which makes EJB-based applications less portable between various application servers. To simplify this task developers usually utilize tools such as XDoclet [54], which generate server configuration automatically.

A common issue during the deployment process is how to ensure that certain code is executed at application start-up time, for example initialization of timers important for preforming recurrent tasks. To achieve this goal Spring-based web applications can use standard means provided by the servlet specification [33]. The first option is to encapsulate start-up processing into a bootstrap servlet that should have a lower `load-on-startup` value in the WAR deployment descriptor than the other servlets in order to be initialized as the first one. Another way is to register an `javax.servlet.ServletContextListener` implementation which will be notified by the web container at the time the web application initialization process has been started. In addition, if there is a need to define explicitly the order one or more beans need to be initialized, Spring allows to configure these indirect dependencies by using the "depends-on" attribute. Speaking of EJB, the specification lacks any support for start-up processing. Therefore, one has no choice but to use a server-specific mechanism, or implement some workarounds, such as sending an initialization event to an MDB from the web tier, if portability is required.

6.3.4 Summary

To sum up, both Spring and EJB provide support for configuring applications in the two basic ways: via source code metadata and via configuration files. Spring also allows to externalize some settings into simple properties files, which are more readable for application administrators. Speaking of the underlying infrastructure, it has been chosen with the respect to the original aims of the frameworks, i.e., to support building collocated applications and distributed applications, respectively. Therefore, Spring's approach based on the JavaBean API is more suitable for configuring fine-grained objects while the JNDI service addresses well the issues related to object distribution. Finally, to ensure that some processing occurs when the application starts up, Spring-based applications can utilize standard facilities provided be the servlet specification, by contrast, EJB-based ones are usually dependent on vendor-specific mechanisms because the specification does not offer any standard means

Feature	**Spring 2.5**	**EJB 3.0**
Configuration infrastructure	bean factory	JNDI
Source code metadata	annotations	annotations
Deployment configuration	XML file	EJB deployment descriptor
Administration configuration	properties file	none
Application start-up processing	bootstrap servlet, `ServletContextListener`, start-up order of beans	application-server-specific mechanisms

Table 6.3: Configuration overview

7 Conclusion

7.1.1 Results of the Comparison

We have analysed Spring and EJB from three main point of views: business object management, provision of services, and application development. As we could seen, they both provide very similar functionality in a lot of areas, although addressing related issues sometimes in different ways, at other times in similar ways at the design level. Thus, for building some types of applications they both are a viable choice. The overlapping areas are summarized in the following table.

Comparison Area	Spring 2.5 & EJB 3.0 Feature
Business Object Management	– POJO-style managed object contract – optional lifecycle callbacks – support for stateless objects – field and setter injection – pooling of managed objects
Services	– method-level interception – integration with ORM solutions – declarative transaction management – support for RMI remoting – support for web services – support for asynchronous message-driven objects – declarative role-based authorization
Application Development	– support for building collocated web applications – support for unit testing – configuration via source code metadata – configuration via XML-based files

Table 7.1: Spring and EJB's overlapping areas

On the other hand, they both offer many features that make them superior to each other for some kinds of application development efforts. Apparently, these additional capabilities may be determining factors for choosing one framework or the other according to given requirements. The table below details these most obvious Spring and EJB's strengths.

Comparison Area	Spring 2.5 Feature	EJB 3.0 Feature
Business Object Management	– no programming restrictions – bean scopes – many ways of creating and initializing objects – constructor injection – DI of stateful objects into stateless objects – autowiring features	– distributed objects – complete lifecycle management of stateful objects
Services	– pointcut language – support for mixins – data access convenience classes – easy switching between transaction strategies – easy switching between remoting protocols – customizable security mechanisms – support for domain objects security – declarative cron-style scheduling	– transaction propagation upon remote method invocation – security context propagation upon remote method invocation
Application Development	– no dependency on an application server – simpler deployment process – immense support for integration testing – application start-up processing facilities	– support for distributed applications – clustering support by application servers

