Comenius University, Bratislava

Faculty of Mathematics, Physics and Informatics

Overview of implementation of Java EE in JBoss AS 7

Bachelor Thesis

Erich Siebenstich **Bratislava, 2013**

Comenius University, Bratislava

Overview of implementation of Java EE in JBoss AS 7

Bachelor Thesis

Study program: Informatics

Branch of Study: 2508 Informatics

Department: Department of Computer Science

Advisor: RNDr Tomáš Kulich, PhD

34602201

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Erich Siebenstich
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický

Názov:	Prehľad implementácie Java EE v JBoss AS 7
Cieľ:	Spraviť prehľad funckionality aplikačného servra JBoss 7, špeciálne zameraný na EJB3, transakcie a dependency injection

Vedúci:	RNDr. Tomáš Kulich, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	17.10.2012

Dátum schválenia: 24.10.2012

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....

študent vedúci práce

34602201

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Erich Siebenstich
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Overview of Java EE on JBoss AS 7

Aim: Create overview of functionality of application server JBoss AS 7. Pay special attention on EJB3, managing transactions and dependency injection.

Supervisor:	RNDr. Tomáš Kulich, PhD.
Department:	FMFI.KI - Department of Computer Science
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	17.10.2012

Approved: 24.10.2012 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

I would like to thank my family for their support and patience, as well as my supervisor RNDr. Tomáš Kulich, PhD for his advices and help.

Abstract

ENG version

Author: Erich Siebenstich

Thesis title: Overview of implementation of Java EE on AS JBoss 7

University: Comenius University, Bratislava

Faculty: Faculty of Mathematics, Physics and Informatics

Department: Department of Computer Science

Advisor: RNDr Tomáš Kulich, PhD

Thesis Length: 48

June 2013, Bratislava

Main goal of this bachelor thesis is to overview capabilities Java EE platform of programing language Java. As a facilitating medium was chosen application server JBoss 7. Version 7 was the latest stable version during the time this thesis has been written. This thesis contains examples of usage of many key technologies of Java EE. It is mostly focused on technologies CDI, EJB3 and Transactions. The thesis is divided into 6 chapters.

KEY WORDS: CDI, EJB, Transactions, Entity Beans, Session Beans, Bean-Managed Transactions, Container-Managed Transactions.

Abstrakt

SVK version

Autor: Erich Siebenstich

Názov práce: Prehľad implementácie Java EE na AS JBoss 7

Škola: Univerzita Komenského v Bratislave

Fakulta: Fakulta Matematiky, Fyziky a Informatiky

Katedra: Katedra Informatiky

Vedúci bakalárskej práce: RNDr Tomáš Kulich, PhD

Rozsah: 48

Jún 2013, Bratislava

Cieľom tejto bakalárskej práce je predviesť možnosti platformy Java EE, programovacieho jazyka Java. Ako médium bol zvolený aplikačný server JBoss 7. Verzia 7 bola v čase písania tejto bakalárskej práce najnovšia stabilna verzia. Táto práca obsahuje príklady používania kľúčových technológií Java EE. Práca je predovšetkým zameraná na technológie CDI, EJB3 a Transactions. Práca je rozdelená do šiestich kapitol.

KĽÚČOVÉ SLOVÁ: CDI, EJB, Transactions, Entity Beans, Session Beans, Bean-Managed Transactions, Container-Managed Transactions.

Table of Abbreviations

CDI Context and Dependency Injection

EJB Enterprise JavaBeans

CMT Container-Managed Transactions

BMT Bean-Managed Transactions

Java SE Java Standard Edition

Java EE Java Enterprise Edition

POJO Plain Old Java Object

API Application Programming Interface

HTTP Hypertext Transfer Protocol

JDBC Java Database Connectivity

JTA Java Transaction API

JPA Java Persistence API

JSP Java Server Pages

JSF Java Server Faces

Table of Contents

Intro

JavaBeans Open Source Software Application Server (JBoss AS or simply JBoss) is an application server that implements the Java Platform, Enterprise Edition (Java EE).

Name of the JBoss AS has changed throughout the years from its original name of EJB-OSS (stands for Enterprise Java Bean Open Source Software), to JBOSS because of use of trademark EJB. Later it changed to JBoss, the name used in this thesis, and just recently (January of 2013) a new name has been voted in by members of JBoss community; WildFly.

JBoss is an application server which offers many benefits. It has been designed for flexibility, it has amplified speed and it takes only a moment to create web applications. But as is obvious even from its name, it is a server with special focus on Java EE.

JBoss AS 7.1, the current stable version, was released in February 2012. The remaining parts of the EE spec were implemented, and this version was certified for the EE full profile. All the examples supporting this thesis have been created to work with version 7.1 and it is highly recommended to use it when deploying them.

Java EE was initially released in 1999. Since then it has been quickly developing, just as it competition Spring, and transformed through better and worse phases of its existence, to arrive in the presence as one of the best platforms for developing portable, large scale and robust and secure Java applications. This thesis works with Java EE 6 version with new version being introduced in June 2013. All information in upcoming chapters however will stay valid or there will be only small changes (hopefully for better).

The most important goal of the Java EE platform is to simplify development by providing a common foundation for the various kinds of components in the Java EE platform.

As a platform of Java, Java EE delivers a plain old java object (POJO) and convention over configuration approach to development. Developers begin with POJOs and annotate them with additional capabilities, such as transaction-enabled business components (i.e. @Stateless), database persistence (@Entity) and many others. Annotations can virtually replace XML configuration, greatly simplifying development.

This thesis concentrates on three core Java EE technologies – Enterprise JavaBeans (EJB), Context and Dependency Injection (CDI) and Transactions. Using these technologies you can quickly turn your regular Java SE application into efficient enterprise application that is connected with its web interface, database and even another application server using few annotations. All this, while staying loosely bound and easily modifiable.

First Chapter

What is application server?

Application server creates a layer between operation system and applications. Similarly to operation system which offers a programmer basic functions to manipulate with data and running processes, application server offers frequently used functions to **enterprise applications**. As such it creates another layer of abstraction to facilitate programming these applications.

To clarify what exactly enterprise application is and how it differs from regular application all that needs to be said is that enterprise application is regular application that is required to guarantee several conditions concerning its reliability, accessibility, robustness and effectivity. Typical condition is often the ability to handle substantial amount of clients. Modern web applications and solutions based on **service oriented architecture** (**SOA**) are common representatives of enterprise applications.

There are additional benefits of server after all the programs are up and running. Most important of these can be summarized in following categories:

- **Data and code integrity** – by centralizing business logic on single or small number of servers, application updates can be guaranteed, minimalizing risk of incompatibility between new and old versions
- **Security** – central point of accessing data decreases responsibility of authentication on potentially insecure client layer
- **Performance** – limiting client-server traffic, improves large applications performance
- **Transaction support** - as the server does a lot of the tedious code-generation, developers can focus on business logic

Most of application servers are developed in Java programming language. It’s not only because Java is platform independent but also because of a standard for implementation of enterprise applications is in Java as well. This standard is called Java Enterprise Edition (JEE).

Second Chapter

Java SE versus Java EE

Before we get started on the actual inner workings and different features of Java EE, it is important to clarify what exactly it is that makes Java EE unique compared to other Java platforms. And what is its place in the overall picture that is Java.

Java technology is both a programming language and a platform.

- The Java **programming language** is a high-level object-oriented language that has a particular syntax and style.
- A Java **platform** is a particular environment in which Java programming language applications run.

This thesis isn't overview of programming language different to Java that is so widely used and known. It is supposed to be an overview of features of Java programming language which take advantage of capabilities of this extended Java Platform. However since Java EE is an extremely broad subject, in order for this thesis to provide more than a shallow understanding of different technologies, it will concentrate on three closely knitted technologies.

- **Context and Dependency Injection** (CDI)
- **Enterprise JavaBeans 3.0** (EJB3)
- **Transactions**

All Java platforms consist of a Java Virtual Machine (VM) and an application programming interface (API). The Java Virtual Machine is a program, for a particular hardware and software platform, that runs Java technology applications. An API is a collection of software components that you can use to create other software components or applications. Each Java platform provides a virtual machine and an API, and this allows applications written for that platform to run on any compatible system with all the advantages of the Java programming language: platform-independence, power, stability, ease-of-development, and security.

There are four platforms of the Java programming language:

- Java Platform, **Standard Edition** (Java SE)
- Java Platform, **Enterprise Edition** (Java EE)
- Java Platform, **Micro Edition** (Java ME)
- **JavaFX**

Java SE

When most people think of the Java programming language, they think of the Java SE API. Java SE's API provides the core functionality of the Java programming language. It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing.

In addition to the core API, the Java SE platform consists of a virtual machine, development tools, deployment technologies, and other class libraries and toolkits commonly used in Java technology applications.

Java EE

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.

Java ME

The Java ME platform provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones. The API is a subset of the Java SE API, along with special class libraries useful for small device application development. Java ME applications are often clients of Java EE platform services.

JavaFX

JavaFX is a platform for creating rich internet applications using a lightweight user-interface API. JavaFX applications use hardware-accelerated graphics and media engines to take advantage of higher-performance clients and a modern look-and-feel as well as high-level APIs for connecting to networked data sources. JavaFX applications may be clients of Java EE platform services.

