

Oct 24, 2011 07:32:40 AM

RICH INTERNET APPLICATION

As a rule of thumb, we can say the following:

“If a web application uses JavaScript to load data asynchronously i.e. AJAX, it is a rich web/Internet application”--Fabian Lange

JBoss SEAM

Seam is a powerful open source development platform for building rich Internet applications in Java. Seam integrates technologies such as Asynchronous JavaScript and XML (AJAX), Java Server Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) and Business Process Management (BPM) into a unified full-stack solution, complete with sophisticated tooling.

Seam has been designed from the ground up to eliminate complexity at both architecture and API levels. It enables developers to assemble complex web applications using simple annotated Java classes, a rich set of UI components, and very little XML. Seam's unique support for conversations and declarative state management can introduce a more sophisticated user experience while at the same time eliminating common bugs found in traditional web applications.

---From seamframework.org/home

Nov 6, 2011 09:10:57 AM

The Container

“A Container is a host that provides services to guest applications”

The intent here is to provide generic services upon which applications may rely. The service support desired is usually defined by some combination of **user code and meta data that together follow a contract for interaction between the application and container**. In the case of Enterprise Java Beans (EJB) 3.1, this contract is provided by a document jointly developed by experts under the authority of the Java Community Process (<http://jcp.org>). Its job is to do all the work you shouldn't be doing.

The Enterprise Java Beans 3.1 Specification

Just as interfaces in code abstract the “what” from the “how,” the EJB Specification dictates the capabilities required of a compliant EJB Container. This 626-page document is the result of lessons learned in the field, requests by the community, and subsequent debate by the JSR-318 Expert Group (<http://jcp.org/en/jsr/detail?id=318>).

EJB defined

The Specification defines itself (EJB 3.1 Specification, page 29):

“The Enterprise Java Beans architecture is a [sic] architecture for the development and deployment of component-based business applications. Applications written using the Enterprise Java Beans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise Java Beans specification”

More simply rewritten:

“Enterprise Java Beans is a standard server-side component model for distributed business applications”

Java EE 6

*"The Java EE 6 platform offers developers the ability to write **distributed, transactional and portable** applications quickly and easily"*

Nov 9, 11 05:44:50 PM

Enterprise Applications

*"We class applications that require these capabilities (mentioned above) as "**enterprise applications**". These applications must be **fast, secure and reliable**"*

Java EE has always offered strong messaging (**JMS**), transactional (**JTA**) and resource (**JCA**) capabilities as well as exposing web services via **SOAP (JAX-WS)**. Java EE 5 started a radical shift for the programming model, offering a powerful, declarative and lightweight object-relational mapper (**JPA**) and **annotation-driven, lightweight access to enterprise services (EJB 3)**. Java EE 6 added a type-safe, loosely coupled programming model (**CDI**), declarative validation of constraints (**Bean Validation**) and **RESTful** web services (**JAX-RS**) to produce a complete, modern development environment.

Transaction Processing

In computer science, **transaction processing** is information processing that is divided into **individual, indivisible operations**, called **transactions**. Each transaction must succeed or fail as a complete unit; it cannot remain in an intermediate state. **Transaction mandatorily requires acknowledgment to get received** as a necessary feedback for accomplishment.

Transaction processing allows multiple individual operations to be linked together automatically as a single, indivisible transaction. The transaction-processing system ensures that either all operations in a transaction are completed without error, or none of them are. If some of the operations are completed but errors occur when the others are attempted, the transaction-processing system "**rolls back**" all of the operations of the transaction (including the successful ones), thereby erasing all traces of the transaction and restoring the system to the consistent, known state that it was in before processing of the transaction began. If all operations of a transaction are completed successfully, the transaction is **committed** by the system, and all changes to the database are made permanent; the transaction cannot be rolled back once this is done.

Transaction processing guards against hardware and software errors that might leave a transaction partially completed, with the system left in an unknown, inconsistent state. If the computer system crashes in the middle of a transaction, the transaction processing system guarantees that all operations in any *uncommitted* (i.e., not completely processed) transactions are canceled.

11:54:10 AM Jan 4, 2012

The Study Of

Weld - JSR-299 Reference Implementation

JSR-299: The new Java standard for dependency injection and contextual life-cycle management.