Table 7.2: Spring and EJB's strengths

On the basis of table 7.2, several conclusions may be drawn. Spring is particularly appropriate for those applications which:

- need as a runtime environment a web server (6.1.1),
- are formed from a lot of collocated fine-grained objects requiring fine-tuned dependency resolution and configuration (4.4, 6.3.1),
- heavily utilize AOP and require advanced AOP features (5.1),
- need flexibility in terms of switching between various transaction strategies, remoting technologies, and security mechanisms (5.3.1, 5.4.1, 5.6),
- harness test-driven development, thus testing is of a big importance (6.2),

- require customizability and extensibility in terms of providing custom implementations of some infrastructural components by utilizing extension points provided by the framework (4.2.2, 4.3.2, 6.3.2).

By contrast, EJB is well suitable especially for building applications which:

- are distributed in their nature (3.2.1, 5.4.1, 6.1.1),
- require a high level of interoperability, particularly with CORBA-based systems (5.3.1, 5.6.1.1),
- are very stateful, supporting as web as non-web clients which both need state in the business service layer (4.3).

7.1.2 Relation to Other Comparative Analyses

Although the question of the relative strengths and weaknesses of Spring and EJB is frequent one may often encounter in discussion forums on the Internet, there are, surprisingly, not many valuable sources of objective information comparing these two technologies as discussed in section 2.1. This comparative analysis is a reaction to this fact, considering usual shortcomings that other public-available materials often suffer.

First of all, this study gives a comprehensive overview of all main capabilities of both frameworks; it does not discuss only selected subjects, which other information sources usually do [23]. For this reason it is well-organized into the three logical comparison areas with tightly-focused sections. Speaking of the structure of sections, the approach taken by [22] is followed, nevertheless, each section is thoroughly divided into an introduction, analysis, and summary part, delving in more details of a given topic than [22]. In this sense the study also avoids just listing frameworks' features without any in-depth explanation, which can be seen particularly in [3]. Further, putting discussed subjects in an appropriate context, it compares frameworks' features side-by-side for better understanding their similarities and distinctions, not describing them separately as [26] which resembles a short version of the reference documentation rather than a comparative analysis. Finally, it pays attention not to mix features of other frameworks, especially ORM frameworks such as Hibernate, together with those of Spring and EJB; this is particularly evident in [22].

7.1.3 Summary

Adopting a framework is a very important decision that can have a crucial impact on the project's success or failure. The more responsible role is assigned to a framework in the application's architecture, the more important the choice of the best fitting framework is. This is particularly true for application frameworks like Spring and EJB. Therefore, before making a commitment it is vital to carry out a thorough evaluation.

A common approach to the evaluation of a framework is to make a checklist of the features the framework has to conform to. This study helps to conduct such an evaluation of both Spring 2.5 and EJB 3.0, providing all of the facts necessary to make well-informed decisions. Analysing all main capabilities of both frameworks and discussing their strengths and weaknesses, it allows selecting an appropriate framework for definite requirements satisfaction. Hence, the study can be useful for IT companies to assist in business component management framework selection and usage.

Since any comparison of EJB and Spring may rather be considered a controversial topic, the thesis does not aim to decide whether EJB is superior to Spring for developing all possible types of applications, or vice versa. It only shows distinction between these frameworks, or more precisely between their features, giving some recommendations. Every application has its own set of requirements and it is up to developers to choose which framework to use. In some situations Spring may be the right solution, in others EJB; as we all know, there is no silver bullet. In addition, EJB and Spring complement each other, rather than compete with each other. There are certain features which are powerful in Spring, and equal number of features powerful on the EJB side as well.

7.1.4 Future Work

As Spring and EJB are continually evolving, features of both frameworks can be re-examined with their new versions, such as with upcoming Spring 3.0 and EJB 3.1. The structure of the analysis is very flexible, thus, new areas of comparison can easily be added or the current ones can be extended to augment the scope of the study.