So what does Java EE do?

As stated above, the Java EE platform is designed to help developers create large-scale, multi-tiered, scalable, reliable, and secure network applications. A shorthand name for such applications is “enterprise applications,” so called because these applications are designed to solve the problems encountered by large enterprises. Enterprise applications are not only useful for large corporations, agencies, and governments, however. The benefits of an enterprise application are helpful, even essential, for individual developers and small organizations in an increasingly networked world.

Features that make enterprise applications sufficiently powerful, reliable and secure, however also result in significant complexity. The Java EE platform is designed to reduce the complexity of enterprise application development by providing a development model, API, and runtime environment that allows developers to concentrate on functionality.

Tiered Applications

In a multi-tiered application, the functionality of the application is separated into isolated functional areas, called tiers. Typically, multi-tiered applications have a **client tier**, a **middle tier**, and a **data tier** (often called the enterprise information systems tier).

- The client tier consists of a client program that makes requests to the middle tier.
- The middle tier's business functions handle client requests and process application data.
- The data tier functions handle long-term storing of information for middle tier.

The Client Tier

The client tier consists of application clients that access a Java EE server and that are usually located on a different machine from the server. The clients make requests to the server. The server processes the requests and returns a response back to the client. Many different types of applications can be Java EE clients, and they are not always, or even often Java applications. Clients can be a web browser, a standalone application, or other servers, and they run on a different machine from the Java EE server.

The Web Tier
Great amount of client-business tier communication goes through web. Because of this Java EE have several technologies to facilitate this exchange of information over web. Its primary tasks are the following:

- Dynamically generate content in various formats for the client.
- Collect input from users of the client interface and return appropriate results from the components in the business tier.
- Control the flow of screens or pages on the client.
- Maintain the state of data for a user's session.
- Perform some basic logic and hold some data temporarily in JavaBeans components.

Table 2.1 of Web Tier Java EE Technologies

Technology	**Description**
Servlets	Java programming language classes that dynamically process requests and construct responses, usually for HTML pages
JavaServer Faces technology	A user-interface component framework for web applications that allows you to include UI components (such as fields and buttons) on a page, convert and validate UI component data, save UI component data to server-side data stores, and maintain component state.
JavaServer Faces Facelets technology	Facelets applications are a type of JavaServer Faces applications that use XHTML pages rather than JSP pages.
Expression Language	A set of standard tags used in JSP and Facelets pages to refer to Java EE components.
JavaServer Pages (JSP)	Text-based documents that are compiled into servlets and define how dynamic content can be added to static pages, such as HTML pages.

JavaServer Pages Standard Tag Library	A tag library that encapsulates core functionality common to JSP pages
JavaBeans Components	Objects that act as temporary data stores for the pages of an application

The Middle (Business) Tier

The middle tier components and functions contain business logic for enterprise applications. Business logic is code providing functionality necessary for processing of requests and data incoming from client tier and to modify data stored in data tier. In properly design enterprise application all of core functionality is contained in business tier components.

Table 2.2 of Business Tier Java EE Technologies

Technology	**Description**
Enterprise JavaBeans (enterprise bean) components	Enterprise beans are managed components that encapsulate the core functionality of an application.
JAX-RS RESTful web services	An API for creating web services that respond to HTTP methods (for example GET or POST methods). JAX-RS web services are developed according to the principles of REST, or representational state transfer.
JAX-WS web service endpoints	An API for creating and consuming SOAP web services.
Java Persistence API entities	An API for accessing data in underlying data stores and mapping that data to Java programming language objects.
Java EE managed beans	Managed components that may provide the business logic of an application, but do not require the transactional or security features of enterprise beans.

The Data (Enterprise Information Systems) Tier

The enterprise information systems (EIS) tier consists of database servers, enterprise resource planning systems, and other legacy data sources, like mainframes. These resources typically are located on a separate machine than the Java EE server, and are accessed by components on the business tier.

Table 2.3 of Data Tier Java EE Technologies

Technology	Description
The Java Database Connectivity API (JDBC)	A low-level API for accessing and retrieving data from underlying data stores. A common use of JDBC is to make SQL queries on a particular database.
The Java Persistence API	An API for accessing data in underlying data stores and mapping that data to Java programming language objects. The Java Persistence API is a much higher-level API than JDBC, and hides the complexity of JDBC from the user.
JEE Connector Architecture	An API for connecting to other enterprise resources.
The Java Transaction API (JTA)	An API for defining and managing transactions, including distributed transactions or transactions that cross multiple underlying data sources.

Third Chapter

Technologies

Although the main focus of this thesis is not client tier technologies, they have been utilized in runnable examples provided with this thesis. Their purpose is to provide simple way of verifying that the example does, what it claims, to be able to do. As such it is fitting to introduce them and their uses in this thesis.

Java Servlets Technology

Servlets are excellent Java platform technology for extending and enhancing Web servers. They provide a component-based, platform-independent method for building Web-based applications, without the performance limitations of CGI (**Common Gateway Interface**) programs. And unlike proprietary server extension mechanisms (such as the Netscape Server API or Apache modules), servlets are server- and platform-independent. This leaves you free to create best strategy for your servers, platforms, and tools, without a need to pay attention to specific hardware.

Servlets have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. Servlets can also access a library of HTTP-specific calls and receive all the benefits of the robust Java language, including portability, performance, reusability, and crash protection. Today servlets are a popular choice for building interactive Web applications.

Closely knitted with servlets is JavaServer Pages (JSP) technology, which acts as an extension of servlet technology, created to support authoring of HTML and XML pages. It aids in combining fixed and static template data with dynamic content. Even if you're comfortable writing servlets, there are several compelling reasons to investigate JSP technology as a complement to your existing work.

JavaServer Pages (JSP) Technology

JavaServer Pages (JSP) technology enables Web developers and designers to rapidly develop and easily maintain, information-rich, dynamic Web pages that leverage existing business systems. As part of the Java technology family, JSP technology enables rapid development of Web-based applications that are platform independent. JSP technology separates the user interface from content generation, enabling designers to change the overall page layout without altering the underlying dynamic content.

Benefits for Developers

If you are a Web page developer or designer who is familiar with HTML, you can:

- **Use JSP technology without having to learn the Java language**: You can use JSP technology without learning how to write Java scriptlets. Although scriptlets are no longer required to generate dynamic content, they are still supported to provide backward compatibility.
- **Extend the JSP language**: Java tag library developers and designers can extend the JSP language with "simple tag handlers," which utilize a new, much simpler and cleaner, tag extension API. This spurs the growing number of pluggable, reusable tag libraries available, which in turn reduces the amount of code needed to write powerful Web applications.
- **Easily write and maintain pages**: The JavaServer Pages Standard Tag Library (JSTL) expression language is now integrated into JSP technology and has been upgraded to support functions. The expression language can now be used instead of scriptlets expressions.

JSP Technology and Java Servlets

JSP technology uses XML-like tags that encapsulate the logic that generates the content for the page. The application logic can reside in server-based resources (such as JavaBeans component architecture) that the page accesses with these tags. Any and all formatting (HTML or XML) tags are passed directly back to the response page. By separating the page logic from its design and display and supporting a reusable component-based design, JSP technology makes it faster and easier than ever to build Web-based applications.

JavaServer Pages technology is an extension of the Java Servlet technology. Servlets are platform-independent, server-side modules that fit seamlessly into a Web server framework and can be used to extend the capabilities of a Web server with minimal overhead, maintenance, and support. Unlike other scripting languages, servlets involve no platform-specific consideration or modifications; they are application components that are downloaded, on demand, to the part of the system that needs them. Together, JSP technology and servlets provide an attractive alternative to other types of dynamic Web scripting/programming by offering: platform independence; enhanced performance; separation of logic from display; ease of administration; extensibility into the enterprise; and, most importantly, ease of use.

JavaServer Faces technology

JavaServer Faces technology includes:

- A set of APIs for representing UI components and managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility.
- A JavaServer Pages (JSP) custom tag library for expressing a JavaServer Faces interface within a JSP page.

Designed to be flexible, JavaServer Faces technology leverages existing, standard UI and web-tier concepts without limiting developers to a particular mark-up language, protocol, or client device. The UI component classes included with JavaServer Faces technology encapsulate the component functionality, not the client-specific presentation, thus enabling JavaServer Faces UI components to be rendered to various client devices. By combining the UI component functionality with custom renderers, that define rendering, attributes for a specific UI component, developers can construct custom tags to a particular client device. As a convenience, JavaServer Faces technology provides a custom renderer and a JSP custom tag library for rendering to an HTML client, allowing developers of Java Platform, Enterprise Edition (Java EE) applications to use JavaServer Faces technology in their applications.

Ease-of-use being the primary goal, the JavaServer Faces architecture clearly defines a separation between application logic and presentation while making it easy to connect the presentation layer to the application code. This design enables each member of a web application development team to focus on his or her piece of the development process, and it also provides a simple programming model to link the pieces together. For example, web page developers with no programming expertise can use JavaServer Faces UI component tags to link to application code from within a web page without writing any scripts.

Main Technologies

This thesis is focused on overview of three JavaEE technologies. The goal is to provide reader with sufficient understanding of these technologies to be able to write their own code that successfully uses them and also realizes when and why is their use desirable.