An Overview of CDI

CDI is a new specification in Java EE 6, inspired by **JBoss Seam** and **Google Guice**, and also drawing on lessons learned from frameworks such as **Spring**. It allows application developers to concentrate on developing their application logic by providing the ability to wire services together,

and abstract out orthogonal concerns, all in a type safe manner.

The most fundamental services provided by CDI are as follows:

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

In addition, CDI provides the following services:

- Integration with the Expression Language (EL), which allows any component to be used directly within a Java Server Faces page or a Java Server Pages page
- The ability to decorate injected components
- The ability to associate interceptors with components using type safe interceptor bindings
- An event-notification model
- A web conversation scope in addition to the three standard scopes (request, session, and application) defined by the Java Servlet specification
- A complete Service Provider Interface (SPI) that allows third-party frameworks to integrate cleanly in the Java EE 6 environment

A major theme of CDI is loose coupling. CDI does the following:

- Decouples the server and the client by means of well-defined types and qualifiers, so that the server implementation may vary
- Decouples the life cycles of collaborating components by doing the following:
 - Making components contextual, with automatic life cycle management
 - Allowing stateful components to interact like services, purely by message passing
- Completely decouples message producers from consumers, by means of events
- Decouples orthogonal concerns by means of Java EE interceptors

Along with loose coupling, CDI provides strong typing by

- Eliminating look up; using string-based names for wiring and correlations, so that the compiler will detect typing errors
- Allowing the use of declarative Java annotations to specify everything, largely eliminating the need for XML deployment descriptors, and making it easy to provide tools that introspect the code and understand the dependency structure at development time.

From Weld Reference Guide:

JSR-299 (CDI) defines a **set of complementary services which helps in improving the structure of application code**. CDI layers an enhanced life cycle and interaction model over the existing java component types, which includes **Managed Beans & Enterprise Java Beans**. The CDI services provides:

- An improved life cycle for stateful objects, which is bound to the well defined contexts.
- A type safe approach to the dependency injection.

- Object interaction via the event-notification facility.
- A better approach to binding interceptors to objects, along with a new kind of interceptor, called a **decorator**, that is more appropriate for use in solving business problems, and
- A **SPI(Service Provider Interface)** for developing portable extensions to the container.

Producer Methods:

In CDI a producer method is a method which acts as a source of bean instances. They are useful when:

- The objects to be injected are not required to be the instances of Web Beans(CDI beans).
- The concrete type of the objects (not the class types!) to be injected may vary at run time.
- **The objects to be injected require some custom initialization i.e. initialization, that cannot be performed by the bean constructor.**

*The CDI services are a core aspect of the Java EE platform and include full support for Java EE **modularity** and the Java EE **component architecture**. But the specification does not limit the use of CDI to the Java EE environment. In the Java SE environment, the services might be provided by a standalone CDI implementation like **Weld**, or even by a container that also implements the subset of EJB defined for embedded usage by the EJB 3.1 specification. CDI is especially useful in the context of web application development, but the problems it solves are general development concerns and it is therefore applicable to a wide variety of applications.*

Sat, February 18, 2012 12:17:26 PM

Web Services

Web services provide software re usability and portability in applications that operate over the internet.

*"A web service is a software component stored on one computer that can be accessed by an application (or other software component) on another computer over a network. Web services communicate using such technologies as XML, **JSON** and **HTTP** ."*

Web Services Basics

- **The system on which a web service is hosted is called web service host.**
- In **JAVA** a Web Service is implemented as a **Class** that resides on the server.
- Making a web service available for the clients over the internet is called **publishing** a web service.
- Using a web service from a client application is called **consuming** a web service.

Benefits Of Using Web Services

- The client application sends a request over a network to the **web service host**, which processes the request and returns a response over the network to the application.
- This kind of distributed computing benefits systems in various ways. For example, an application without direct access to data on another system might be able to retrieve the data via a web service.
- Similarly, an application lacking the processing power to perform specific computations could use a web service to take advantage of another system's superior resources.

REST(Representational State Transfer)

- REST---A network architecture that uses the web's traditional request/response mechanisms such as **GET** & **POST**.

February 19, 2012 03:25:05 PM 29 C

- Each method in a REST web service has a unique URL, therefore when a server receives a request it immediately knows which operation to perform.
- **RESTful** web services are alternatives to those implemented with **SOAP**. Unlike SOAP based web services, the request and response of REST services are not wrapped in envelopes. REST is also not limited to returning data in **XML** format. It can use a variety of formats, such as **XML**, **JSON**, **HTML**, **plain text** and **media files**.