Appendix A: The Sample Application

A.1 Description

To illustrate Spring and EJB's capabilities, the DVD enclosed contains two new versions of the SSDF server, a part of the SSDF application which was originally developed as a "software project" on MFF UK in 2006. To put it simple, SSDF allows users to publish their photos on the Internet via its central server. The SSDF server is a simple web application implemented in Java; its presentation layer is based on the Struts framework, the business service layer and the data access layer are formed by POJOs. For more information about SSDF please consult the user and technical documentation on the DVD.

A.2 Re-engineering of the Application

The original version of the SSDF server does not take advantage of any application framework to manage business and DAO objects. Apparently, this makes a room for its re-engineering, thus, two new versions of the SSDF server has been implemented; one is Spring-based, the other is EJB-based. Further in the text the consequences of the adoption of both frameworks are discussed.

A.2.1 Packaging

The original SSDF server is packaged as a WAR. The adoption of Spring has not influenced the way the SSDF server is packaged, by contrast, the EJB-based version is packaged as an EAR with web components assembled in a web module, and business and DAO objects assembled in an EJB module. Because of class loading restrictions, classes shared between the WAR and EJB JAR have been placed into the EJB JAR, further, shared libraries have been placed directly into the EAR.

A.2.2 Dependency Injection of Collaborators

Within the original SSDF server, web components create new instances of business service objects they depend on per HTTP request, although business service objects are thread-safe in their nature; in a similar way DAO objects are instantiated within the business service layer. Both Spring and EJB have simplified the way the objects are wired up. In both scenarios a service-locator-pattern-based approach has been implemented to allow components in the presentation layer to access business services via the dependency lookup concept (see the `cz.ssdf.server.actions.BaseAction` class). Speaking of dependencies between business services and DAO objects, the dependency injection mechanism has been applied via Spring-specific and EJB-specific annotations.

A.2.3 Dependency Injection of Resources

The only resource the original SSDF server depends on is a `javax.sql.DataSource` object to get access to the underlying database. This object is managed by the Struts framework and propagated through the layers, from the presentation layer to the data access layer. This drawback has been eliminated as in the Spring-based as in the EJB-based version of the SSDF server; the `DataSource` object is managed by the Spring and EJB container, respectively. Thus, its propagation through the layers has been avoided.

A.2.4 Application Start-Up Processing and Sharing Data Across Layers

The original SSDF server performs a start-up processing in the `cz.ssdf.server .SSDFActionServlet` which collects some configuration information, putting it into a `cz.ssdf.server.ServerConfig` object held within the servlet context. Further, because the `ServerConfig` object encapsulates data important as for web components as for business services, it is propagated to the business service layer upon each method invocation. Speaking of start-up processing, both Spring-based and EJB-based SSDF server draw on the same servlet-based approach as the original one. As regards to data sharing, within the Spring-based SSDF server the `ServerConfig` object is managed by the Spring container, thus being available for both layers in a consistent way, by contrast, the EJB-based SSDF server addresses data sharing in the same way as the original one in order to avoid sharing the `ServerConfig` object via the JNDI service.

A.2.5 Transaction Management

Accessing only a single database to store persistent data, the original SSDF server manages local transactions in a programmatic way in the data access layer. Both Spring and EJB have helped to move transaction management to the right place, the business service layer, by using declarative configuration. In contrast to the Spring-based SSDF server, to achieve this kind of functionality in the EJB-based SSDF server, it was necessary to use a `javax.sql.XADataSource` object instead of the original `DataSource` because the EJB container does not support local transactions, but only distributed ones.

A.2.6 Data Access

To access the underlying database, the original SSDF server uses the JDBC technology, utilizing some helper classes to make data access easier. Further, it defines a set of data-access-layer-specific runtime exceptions to inform business services about various non-standard situations. Using provided convenience classes for JDBC-based data access, such as `org.springframework.jdbc.core.simple.SimpleJdbcTemplate`, the application logic of DAOs has been simplified and the helper classes have been removed from the Spring-based SSDF server. Because EJB does not offer any similar facilities, the data access logic of the EJB-based server has remained the same as in the original SSDF server.