All examples demonstrating functionality of following technologies can be found at: *www.vacuunapps.com/EJB-TUTORIAL*

Technologies of focus of this thesis are:

- ***CDI – Context and Dependency Injection***
  Contexts and Dependency Injection (CDI) for the Java EE platform is one of several features that help to knit together the web tier and the transactional tier of the Java EE platform. CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with JavaServer Faces technology in web applications. Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate various kinds of components in a loosely coupled but typesafe way.

- ***EJB3 – Enterprise JavaBeans***
  An enterprise bean is a server-side component that encapsulates the business logic of an application. Enterprise beans simplify the development of large, distributed applications. The EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services, such as transaction management and security authorization. The beans rather than the clients contain the application's business logic. The client developer can focus on the presentation of the client and does not have to code the routines that implement business rules or access databases. As a result, the clients are lighter, a benefit that is particularly important for clients that run on small devices. Enterprise beans are portable components.

- ***Transactions***

Typical enterprise application accesses and stores information in several databases. This information is critical for business operations and it must be accurate, current, and reliable. Data integrity would be lost if multiple programs were allowed to update the same information simultaneously or if a system that failed while processing a business transaction were to leave the affected data only partially updated. By preventing both of these scenarios, software transactions ensure data integrity. Transactions control the concurrent access of data by multiple programs. In the event of a system failure, transactions make sure that after recovery, the data will be in a consistent state.

Chapter Four

Context Dependency Injection (CDI)

What CDI stands for?

- **Contexts**: The ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts
- **Dependency injection**: The ability to inject components into an application in a typesafe way, including the ability to choose at deployment time which implementation of a particular interface to inject

What is Java Bean?

To explain the idea behind CDI, first we need to understand what Java Bean is. Bean is pretty much exactly what it sounds like to be. Only in terms of Java, it is a container environment.

Prior to Java EE 6, there was no clear definition of the term "bean" in the Java EE platform. Of course, Java classes used in web and enterprise applications have been called "beans" for years. There were even a couple of different kinds of things called "beans" in EE specifications, including EJB beans and JSF managed beans. Meanwhile, other third-party frameworks such as Spring and Seam introduced their own ideas of what it meant to be a "bean". What has been missing is a common definition.

Java EE 6 finally lays down that common definition in the Managed Beans specification. Managed Beans are defined as container-managed objects with minimal programming restrictions, otherwise known by the acronym POJO (Plain Old Java Object). They support a small set of basic services, such as resource injection, lifecycle callbacks and interceptors. Companion specifications, such as EJB and CDI, build on this basic model. With very few exceptions, almost every concrete Java class that has a constructor with no parameters (or a constructor designated with the annotation @Inject) is a bean. This includes every JavaBean and every EJB session bean.

All of the JavaBeans and EJBs you have been writing so far can take advantage of CDI—allowing the container to create and destroy instances of your beans and associate them with a designated context, injecting them into other beans, using them in EL expressions, specializing them with qualifier annotations, even adding interceptors and decorators to them—without modifying your existing code. At most, you'll need to add some annotations.

More specifically, a bean has the following attributes:

- A (nonempty) set of bean types
- A (nonempty) set of qualifiers
- A scope
- A set of interceptor bindings

- A bean implementation
- *Optionally*, a bean EL name

A bean type defines a client-visible type of the bean

- An interface, a concrete class, or an abstract class and may be declared final or have final methods.
- Parameterized type with type parameters and type variables.
- An array type. Two array types are considered identical only if the element type is identical.
- Primitive type.
- Raw type.

CDI Managed Beans

A managed bean is implemented by a Java class, which is called its bean class. A top-level Java class is a managed bean if it is defined to be a managed bean by any other Java EE technology specification, such as the JavaServer Faces technology specification, or if it meets all the following conditions:

- It is not a non-static inner class.
- It is a concrete class or is annotated `@Decorator`.
- It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in *ejb-jar.xml*.
- It has an appropriate constructor. That is, one of the following is the case:
    - The class has a constructor with no parameters.
    - The class declares a constructor annotated *@Inject*.

Beans as Injectable Objects

The concept of injection has been part of Java technology for some time. Since the Java EE 5 platform was introduced, annotations have made it possible to inject resources and some other kinds of objects into container-managed objects. CDI makes it possible to inject more kinds of objects and to inject them into objects that are not container-managed.

The following object types can be injected:

- Almost any Java class
- Session beans
- Java EE resources: data sources, queues, connection factories, etc.
- Persistence contexts (JPA *EntityManager* objects)
- Producer fields
- Objects returned by producer methods
- Web service references
- Remote enterprise bean references

Injections, Qualifiers and Alternatives

Let's our first application using CDI, be a simple translator. It will be capable to greet us in several languages (English and Spanish) and will demonstrate use of injection, qualifiers, alternatives and ability to choose injected bean during runtime.
The source files can be located in the folder named CDI-Translator.

First, let's create bean that will be used for backing entire process of translation and will use all objects injected by CDI.

```
@Named("translation")
public class TranslationBackingBean {
    @Inject
    @Spanish
    private TranslateService spanishTranslateService;

    @Inject
    @English
    private TranslateService englishTranslateService;

    public String getSpanishHello() {
        return spanishTranslateService.hello();
    }

    public String getEnglishHello() {
        return englishTranslateService.hello();
    }
}
```

At first look it may appear a bit confusing, but I am sure you will be on top of it in a second.

There are three unique types of annotation used here:

- *@Named* is used to reference a bean in non-Java code that supports Unified EL expressions. It is used by JSF, JSP and others.
- *@Inject* uses field injection to obtain some version of bean implementing *TanslateService* interface.
- *@Spanish* & *@English* are qualifiers specifying, which bean implementing *TanslateService* interface to choose.

Dependency injection always occurs when the bean instance is first instantiated by the container. Things happen in this order:

- First, the container calls the bean constructor (the default constructor or the one annotated *@Inject*), to obtain an instance of the bean.
- Next, the container initializes the values of all injected fields of the bean.
- Next, the container calls all initializer methods of bean (the call order is not portable, don't rely on it).
- Finally, the *@PostConstruct* method, if any, is called.

Next stop is *TranslateService* interface.

```
public interface TranslateService {
    String hello();
}
```

Two language specific beans are *EnglishTranslateService* and *SpanishTranslateService*

```
@Spanish
public class SpanishTranslateService implements TranslateService {
    @Override
    public String hello() {
        return "Hola";
    }
}
```

It's probably unnecessary to show how *EnglishTranslateService* looks like. If you have any doubts feel free to check it in attached source files.
Qualifier *@Spanish* acts as identification for injection to *TranslationBackingBean.*

Creating qualifier is very simple and doesn't take more than few seconds.

```
@Qualifier
@Target({TYPE, METHOD, FIELD, PARAMETER})
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Spanish {}
```

You can use qualifiers to provide various implementations of a particular bean type. A qualifier is an annotation that you apply to a bean. A qualifier type is a Java annotation defined as *@Target ({METHOD, FIELD, PARAMETER, TYPE})* and *@Retention (RUNTIME).*
Not all beans must have qualifier. If bean doesn't have one it is automatically considered to have qualifier *@Default*. The same is true for injection point in *TranslationBackingBean*.

At this point we have successfully created working translator, so let's make a few improvements.

Alternatives

If we wanted created another *SpanishTranslateService* there would be certain issues.
For Example:

```
@Spanish
public class SpanishTranslateService_2 implements TranslateService {
    @Override
    public String hello() {
        return "Hola Hej :)";
    }
}
```

Translator would crash on startup, because of *Ambiguity Error*. Put simply, the container has no way of knowing which of the two Spanish translators to choose, because in terms of qualifiers and type they are the same.

Alternatives allow us to pick at deployment time which version the container is supposed to pick. There are two steps to implement them:

- All but one bean must have *@Alternative* qualifiers. The one without it will be chosen by default.
- In *bean.xml* we have to pick which alternative is the container supposed to choose. If no alternative is picked, default bean will be chosen.

File *bean.xml* should be placed in META-INF or WEB-INF folder of application.

The result would look like this

```
package languageInjections;
@Spanish
@Alternative
public class SpanishTranslateService_2 implements TranslateService {...}
```

```
<?xml version="1.0"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
 <alternatives>
  <class>languageInjections.SpanishTranslateService_2</class>
 </alternatives>
</beans>
```

Injection during runtime

While Alternatives can be very useful, its dependency on redeployment to take effect, limits its use in programming itself. There is however a way to pick different beans using single interface through qualifiers.

Let’s add another *TranslateService* into the *TranslationBackingBean*. Language of this one will be able to change during runtime.

```
@Named("translation")
public class TranslationBackingBean {

      private String choice;

...

    @Inject
    @Any
    private Instance<TranslateService> unknownTranslateService;

...

    public String getUnknownHello() {
      Annotation qualifier = choice.equals("ENG") ?
                   new EnglishTranslateService() : new SpanishTranslateService();
      TranslateService TS;
      TS = unknownTranslateService.select(qualifier).get();
      return TS.hello();
    }
}
```

As you can see we are using variable *choice* to store information about our choice of translator.

Instead of *@English* or *@Spanish* we are using qualifier *@Any* which allows us to choose from all beans implementing *TranslateService* interface. If we had qualifier that is used by several specific beans we could use that one instead of *@Any* and we would be able to get an instance of those beans.