JSON (Javascript Object Notation)

- **JSON** is an alternative to **XML** for representing the data.
- **JSON** is a text-based data-interchange format used to represent objects in JavaScript as collections of name/value pairs represented as Strings .
- **JSON** is commonly used in **Ajax** applications.
- **JSON** is a simple format that makes objects easy to read, create and parse and, because it's much less **verbose** than XML, allows programs to transmit data efficiently across the internet.
- Each **JSON** object is represented as a **list of property names and values contained in curly braces**, in the following format:
 { **Property1** : value, **Property2** : value, **Property3** : value }
- Arrays are represented in **JSON** with square brackets in the following format:
 [**value1**, **value2**, **value3**]
- Each value in an array can be a string, a number, a JSON object, true, false or null. To appreciate the simplicity of JSON data, examine this representation of an array of address book entries:

```
[ { FirstName : 'Cheryl', LastName : 'Black' },
  { FirstName : 'John', LastName : 'Smith' }, ]
```

Wed, Feb 22, 12 10:58:24 AM 22 C

How to add RichFaces 4.x to maven based project

If we generated a JAVA EE 6 project using Maven arch-types such as

jboss-javaee6-webapp-archetype then to configure RichFaces into this project we must settle the maven dependencies in the **pom.xml** file as shown below:

```
<properties>
  <org.richfaces.bom.version>4.1.0.Final</org.richfaces.bom.version>
  ...
</properties>
```

```

<dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.richfaces</groupId>
      <artifactId>richfaces-bom</artifactId>
      <version>${org.richfaces.bom.version}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

```

RichFaces Dependencies

```

<dependency>

  <groupId>org.richfaces.ui</groupId>
  <artifactId>richfaces-components-ui</artifactId>
</dependency>
<dependency>
  <groupId>org.richfaces.core</groupId>
  <artifactId>richfaces-core-impl</artifactId>
</dependency>

```

CDK annotations dependency

```

<dependency>

  <groupId>org.richfaces.cdk</groupId>
  <artifactId>annotations</artifactId>
  <scope>provided</scope>
</dependency>

```

Then every thing will be settled down spontaneously.

March 12, 2012 11:34:54 AM 25 C

How To Add Data Source In JBoss AS 7.1

There are 2 ways to add a new data source in JBoss AS 7:

1. As a deployment.
2. As a module.

We will add the data source as a **module**.

First install data base such as **MySQL**. Then install it's connector;
<http://dev.mysql.com/downloads/connector/j/> then place that connector into the

following directory: **JBoss-HOME/modules/com/mysql/main**

then create a new file **module.xml** in that same directory.

Then define the module in XML. This is the key part of the process:

If the version of the jar has changed, remember to update it here:

```
<?xml version="1.0" encoding="UTF-8"?>

<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.17-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
  </dependencies>
</module>
```

Then create a driver reference i.e. in **JBoss-HOME/standalone/configuration/standalone.xml**

file add a new driver reference as shown below:

```
<drivers>

  <driver name="mysql" module="com.mysql"/>
  <driver name="h2" module="com.h2database.h2">
    <xa-datasource-class>
      org.h2.jdbcx.JdbcDataSource

    </xa-datasource-class>
  </driver>
</drivers>
```

Then in the data source tag add a new data source as follows:

```
<datasource
  jndi-name="java:/mydb" pool-name="my_pool"
  enabled="true" jta="true"
  use-java-context="true" use-cm="true">
```

```

<connection-url>
    jdbc:mysql://localhost:3306/mydb

</connection-url>
<driver>
    mysql

</driver>
<security>
    <user-name>
        root

    </user-name>
    <password>

</password>
</security>
<statement>
    <prepared-statement-cache-size>
        100

    </prepared-statement-cache-size>
    <share-prepared-statements/>
</statement>
</datasource>

```

that is all, now start the server you should see in the console following text:

```

INFO [org.jboss.as.connector.subsystems.datasources] (MSC service thread 1-1)
Bound data source [java:/mydb]

```

Connecting JAVA EE 6 Application to a data source

The only important file is **persistence.xml**. Add there the value of **jndi-name** attribute which you added in the **standalone.xml** file under the **datasource** tag. The process is highlighted below.

```

<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```