A.2.7 Remoting

Besides standard web browser clients, the original SSDF server supports rich standalone clients via its special HTTP-based protocol. Because the remoting is handled by the Struts framework within the presentation layer, no changes have been implemented in both Spring-based and EJB-based version of the SSDF server.

A.2.8 Security

Similar to remoting, security policies are enforced in the original SSDF server by the Struts framework per each Struts `org.apache.struts.action.Action` class servicing HTTP requests. As there are no clients accessing business service objects directly, declarative authorization in the business service layer has not been applied in any of the two SSDF servers.

A.3 Summary

To sum up, the adoption of both Spring and EJB has been beneficial; a lot of issues have been addressed in a more convenient ways. Nevertheless, the use of Spring seems to be more suitable for this type of collocated application because of its better support for several tasks, particularly sharing data across layers and JDBC data access.

References

1. Johnson R.: Expert One on One J2EE Design and Development, Wiley Publishing, Inc., 2003
2. Johnson R., Hoeller J.: Expert One on One J2EE Development without EJB, Wiley Publishing, Inc., 2004
3. Graudins J., Zaitseva L.: Comparative Analysis of EJB3 and Spring Framework, 2006, http://ecet.ecs.ru.acad.bg/cst06/Docs/cp/SIII/IIIA.18.pdf
4. Paleta P.: Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi, Computer Press, 2003
5. JSR 153: Enterprise JavaBeans, version 2.1, Sun Microsystems, November 2003
6. Gamma E., Helm R., Johnson R., Vlissades J. O.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
7. Fowler M.: Patterns of Enterprise Application Architecture, Addison-Wesley, 2002
8. Sarang P. G. et al.: Professional EJB, Wrox Press, July 2001
9. Roman E., Sriganesh R. P., Brose G.: Mastering Enterprise JavaBeans, Third Edition, Wiley Publishing, Inc., 2005
10. Fayad M. E., Schmidt D. C.: Object-Oriented Application Frameworks, October 1997, http://www.cs.wustl.edu/~schmidt/CACM-frameworks.html
11. Sriganesh R. P., Brose G., Silverman M.: Mastering Enterprise JavaBeans 3.0, Wiley Publishing, Inc., 2006
12. JSR 250: Common Annotations for the Java Platform, Sun Microsystems, 2006
13. JSR 220: Enterprise JavaBeans, version 3.0, EJB Core Contracts and Requirements, Sun Microsystems, May 2006
14. JSR 220: Enterprise JavaBeans, version 3.0, Java Persistence API, Sun Microsystems, May 2006
15. The Spring Framework Reference Documentation, version 2.5, SpringSource, 2008
16. Hibernate Reference Documentation, version 3.3, Red Hat Middleware, 2008
17. The J2EE 1.4 Tutorial, Sun Microsystems, December 2005
18. The JEE 5.0 Tutorial, Sun Microsystems, June 2006
19. The Spring Framework Reference Documentation, version 3.0, SpringSource, 2009
20. JSR 318: Enterprise JavaBeans, version 3.1, Sun Microsystems, 2009
21. Hunt A., Thomas D.: The Pragmatic Programmer, Addison-Wesley, 2000
22. Coffin R.: Spring and EJB 3 Comparison, 2006, http://wiki.rodcoffin.com/index.php?title=Spring_and_EJB_3_Comparison