Also you have probably notice that instead of *TranslateService* we are using `Instance<TranslateService>`. This is necessary because Instance interface is the thing that allows us to dynamically obtain instances of beans with specified combination of type and qualifiers. We have to add *AnnotationLiteral* to our beans as well.

```
@Spanish
public class SpanishTranslateService extends AnnotationLiteral<Spanish> implements
TranslateService {
    @Override
    public String hello() {
        return "Hola";
    }
}
```

Scope and Producer methods

Next example demonstrates uses of *Scope* and *Producer methods* which are continuation of last theme Injection during runtime. The source files can be located in the folder named CDI-GuessNumber.

Producer methods

Not everything that needs to be injected can be boiled down to a bean class instantiated by the container using new. There are plenty of cases where we need additional control. What if we need to decide at runtime which implementation of a type to instantiate and inject or we need to inject an object that is obtained by querying a service or transactional resource, for example by executing a JPA query? However unlike in the last example, instead of obtaining right bean trough qualifiers we can directly create bean of our liking and send it out to be injected as a product of producer method.

A producer method is a method that acts as a source of bean instances. The method declaration itself describes the bean and the container invokes the method to obtain an instance of the bean when no instance exists in the specified context. A producer method lets the application take full control of the bean instantiation process. A producer method is declared by annotating a method of a bean class with the *@Produces* annotation.

We can't write a bean class that is itself a random number. But we can certainly write a method that returns a random number. By making the method a producer method, we allow the return value of the method—in this case an Integer-to be injected. We can even specify a qualifier, a scope-which in this case defaults to *@Dependent*. Now we can get a random number anywhere:

```
@ApplicationScoped
public class Generator implements Serializable {

   private java.util.Random random = new
java.util.Random(System.currentTimeMillis());

   private int maxNumber = 100;

   java.util.Random getRandom() {
      return random;
   }

   @Produces
   @Random
   int next() {
      return getRandom().nextInt(maxNumber - 1) + 1;
   }

   @Produces
   @MaxNumber
   int getMaxNumber() {
      return maxNumber;
   }
}
```

These two products will be injected to the following variables:

```
   @Inject
   @MaxNumber
   private int maxNumber;

   @Inject
   @Random
   Instance<Integer> randomNumber;
```

As you can see, only one of the variables is *Instance*. The reason for this is that *maxNumber* is injected only once before the first game but *randomNumber* is being injected before every new game.

```
   @PostConstruct
   public void reset() {
      ...
      this.biggest = maxNumber;
      this.number = randomNumber.get();
   }
```

There is couple of additional rules for Produce methods:

- The bean types of a producer method depend upon the method return type.
  If the return type is-
  - An **interface**, the unrestricted set of bean types contains the return type, all interfaces it extends directly or indirectly and *Object*.
  - **Primitive or** is a Java **array type**, the unrestricted set of bean types contains exactly two types: the method return type and *Object*.
  - **Class**, the unrestricted set of bean types contains the return type, every superclass and all interfaces it implements directly or indirectly.

- If the producer method has method parameters, the container will look for a bean that satisfies the type and qualifiers of each parameter and pass it to the method automatically—another form of dependency injection.

There is additional method of producing instances and that is producer fields.
In this example it could be used instead of *getMaxNumber* method. To change bean so it produces variable *maxNumber* in this way all we have to do is move annotations from *getMaxNumber* method to *maxNumber* variable.

```
@Produces @MaxNumber
private int maxNumber = 100;
```

Some producer methods return objects that require explicit destruction. For example, somebody needs to close this JDBC connection:

```
@Produces Connection connect(User user) {
      return createConnection(user.getId(), user.getPassword())
}
```

Destruction can be performed by a matching *disposer method*, defined by the same class as the producer method:

```
void close(@Disposes Connection connection) {
      connection.close();
}
```

The disposer method must have at least one parameter, annotated @Disposes, with the same type and qualifiers as the producer method.

Scope

For a web application to use a bean that injects another bean class, the bean needs to be able to hold state over the duration of the user's interaction with the application. The way to define this state is to give the bean a scope. You can give an object any of the scopes described in Table 4-1 Scopes, depending on how you are using it.

Table 4.1 Scopes

Scope	**Annotation**	**Duration**
Request	@RequestScoped	A user's interaction with a web application in a single HTTP request.
Session	@SessionScoped	A user's interaction with a web application across multiple HTTP requests.
Application	@ApplicationScoped	Shared state across all users' interactions with a web application.
Dependent	@Dependent	The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).
Conversation	@ConversationScoped	A user's interaction with a JavaServer Faces application, within explicit developer-controlled

		boundaries that extend the scope across multiple invocations of the JavaServer Faces lifecycle. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

According to the CDI specification, a scope determines:

- When a new instance of any bean with that scope is created
- When an existing instance of any bean with that scope is destroyed
- Which injected references refer to any instance of a bean with that scope

In the example GuessNumber, Session and Application scopes are being used.

```
@ApplicationScoped
public class Generator implements Serializable {}
@SessionScoped
public class Game implements Serializable {}
```

This means that once application is deployed *Generator* bean will be initiated end will exist until the application ends. Also there will always be only one *Generator*.
On the other hand *Game* bean will be specific for every client session and will end when session is destroyed. All beans that would be called in the context of the same *HttpSession*, would see the same instance of *Game*

As you can see both beans implement Serializable interface. This is necessary for every bean with scope that isn't *Request*. Reason behind this is that server reserves the ability to save momentarily unused bean in order to free up memory and then restore it when it is needed again.

It is possible to define new Scopes but that is outside of scope of this thesis.

Interceptors and Decorators

Next example demonstrates uses of Interceptors and Decorators. The source files can be located in the folder named CDI-Interceptor and CDI-Decorator.
These examples don't contain web interface. To test their functionality use *JUnitTest.*

Interceptors

Interceptor functionality is defined in the Java Interceptors specification. CDI enhances this functionality with a more sophisticated, semantic, annotation-based approach to binding interceptors to beans.

.
The Interceptors specification defines two kinds of interception points:

- Business method interception

```
@AroundInvoke
public Object Method(Object obj) { ... }
```

- Lifecycle callback interception

```
@PostConstruct // or different
public void Method(Object obj) { ... }
```

In order to implement interceptor we need to create interceptor binding to specify beans and methods that are to be intercepted.

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Log {}
```

As you can see it is very similar to qualifier. The only difference is that instead of annotation *@Qualifier* it contains annotation *@InterceptorBinding.*

Of course, without an actual *Interceptor* this binding won't do us any good so…

```
@Interceptor
@Log
public class LoggingInterceptor implements Serializable {
    @AroundInvoke
    public Object logMethodEntry(InvocationContext ctx) throws Exception {
        System.out.println("Intercepting entered method: " +
                                   ctx.getMethod().getName());
        return ctx.proceed();
    }
}
```

A interceptor is a bean that is annotated with interceptor binding annotation and *@Interceptor*. Once we are satisfied what has our interceptor done we can allow the original method to proceed using *ctx.proceed()* method.

Interceptor can work together. Multiple Interceptors can use the same interceptor binding.
By default, all interceptors are disabled. We need to enable our interceptor in the *beans.xml*.

```
<interceptors>
    <class>interceptors.LoggingInterceptor
    </class>
</interceptors>
```

Having the XML declaration solves two problems, it enables us to:

- specify a total ordering for all the interceptors in our system, ensuring deterministic behavior
- enable or disable interceptor classes at deployment time.

Last thing to do is to annotate beans and methods that are to be intercepted. Interceptors allow us to annotate either entire bean, which results in every method of this bean to be intercepted or we can pick specific methods. Annotating bean acts as creating invisible annotation for every method.

It is also possible to create Interceptor using combination of different Interceptor Binding. Method of a bean would be than intercepted if it would contain all bindings whether they would be real or “invisible”.

```
	public class Bean {
		@IB1 @IB2 public void Method() { ... }
	}
	@IB1
	public class Bean {
		@IB2 public void Method() { ... }
	}
	@IB2
	public class Bean {
		@IB1 public void Method() { ... }
	}
	@IB1 @IB2
	public class Bean {
		public void Method() { ... }
	}
```

Java EE doesn’t support annotation inheritance but new binding can by annotated by multiple old ones.

```
@OldBinding @AnotherBinding
@InterceptorBinding
@Target(TYPE)
@Retention(RUNTIME)
public @interface NewBinding {}
```

Decorators

Interceptors are a powerful way to capture and separate concerns which are orthogonal to the application (and type system). Any interceptor is able to intercept invocations of any Java type. This makes them perfect for solving technical concerns such as transaction management, security and call logging. However, by nature, interceptors are unaware of the actual semantics of the events they intercept. Thus, interceptors aren't an appropriate tool for separating business-related concerns.

The reverse is true of decorators. A decorator intercepts invocations only for a certain Java interface, and is therefore aware of all the semantics attached to that interface. Since decorators directly implement operations with business semantics, it makes them the perfect tool for modeling some kinds of business concerns. It also means that a decorator doesn't have the generality of an interceptor. Decorators aren't able to solve technical concerns that cut across many disparate types. Interceptors and decorators, though similar in many ways, are complementary.

In the CDI-Decorator example, capabilities of *decorators* are demonstrated on simple application of Calculator. *Decorators* are bound to *interface*, in this case:

```
public interface Calculator {
    public int add(int a, int b);
    public int subtract(int a, int b);
    public int multiply(int a, int b);
    public int divide(int a, int b);
    public int remainder(int a, int b);
}
```

Now let's take a look at *CalculatorDecorator*.

```
@Decorator
public abstract class CalculatorDecorator implements Calculator {
    @Inject @Delegate private Calculator calculator;

    @Override
    public int add(int a, int b) {
        System.out.println("add(a,b) DECORATED");
        return calculator.add(a, b);
    }
}
```

This is smaller version of decorator used in *Calculator*, but is also functional and sufficient for our purposes.

A decorator is a bean (can be even an abstract class) that implements the type it decorates and is annotated *@Decorator*. It doesn't have to override all methods defined by the type it implements only the methods of the decorated type that it wants to intercept.

Decorators have a special injection point, called the delegate injection point, with the same type as the beans they decorate, and the annotation *@Delegate*. There must be exactly one *delegate injection point (DIP)*, which can be a constructor parameter, initializer method parameter or injected field.

A decorator is bound to any bean which has:

- The type of the delegate injection point as a bean type, and
- All qualifiers that are declared at the delegate injection point.

DIP can also declare any number of qualifiers to further specify which types it wants to decorate. This includes annotation *@Any*. For example:

```
@Inject @Delegate @Any private Calculator calculator;
@Inject @Delegate @GraphicCalc private Calculator calculator;
```

First *DIP* will be bound to any type that implements *Calculator* interface.
Second *DIP* will be bound only with a type that also declares qualifier *@GraphicCalc*.

The decorator may invoke the delegate object, which has much the same effect as calling *InvocationContext.proceed()* from an interceptor. The main difference is that the decorator can invoke any business method on the delegate object.

As with interceptors, all decorators are by default disabled. To enable them we have to add them to *bean.xml*.

```
<decorators>
    <class>decorators.CalculatorDecorator</class>
</decorators>
```

The order of decorators in *bean.xml* defines order of their invocation. Also, it is important to remember that all interceptors are invoked before decorators. This declaration serves the same purpose for decorators that the *<interceptors>* declaration serves for interceptors

Stereotypes

There is no runnable example added for stereotypes.

In many systems, use of architectural patterns produces a set of recurring bean roles. A stereotype allows a framework developer to identify such a role and declare some common metadata for beans with that role in a central place.

The CDI specification defines a stereotype as follows:

- A stereotype encapsulates any combination of:
  - default scope
  - set of interceptor bindings.
- A stereotype may also specify that, all beans with the stereotype:
  - have defaulted bean EL names, or that
  - are alternatives.
- A bean may declare zero, one or multiple stereotypes. Stereotype annotations may be applied to a bean class or producer method or field.
- A stereotype is an annotation, annotated *@Stereotype* that packages several other annotations.

```
@RequestScoped
@OldSteretype1
@OldSteretype2
@MyInterceptorBinding
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface NewStereotype {}
```

This stereotype declares all that a stereotype can declare. It declares:

- Scope
- 2 different stereotypes – this is called stereotype stacking
- Interceptor binding
- Defaulted EL name – can also be specified
- Annotation *@Stereotype – mandatory*
- Annotation *@Retention – mandatory*
- Annotation *@Target – mandatory*

CDI uses two built-in stereotypes

- *@Interceptor*
- *@Model*

@Model is expected to be used frequently in web applications:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Model {}
```

Chapter Five

Enterprise JavaBeans 3.x – EJB3

What is EJB?

Enterprise JavaBeans (EJB) is an architecture for setting up program components, written in the Java programming language, that run in the server parts of a computer network that uses the client/server model. EJB is built on the JavaBeans technology for distributing program components to clients in a network. EJB offers enterprises the advantage of being able to control change at the server rather than having to update each individual computer with a client whenever a new program component is changed or added. EJB components have the advantage of being reusable in multiple applications. To deploy an EJB Bean or component, it must be part of a specific application, which is called a container. Like all Java-based architectures, programs can be deployed across all major operating systems, not just Windows. EJB's program components are generally known as *servlets*.

In EJB, there are two types of beans: **session beans** and **entity beans**. An entity bean is described as one that, unlike a session bean, has persistence and can retain its original behavior or state.

Session Beans

A session bean encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.
A session bean is not persistent. (That is, its data is not saved to a database.)

EJB3 defines three types of session bean:

- **Stateless**
- **Stateful**
- **Singleton**

Example EJB-SessionBeans demonstrates how to define all three types of session beans, how to inject them into controller/client and use of Local and Remote interfaces.

First let's take a look at controller used to manage these session beans.

```
@Named("greeter")
@SessionScoped
public class Greeter implements Serializable {

    private StatefulGreeterEJB greeterStateFul;
    @EJB private StatelessClockEJB clockStateless;
    @EJB private SigletonClockEJB clockSingleton;
    @EJB
    public void setGreeter(StatefulGreeterEJB g) { ... }
    public String getDate() { ... }
    public String setName(String name) { ... }
}
```

Greeter is managed bean that uses three EJB. These beans are initialized using annotation *@EJB*. This annotation can be used with field or a setter method and signifies an injection point where server can insert required bean when needed.

Now let's take a closer look at our three types of session beans.
For all of them is true that they are regular JavaBeans with added annotation *@Stateful*, *@Stateless* or *@Singleton*

Stateful Session Beans

The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client/bean session. Because the client interacts ("talks") with its bean, this state is often called the conversational state.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. When the client terminates, its session bean appears to terminate and is no longer associated with the client.

The state is retained for the duration of the client/bean session. If the client removes the bean, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends, there is no need to retain the state.

```
@Stateful
public class StatefulGreeterEJB implements LocalGreeter, RemoteGreeter{

      String name;

      public String sayHello() {
            if (!name.equals(""))
                  return "Hello " + name+" !";
            else return "";
      }

      public void setHello(String n) {
            name = n;
      }
}
```

Session beans can make use of Remote and Local interfaces, more about those later.

Stateless Session Beans

A stateless session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained.

Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. That is, the state of a stateless session bean should apply across all clients.

Because they can support multiple clients, stateless session beans can offer better scalability for applications that have to manage large numbers of clients at the same time. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.
A stateless session bean can implement a web service, but a stateful session bean cannot.

```
@Stateless
public class StatelessClockEJB {

      public String getTime() {
            DateFormat dateFormat = new SimpleDateFormat("HH:mm:ss dd/MM/yyyy");
            Date date = new Date();
            return "Stateless: Right now it is "+ dateFormat.format(date);
      }
}
```

Singleton Session Beans

A singleton session bean is instantiated once per application and exists for the lifecycle of the application. Singleton session beans are designed for circumstances in which a single enterprise bean instance is shared across and concurrently accessed by clients.

They offer similar functionality to stateless session beans but differ from them in that there is only one singleton session bean per application, as opposed to a pool of stateless session beans, any of which may respond to a client request. Like stateless session beans, singleton session beans can implement web service endpoints.

Singleton session beans maintain their state between client invocations but are not required to maintain their state across server crashes or shutdowns.

Applications that use a singleton session bean may specify that it should be instantiated upon application startup, which allows the singleton to perform initialization tasks for the application. The singleton may perform cleanup tasks on application shutdown as well, because the singleton will operate throughout the lifecycle of the application.

```
@Startup
@Singleton
public class SigletonClockEJB {
      private String appStarted;

      @PostConstruct
      private void startUp() {
            appStarted = new SimpleDateFormat("HH:mm:ss dd/MM/yyyy")
      .format(new Date());
      }

      public String getTime() {
            return "Singleton: Application started at "+appStarted;
      }
}
```

Thanks to annotation *@Startup* our singleton bean will be initialized the very moment the application has started and annotation *@PostConstruct* will invoke method *startUp()* after successful initialization of singleton bean. This way in variable *appStarted* will be stored aproximate date of application start.

Local and Remote Interface

StatefulGreeterEJB also implements two interfaces. One is the remote interface of the EJB the other is the local interface. Any session bean can implement remote and local interfaces.

Take a look at *interfaces.RemoteGreeter*. To define this as the remote interface of *StatefulGreeterEJB* you either, annotate the bean class and specify what the remote interfaces are, or you annotate each remote interface the bean class implements with *@Remote*. Similar for *interfaces.LocalGreeter* as you need to annotate the bean class with *@Local*.

```
@Remote
public interface RemoteGreeter {}
```

```
@Local
public interface LocalGreeter {}
```

By default, JBoss will use *ejbName*/local and *ejbName*/remote for the local and remote interfaces, respectively.

Entity JavaBeans

An entity bean represents a business object in a persistent storage mechanism. Some examples of business objects are customers, orders, and products. In the Application Server, the persistent storage mechanism is a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table.

Entity beans differ from session beans in several ways:

- **Persistence**
  The state of an entity bean is saved in a storage mechanism. State of persistent bean exists beyond the lifetime of the application or the Application Server. There are two types of persistence for entity beans: bean-managed and container-managed. With bean-managed persistence, the entity bean code that you write contains the calls that access the database. If your bean has container-managed persistence, the EJB container automatically generates the necessary database access calls. The code that you write for the entity bean does not include these calls. All examples in this thesis use the latter method.
- **Shared access**
  Entity beans can be shared by multiple clients. Because the clients might want to change the same data, it's important that entity beans work within transactions. Typically, the EJB container provides transaction management, although it is possible to manage it manually.