23. Yuan M. J.: POJO Application Frameworks: Spring Vs. EJB 3.0, 2005, http://www.onjava.com/pub/a/onjava/2005/06/29/spring-ejb3.html
24. Rahman R.: EJB 3, Spring and Hibernate: A Comparative Analysis and Recommendations, 2007, http://phillyjug.jsync.com/file_download/32
25. Alur D., Crupi J., Malks D.: Core J2EE Patterns: Best Practices and Design Strategies, 2nd edition, Prentice Hall/Sun Microsystems Press, 2003
26. Kleiman M.: Porovnání EJB 3.0 a Spring 2.5, master thesis, ČVUT, January 2008
27. Ghaffaripour R.: EJB 3.0 and Spring, June 2008, http://www.rezagh.com/docs/ejb_spring.pdf
28. AOSD homepage, http://www.aosd.net
29. The Inversion of Control principle, http://martinfowler.com/bliki/InversionOfControl.html
30. The Remote Method Invocation (RMI) technology, http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper
31. The Java Naming and Directory Interface (JNDI) technology, http://java.sun.com/products/jndi/tutorial
32. The JavaBeans technology, http://java.sun.com/docs/books/tutorial/javabeans
33. JSR 154: Java Servlet Specification, version 2.5, Sun Microsystems, May 2007
34. JSR 286: Portlet Specification, version 2.0, Sun Microsystems, June 2008
35. The Java Reflection API, http://java.sun.com/docs/books/tutorial/reflect
36. JSR 224: Java API for XML-Based Web Services 2.1, Sun Microsystems, May 2007
37. Web Services Description Language (WSDL), version 2.0, the World Wide Web Consortium (W3C), June 2007
38. JSR 101: Java APIs for XML based RPC 1.1, Sun Microsystems, 2006
39. Fielding R. T.: Architectural Styles and the Design of Network-based Software Architectures, dissertation, University of California, 2000
40. JSR 311: The Java API for RESTful Web Services 1.0, Sun Microsystems, September 2008
41. Richardsom C.: POJOs in Action, January 2006, Manning Publications Co.
42. AOP Alliance homepage, http://aopalliance.sourceforge.net
43. Spring homepage, http://www.springsource.org
44. AspectJ homepage, http://www.eclipse.org/aspectj
45. Sun homepage, http://java.sun.com
46. CORBA homepage, http://www.corba.org
47. The Java Message Service (JMS) API, http://java.sun.com/products/jms
48. Java Authentication and Authorization Service (JAAS) reference guide for the Java SDK 1.4
49. Alex B., Taylor L.: Spring Security Reference Documentation 2.0.x, SpringSource

50. Zhu W.: Service-context propagation over RMI, http://www.javaworld.com/javaworld/jw-01-2005/jw-0117-rmi.html?page=1
51. Hartman B., Flinn J. D., Beznosov K.: Enterprise Security with EJB and CORBA, John Wiley & Sons, April 2001
52. Quartz homepage, http://www.opensymphony.com/quartz
53. Apache MyFaces Orchestra homepage, Apache Foundation, http://myfaces.apache.org/orchestra
54. XDoclet homepage, http://xdoclet.sourceforge.net,
55. CGLIB homepage, http://cglib.sourceforge.net
56. Jakarta Commons Pool homepage, http://jakarta.apache.org/commons/pool
57. Kopparapu Ch.: Load Balancing Servers, Firewalls, and Caches, Wiley, January 2002
58. Wang Yu: Uncover the hood of J2EE Clustering, http://www.theserverside.com/tt/articles/article.tss?l=J2EEClustering
59. Bracha G., Cook W.: Mixin-based Inheritance, 1990
60. Terracotta, Inc.: The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability, Apress, June 2008
61. JSR 907: The Java Transaction API, Sun Microsystems, November 2002
62. Distributed TP: The XA Specification, The Open Group, February 1992
63. The Java Transaction Service (JTS), version 1.0, Sun Microsystems, December 1999
64. Beck K.: Test Driven Development: By Example, Addison-Wesley, November 2002
65. JUnit homepage, http://www.junit.org
66. JBoss homepage, http://www.jboss.com
67. EJB3Unit homepage, http://ejb3unit.sourceforge.net
68. Pitchfork homepage, http://www.springsource.com/pitchfork
69. Jakarta Cactus homepage, http://jakarta.apache.org/cactus