In the former method you do not have to code the transaction boundaries in the bean. The container marks the boundaries for you.

- **Primary key**

Each entity bean has a unique object identifier. The unique identifier, or primary key, enables the client to locate a particular entity bean (using method *findByPrimaryKey*). You must define an EJB QL query for every finder method not using primary key.

- **Relationships**

Like a table in a relational database, an entity bean may be related to other entity beans. For example, in a shopping application, OrderEB and CustomerEB and GoodsEB would be related because customers make orders and order consists of goods.

As you have probably realized from definition of entity beans, to test our entity beans we will need some sort of storage mechanism. In examples provided with this thesis, *in memory* data source is used.

Next three examples contain very similar versions of GuestBook application. They will introduce entity beans, embeddable classes and mapping, and defining tables.

Simple Entity Bean

First examples source code can be found in folder EJB-EntityBasic. There are three important classes:
Guest – Entity Bean
GuestDao – Stateless Session bean that act as data access object (DAO)
GuestServlet – handles web interface and uses GuestDao

Let's examine Guest entity bean first:

```
@Entity
@Table(name = "Guests")
public class Guest implements Serializable {

    // Persistent Fields:
    @Id @GeneratedValue
    Long id;
    private String name;
    private Timestamp signingDate;

    //Constructors:
    public Guest() {}

    public Guest(String name) {
        this.name = name;
        this.signingDate = new Timestamp(System.currentTimeMillis());
    }

    // String Representation:
      @Override
    public String toString() { ... }
}
```

Entity bean has to:

- Declare annotation *@Entity*

- Implement interface *Serializable*
- Define Primary key (@*Id*), that can be automatically generated (@GeneratedValue)
- Define *constructor* with zero parameters
- *optional* Can define name of its table through @Table(name = "TableName")

Every entity bean class represents a table and instances of entity bean are rows in it. Persistent fields than represent individual columns. In later example we will demonstrate that these columns can also be specified through different annotations.

```
@Stateless
public class GuestDao {

	@PersistenceContext
	private EntityManager em;

	public String addGuest(String name) {
		if (name != null && !name.equals("")) {
		Guest g = new Guest(name);
		em.persist(g);
		}
		return createInformation();
	}

	private String createInformation(){
		final List<Guest> list = em.createQuery
	("select g from Guests g").getResultList();
		String str = "<br />All Guests:<br />";
		for (Guest G : list) str += G +"<br />";
		return str;
	}
}
```

GuestDao is a session bean used for by servlet for communication with data source. To simplify data exchange GuestDao uses EntityManager. This object is injected into the GuestDao by EJB Container. EntityManager than can be used to send queries and four predefined methods:

- **Persist** - *Make an instance managed and persistent.*
  This means a new instance is created and if later changed within the same connection, the changes will be made in database as well.
- **Merge -** *Merge the state of the given entity into the current persistence context*.
  Changes are saved if instance exists, otherwise creates a new instance. If additional changes are made later they won't be persisted unless *merge* is invoked again.
- **Refresh -** *Refresh the state of the instance from the database, overwriting changes made to the entity, if any.*
- **Remove -** *Remove the entity instance.*

Embedded bean

Not all of beans that are part of enterprise application and are used by entity beans need their own tables. Embedding beans is a great way to separate part of self-sufficient code from entity bean or a useful way of marking several fields, which are often used together, for simpler handling.

In the following example embeddable bean Name is used. Unlike in last example, name of the guest will consist of separate first and last name. Source code can be found in folder EJB-EntEmb.

Let's take a look at this embeddable bean:

```
@Embeddable
public class Name implements Serializable {
	private String fname;
	private String lname;

	public Name() {}
	public Name(String fn, String ln) {
		fname = fn;
		lname = ln;
	}

	@Column(name="FIRST_NAME", nullable=true, length=128)
	public String getFname() {
		return fname;
	}

	@Column(name="LAST_NAME", nullable=true, length=128)
	public String getLname() {
		return lname;
	}

	public void setFname(String fname) {
		this.fname = fname;
	}

	public void setLname(String lname) {
		this.lname = lname;
	}

	@Override
	public String toString() {
		return fname + " " + lname;
	}
}
```

As you can see embeddable bean and entity bean are very similar. They differ in defining annotation (*@Embeddable* instead of *@Entity*) and embeddable bean doesn't need a key. They both however need *zero-argument constructor* and implement *Serializable* interface.

For the first time we encounter annotation *@Column*. It allows us to define properties of columns of table entity bean or embeddable bean are part of. In this case they separate column, which would otherwise contain bean *Name*, into two separate columns named FIRST_NAME and LAST_NAME. It also defines their length as 128 characters and allows entering of *null* as instance. Embeddable bean written in this way allows us to get first or last name directly, using query, or get instance of *Name* from *Guest.*

It is important to remember that all data contained in embeddable bean is stored in single row of table and one (or more if defined using @Column) cell. As such it is inefficient and difficult to store Collections or Arrays in these beans. These are better to be stored in separate tables that are bound together. This is a theme of next example.

There has been only sight change in Guest bean:

```
    private String name;

public Guest(String name) {
        this.name = name;
        this.signingDate = new Timestamp(System.currentTimeMillis());
    }
```

Was changed into:

```
    @Embedded private Name name;

    public Guest(String fname, String lname) {
        this.name = new Name(fname,lname);
        this.signingDate = new Timestamp(System.currentTimeMillis());
    }
```

Binding multiple entity beans

Real applications use more than single table or entity bean. Although with information on contained in this thesis so far we could easily create multiple entity beans any association between them would have to be manually managed.

An example for this theme is more advanced version of Guestbook that can store multiple *SignIn*-s for single *Guest*. We have two tables whose rows reference each other in One-to-Many and Many-to-One way. The example source code can be found in folder EJB-EntAdv. EJB3 allows us define this association using following annotations:

- One-to-One (O2O)
- One-to-Many (O2M)
- Many-to-One (M2O)
- Many-to-Many (M2M)

These together with other annotations define proper way of joining.

We differentiate between bidirectional and unidirectional associations. Only M2M has to be defined on both sides. If association is bidirectional the non-owning side must use *mappedBy* element to specify field or property of the owning side. If the field or property is part of embedded bean the dot (".") notation syntax must be used (mapped by =”guest.fname”). The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Let’s take a look at entity classes Guest and SignIn:

```
@Entity
@Table(name = "Guests")
public class Guest implements Serializable {

    @Id @Column(unique = true)
    private String name;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
mappedBy="guest")
```

```
    private Collection<SignIn> signIns;

    public Guest() {}
    public Guest(String name) {
        this.name = name;
    }

    public void addSignIn(SignIn s) {
       if (signIns == null) signIns = new ArrayList<SignIn>();
       signIns.add(s);
    }
}
```

```
@Entity
public class SignIn implements Serializable {

        @Id @GeneratedValue
        Long id;
        private Timestamp signingDate;

        @ManyToOne(optional=false)
        @JoinColumn(name = "GUEST", nullable=false)
        private Guest guest;

        public SignIn() {}
        public SignIn(Guest guest) {
                // TODO Auto-generated constructor stub
                setSigningDate(new Timestamp(System.currentTimeMillis()));
                setGuest(guest);
        }
}
```

In Guest bean annotation @OneToMany and @ManyToOne have four previously unseen elements:

- **Cascade**
  CascadeType.ALL specifies that when a *Guest* is created, any *SignIn*-s held in the *signIns* collection will be created as well (*CascadeType.PERSIST*). If the *Guest* is deleted from persistence storage, all related *SignIn*-s will be deleted (*CascadeType.REMOVE*). If a Guest instance is reattached to persistence storage, any changes to the *SignIn*-s collection will be merged with persistence storage (*CascadeType.MERGE*).
- **Fetch**
  FetchType.EAGER specifies that when the *Guest* is loaded whether or not to pre-fetch the relationship as well. If you want the *SignIn*-s to be loaded on demand, then specify *FetchType.LAZY*.
- **MappedBy**
  The *mappedBy* attribute specifies that this is a bi-directional relationship that is managed by the *guest* property on the *SignIn* entity.
- **Optional**
  Defines whether the association is optional. If set to false then a non-null relationship must always exist.

Inheritance

Last part of EJB chapter will explain inheritance of entity beans. The EJB3 specification allows defining entities that inherit from one another. The inheritance relationships can be reflected in queries as well. So, if you queried based on the base class, the query is polymorphic. Which means that query "From EntityBean" would return not only instances of EntityBean but also of all subclasses of EntityBean.

There is no runnable example for inheritance but let's demonstrate on simple ad-hoc classes.

```
@Entity @Inheritance(strategy=STRATEGY)
abstract public class Vehicle implements Serializable {
   @Id @GeneratedValue(strategy = AUTO)
   private Long id;
   private String manufacturer;
   private String type;
   private String model;
   private Integer wheels;
   private Integer doors;
   ...
}
@Entity @Inheritance(strategy=STRATEGY)
public class Car extends Vehicle {
  private Integer valves;
  ...
}
@Entity @Inheritance(strategy=STRATEGY)
public class Bus extends Vehicle {
  private Integer seats;
  ...
}
@Entity @Inheritance(strategy=STRATEGY)
public class SUV extends Car {
  private Boolean is4x4;
  ...
}
```

And this is how the tables would look for different strategies defined in persistence API:

Table 5.1 of Joined Inheritance Strategy

JOINED	
Vehicle	ID, MANUFACTURER, TYPE, MODEL, WHEELS, DOORS
Car	ID, VALVES
Bus	ID, SEATS
SUV	ID, IS4X4

Table 5.2 of Table per Class Inheritance Strategy

Table_PER_CLASS	
Vehicle	*Will not be created*
Car	ID, MANUFACTURER, TYPE, MODEL, WHEELS, DOORS, VALVES
Bus	ID, MANUFACTURER, TYPE, MODEL, WHEELS, DOORS, SEATS
SUV	ID, MANUFACTURER, TYPE, MODEL, WHEELS, DOORS, IS4X4

Table 5.3 of Single Table Inheritance Strategy

SINGLE_TABLE	
Vehicles	ID, **DTYPE**, MANUFACTURER, TYPE, MODEL, WHEELS, DOORS, VALVES, SEATS, IS4X4

When compared it's clear that the most space efficient strategy is JOINED and the least efficient strategy is SINGLE_TABLE.
On the other hand, data from tables implementing SINGLE_TABLE strategy can be retrieved fastest and by simplest queries and JOINED strategy implementing tables are slowest and require significantly more complicated queries.

Let's compare queries retrieving all stored instances from all tables:

- **JOINED**

```
SELECT * FROM vehicle RIGHT JOIN car ON vehicle.id=car.id RIGHT JOIN suv ON car.id=suv.id
SELECT * FROM vehicle RIGHT JOIN car ON vehicle.id=car.id
SELECT * FROM vehicle RIGHT JOIN bus ON vehicle.id=bus.id
```

- **TABLE_PER_CLASS**

```
SELECT * FROM suv
SELECT * FROM car
SELECT * FROM bus
```

- **SINGLE_TABLE**

```
SELECT * FROM vehicles
```

You have probably noticed that table created by SINGLE_TABLE strategy contains one additional column to differentiate between individual entity classes. It is called Discriminator Column and can be defined using *@DiscriminatorColumn* annotation. By default it is named DTYPE and uses names of entity beans as identifiers.

Chapter Six

Transactions

What is Transaction in Java EE?

Unlike CDI or EJB, there are not many people, who haven't heard the term transaction. And although the idea is the same, most people automatically think about database transactions. Part of EJB transaction can be database transaction, and often even multiple are, but EJB transactions go far beyond the datasource.

Basic properties of transactions can be summarized using the `ACID` mnemonic:

- **Atomic**
  All or nothing. If a transaction is interrupted, all previous steps within that transaction are undone.
- **Consistent**
  The state of objects and/or the state of tables within a database move from one consistent state to another consistent state.
- **Isolated**
  What happens within one transaction should not affect or be visible within another transaction.
- **Durable**
  The effects of a transaction are persistent.

There are two ways to manage transactions within your enterprise application:

- **Container-Managed Transactions**
- **Bean-Managed Transactions**

Container-Managed Transactions (CMT)

In an enterprise bean with **container-managed transaction**, the EJB container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean: session or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction. If no method of management is specified, enterprise beans use container-managed transaction demarcation by default.

Typically, the container begins a transaction right before an enterprise bean method starts and commits the transaction just before the method exits. Each method can be associated with a **single transaction**. Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When developing a bean, you can set the transaction attributes to specify which of the bean's methods are associated with transactions.

```
@Stateless
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public static class TransactionBean {

    @TransactionAttribute(TransactionAttributeType.NEVER)
    public String codeRed(String s) {
        return s;
    }

    public String codeBlue(String s) {
        return s;
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public String codeGreen(String s) {
        return s;
    }
}
```

- *codeRed* will be invoked with the attribute of *NEVER*
- *codeBlue* will be invoked with the attribute of *SUPPORTS*
- *codeGreen* will be invoked with the attribute of *REQUIRED*
- if *TransactionBean* wouldn't be annotated by *@TransactionAttribute*, *codeBlue* would not be associated with transaction

There is six transaction attributes that can be used to define how are enterprise bean methods react to being invoked inside or outside of transaction:

- **MANDATORY**
  A MANDATORY method is guaranteed to always be executed in a transaction. However, it's the caller's job to take care of supplying the transaction. If the caller attempts to invoke the method outside of a transaction, then the container will block the call and throw them an exception.
- **REQUIRED**
  A REQUIRED method is guaranteed to always be executed in a transaction. If the caller attempts to invoke the method outside of a transaction, the container will start a transaction, execute the method, and then commit the transaction.
- **REQUIRES_NEW**
  A REQUIRES_NEW method is guaranteed to always be executed in a transaction. If the caller attempts to invoke the method inside or outside of a transaction, the container will still start a transaction, execute the method, and then commit the transaction. Any transaction the caller may have in progress will be suspended before the method execution then resumed afterward.
- **NEVER**
  A NEVER method is guaranteed to never be executed in a transaction. However, it's the caller's job to ensure there is no transaction. If the caller attempts to invoke the method inside of a transaction, then the container will block the call and throw them an exception.
- **NOT_SUPPORTED**

A NOT_SUPPORTED method is guaranteed to never be executed in a transaction. If the caller attempts to invoke the method inside of a transaction, the container will suspend the caller's transaction, execute the method, and then resume the caller's transaction.

- **SUPPORTS**
  A SUPPORTS method is guaranteed to adopt the exact transactional state of the caller. These methods can be invoked by callers inside or outside of a transaction. The container will do nothing to change that state

Table 6.1 All possible outcomes of using transaction attributes

Transaction Attribute	**Client's Transaction**	**Business Method's Transaction**
Required	None	T2
Required	T1	T1
RequiresNew	None	T2
RequiresNew	T1	T2
Mandatory	None	Error
Mandatory	T1	T1
NotSupported	None	None
NotSupported	T1	None
Supports	None	None
Supports	T1	T1
Never	None	None
Never	T1	Error

In example Transactions-CMT we can see 6 of the 12 possibilities in action. The other six outcomes are either the same or invers. This example doesn't have web interface, instead it uses JUnit Test to confirm validity of enterprise beans.

In the example we are testing behavior of 3 EJBs. They are practically the same except their transaction attributes:

- *MoviesMandatory* declares *MANDATORY* attribute
- *MoviesSupports* declares *SUPPORTS* attribute
- *MoviesRequired* declares *REQUIRED* attribute

For every bean we invoke method actions twice – once within a transaction and second time out of it.

```
    private void actions(Movies m) throws Exception{
      m.addMovie(new Movie("Quentin Tarantino", "Reservoir Dogs", 1992));
        m.addMovie(new Movie("Joel Coen", "Fargo", 1996));
        m.addMovie(new Movie("Joel Coen", "The Big Lebowski", 1998));

        List<Movie> list = m.getMovies();
        assertEquals("List.size()", 3, list.size());

        for (Movie movie : list) {
            m.deleteMovie(movie);
        }
        assertEquals("MoviesMandatory.getMovies()", 0, m.getMovies().size());
    }
```

Methods used look like this:

```
    public void addMovie(Movie movie) throws Exception {
      entityManager.persist(movie);
    }

    public void deleteMovie(Movie movie) throws Exception {
      entityManager.remove(movie);
    }

    public List<Movie> getMovies() throws Exception {
      Query query = entityManager.createQuery("SELECT m from Movie as m");
      return query.getResultList();
    }
```

If methods are annotated by transaction attribute:

- ***MANDATORY***
  - Inside transaction – everything finishes without a glitch
  - Outside of transaction – methods would invoke *EJBTransactionRequiredException*
- ***SUPPORTS***
  - Inside transaction – everything finishes without a glitch.
  - Outside of transaction – methods would invoke *EJBException*, because although the transaction attribute would allow method to continue methods *persist*, *remove* and *createQuery* can't work outside of transaction.
    However if we used regular method without database connection, it would finish alright.
- ***REQUIRED***
  - Inside transaction – everything finishes without a glitch.

Outside of transaction – adding movies and creating queries would finish without any problem but removing instances with fail because instance is detached. Put simply instance would be persisted in different transaction than the one remove method is part of. Because of this we have no guarantee that the instance hasn't been changed or even deleted between now and then. To correct this, we need to use method merge or find to get a real instance.

```
public void deleteMovie(Movie movie) throws Exception {
      em.remove(em.merge(movie));
}
```

Enterprise beans that use CMT transaction attributes must not use any BMT methods.

Container-Managed Transaction Rollback

There are two ways to roll back a container-managed transaction. First, if a system exception is thrown, the container will automatically roll back the transaction. Second, by invoking the *setRollbackOnly* method of the EJBContext interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the rollback is not automatic but can be initiated by invoking *setRollbackOnly*.

Stateful Bean Transaction Synchronization

A stateful session bean using container-managed transactions can implement the *SessionSynchronization* interface to provide transaction synchronization notifications:

- **afterBegin()**
  Notifies the bean that a new transaction has started – called before the EJB delegates the business methods to the instance.
- **beforeCompletion()**
  Notifies the bean that the transaction is about to be committed. This is the last chance to invoke *setRollbackOnly*.
- **afterCompletion()**
  Notifies the bean that the transaction has completed – can be used to reset instance variables since it will always be invoked. This method has a single boolean parameter whose value is true if the transaction was committed and false if it was rolled back.

In addition, all methods on the stateful session bean must support one of the following transaction attributes: REQUIRES_NEW, MANDATORY or REQUIRED.

Bean-Managed Transactions (BMT)

In **bean-managed transaction**, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your bean difficult, you should consider using bean-managed transactions.

JTA Transactions

JTA is the abbreviation for the Java Transaction API. This API allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. The Application Server implements the transaction manager with the Java Transaction Service (JTS). But your code doesn't call the JTS methods directly. Instead, it invokes the JTA methods, which then call the lower-level JTS routines.

A JTA transaction is controlled by the Java EE transaction manager. You may want to use a JTA transaction because it can span updates to multiple databases from different vendors. A particular DBMS's transaction manager may not work with heterogeneous databases. However, the Java EE transaction manager does have one limitation: it does not support nested transactions. In other words, it cannot start a transaction for an instance until the preceding transaction has ended.
To demarcate a JTA transaction, you invoke *begin*, *commit*, and *rollback* methods of the *UserTransaction* interface

- ***UserTransaction.begin()***
  Starts a global transaction and associates a transaction with the execution thread. It throws the *NotSupportedException* when the calling thread is already associated with a transaction and the transaction manager implementation doesn't support nested transactions.
- ***UserTransaction.commit()***
  *C*ompletes the transaction. If at this point the transaction needs to be rolled back instead of being committed, the transaction manager does so and throws a *RollbackException* to indicate it.
- ***UserTransaction.rollback()***
  *U*ndoes any changes made since the start of the transaction and removes the association between the Transaction and the execution thread.

Use of these methods is demonstrated in example Transactions-BMT. Example consist of simple entity bean *Pair* (String key, String value), *TransactionServlet* that manages web interface and *Transaction* EJB. It is the last one where the magic happens.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Transaction {

    @PersistenceUnit(unitName = "primary")
    private EntityManagerFactory entityManagerFactory;
    @Inject
    private UserTransaction userTransaction;

    public String updateKeyValueDatabase(String key, String value) {
      EntityManager entityManager = entityManagerFactory.createEntityManager();
      try {
            userTransaction.begin();

            entityManager.joinTransaction();
            String result = decipherInputValues(entityManager, key, value);

            userTransaction.commit();
            return result;

        } catch (RollbackException e) {
      // We tried to commit the transaction but it has already been rolled back
            Throwable t = e.getCause();
            return t != null ? t.getMessage() :  e.getMessage();
        } catch (Exception e) { return e.getMessage(); }
        finally {
            try {
                if (userTransaction.getStatus() == Status.STATUS_ACTIVE)
                    userTransaction.rollback();
            } catch (Throwable e) { /* ignore */ }
            entityManager.close();
        }
    }
```

We have quite a few previously unused annotations and methods.
Annotation *TransactionManagment* is used to declare whether it is container or bean managed bean. If not used it is by default assumed it is container managed bean.

Unlike in other examples that use database connection we don't request *EntityManager* directly, because it is not thread safe, but request *EntityManagerFactory* that can create one for us. Since our newly created *EntityManager* isn't managed by container we must explicitly tell it to join the transaction. This way once the transaction commits *EntityManager* will detach its entities.

In truth this is redundant because session bean is guaranteed to be thread safe, and would be automatically joined with transaction, but would be necessary if we removed the *@Stateless* annotation and wanted it to stay thread safe.

EJB is injecting *UserTransaction* so it can manage its transaction. We use it to begin and commit a transaction and also rollback transaction if specific key ("R") is used to demonstrate its capabilities.

```
        entityManager.merge(pair);
        if (key.equals("R"))
                    try {  userTransaction.setRollbackOnly(); }
                    catch (Exception e) { /* ignore */ }
```

We are catching *RollbackException* in case transaction was rolled back. We are also catching universal *Exception* in case begin or commit throw any other exception because application can't handle them.

In the *finally* block we are making sure transaction is finished otherwise we use *rollback.* This is necessary because EntityManager has transaction scope and won’t detach the entity until transaction is finished, even if we call e*ntityManager.close().*

Returning without Committing

In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction.

In a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each business method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction.

In a stateful session bean with a transaction, the JDBC connection retains the association between the bean instance and the transaction across multiple calls. If the connection is closed, the association is lost as well.

Transaction Isolation Levels

The transaction isolation level determines how isolated one transaction is from another for read purposes only. These isolation levels are defined in terms of three phenomena that must be prevented between concurrently executing transactions.

- **Dirty reads**
  A transaction reads data written by another transaction that hasn't been committed yet.
  In other words, a transaction reads a database row containing uncommitted changes from a second transaction.
- **Non-repeatable reads**
  A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data.
  In other words, one transaction reads a row in a table, a second transaction changes the same row and the first transaction rereads the row and gets a different value.
- **Phantom reads**
  A transaction re-executes a query, returning a set of rows that satisfies a search condition, and finds that another committed transaction has inserted additional rows that satisfy the condition.
  In other words, one transaction reads all rows that satisfy a SQL WHERE condition and a second transaction inserts a row that also satisfies the WHERE condition. The first transaction applies the same WHERE condition and gets the row inserted by the second transaction.

Isolation levels aren't new to EJBs; EJB defines these levels based on the ANSI-SQL92 standards. They're mapped in JDBC to the static variables defined in the *Connection* interface.

Isolation level, like attributes, can be fine-tuned by specifying them at the method level for EJBs; however, all methods invoked in the same transaction must have the same isolation level.

There are four isolation levels and their attitude to three mentioned phenomena is demonstrated in the following table.

Table 6.2 Isolation levels in EJB transactions

Isolation level	*DR* may occur	*NRR* may occur	PR may occur
TRANSACTION_READ_UNCOMMITTED (no isolation)	Yes	Yes	Yes
TRANSACTION_READ_COMMITTED (partial isolation)	No	Yes	Yes
TRANSACTION_REPEATABLE_READ (partial isolation)	No	No	Yes
TRANSACTION_SERIALIZABLE (full isolation)	No	No	No

Although the TRANSACTION-SERIALIZABLE attribute guarantees the highest level of data integrity, it is offset by a performance slag because even simple reads must wait in line. EJBs that need to handle a large number of concurrent transactions should avoid this level.

Conclusion

This thesis deals with core technologies of Java EE and is meant to provide reader who is already versed with programming in Java SE, with knowledge necessary to combine, and incorporate these technologies into his project.

Technologies of focus in this thesis are Context and Dependency Injection, Enterprise JavaBeans and Transactions. Even as separate entities these technologies are very broad subjects that couldn’t possibly be fully processed in the scope of single bachelor thesis, instead, goal of this work is to overview big part of their basic features and to show how they can be set-up in a single project, so they would effectively interact with each other.

I hope reader will obtain sufficient sense of capabilities and advantages of this technologies as well as knowledge base to, with ease, delve deeper into the possibilities of Java EE.

In the future I hope to further explore Java EE and obtain better understanding of inner workings of this platform and its technologies.

Bibliography

Java EE. ORACLE. *Http://www.oracle.com/* [online]. [2013-05-31]. Available at: http://www.oracle.com/technetwork/articles/javaee/javaee6overview-141808.html

Java EE - Tutorial. ORACLE. *Http://www.oracle.com/* [online]. [2013-05-31]. Available at: http://docs.oracle.com/javaee/6/tutorial/doc/

Transactions. JAVA BOOT. *Http://java.boot.by/* [online]. [2013-05-31]. Available at: http://java.boot.by/ibm-287/ch05.html

Java Persistence API. ORACLE. *Http://www.oracle.com/* [online]. [2013-05-31]. Available at: http://docs.oracle.com/cd/E12839_01/apirefs.1111/e13946/ejb3_overview.html

Java Persistence API. JAVAWORLD. *Http://www.javaworld.com/* [online]. [2013-05-31]. Available at: http://www.javaworld.com/javaworld/jw-01-2008/jw-01-jpa1.html

EJB Transactions. JAVA ENTERPRISE STUFF. *Http://entjavastuff.blogspot.sk/* [online]. [2013-05-31]. Available at: http://entjavastuff.blogspot.sk/2011/02/ejb-transaction-management-going-deeper.html

EJB Tutorial. JBOSS. *Http://www.jboss.org/* [online]. [2013-05-31]. Available at: http://docs.jboss.org/ejb3/docs/tutorial/1.0.7/html/

CDI Tutorial. JBOSS. *Http://www.jboss.org/* [online]. [2013-05-31]. Available at: http://docs.jboss.org/weld/reference/latest/en-US/html/index.html

EJB Applications. IBM. *Http://www.ibm.com/* [online]. [2013-05-31]. Available at: http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.nd.iseries.doc%2Fae%2Fcejb_emcontainer.html

Java EE vs. SE. ORACLE. *Http://www.oracle.com/* [online]. [2013-05-31]. Available at: http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html

Transactions in EJB. ORACLE. *Http://www.oracle.com/* [online]. [2013-05-31]. Available at: http://docs.oracle.com/cd/E11035_01/wls100/jta/trxejb.html