```

xsi:schemaLocation="
    http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
<persistence-unit name="primary">
    <!-- If you are running in a production environment, add a managed
         data source, the example data source is just for proofs of concept! -->
    <jta-data-source>java:/mydb</jta-data-source>
    <!-- <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source> -->
    <properties>
        <!-- Properties for Hibernate -->
    <!--         <property name="hibernate.hbm2ddl.auto" value="create-drop" /> -->
    <!--         <property name="hibernate.show_sql" value="false" /> -->
    </properties>
</persistence-unit>
</persistence>

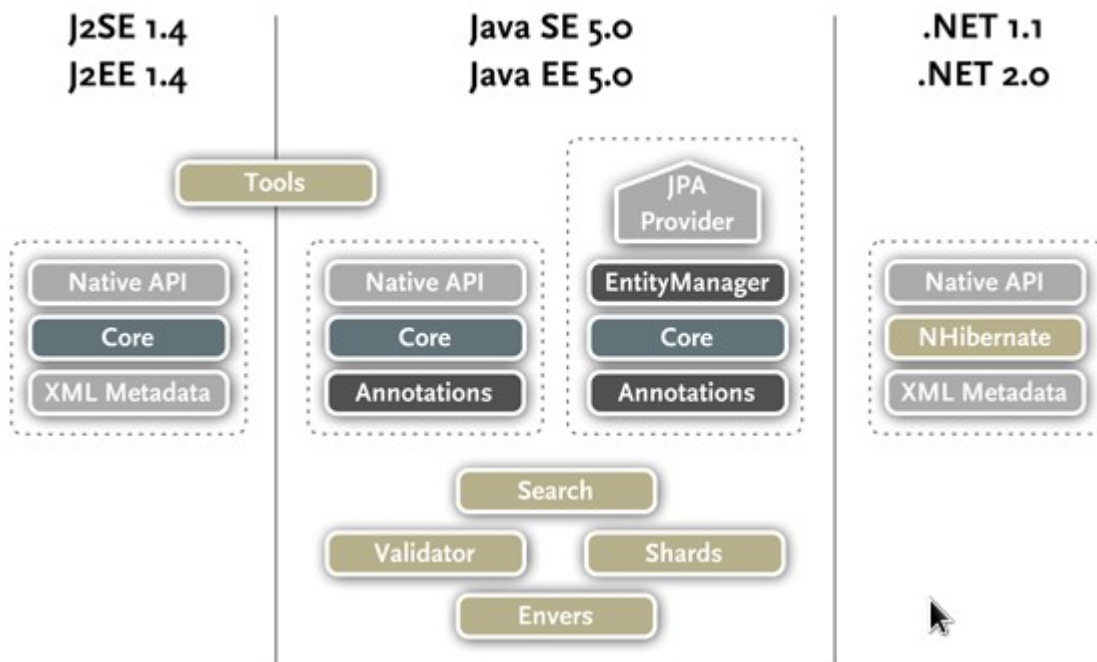
```

March 12, 2012 12:10:56 PM 25 C

Hibernate Studie

What is **Hibernate**?

Hibernate is a relational persistence for **JAVA & .NET**. Historically, Hibernate facilitated the storage and retrieval of Java domain objects via [Object/Relational Mapping](#). Today, Hibernate is a collection of related projects enabling developers to utilize POJO style domain models in their applications in ways extending well beyond Object/Relational Mapping.



History Of Hibernate:

Hibernate was started in 2001 by Gavin King as an alternative to using EJB2-style entity beans. Its mission back then was to simply offer better persistence capabilities than offered by EJB2 by simplifying the complexities and allowing for missing features.

Early in 2003, the Hibernate development team began Hibernate2 releases which offered many significant improvements over the first release and would go on to catapult Hibernate as the "de facto" standard for persistence in Java...Hibernate3, JPA, etc...

Hibernate takes care of the **mapping from Java classes to database tables**, and **from Java data types to SQL data types**. In addition, it provides data query and retrieval facilities. It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate's design goal is to relieve the developer from 95% of common data persistence-related programming tasks by eliminating the need for manual, hand-crafted data processing using SQL and JDBC. However, unlike many other persistence solutions, **Hibernate does not hide the power of native SQL queries from you** and guarantees that your investment in relational technology and knowledge is as valid as always.

Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you **to remove or encapsulate vendor-specific SQL code** and streamlines the common task of **translating result sets from a tabular representation to a graph of objects**.

What is object-relational Mapping?

Persistence

Hibernate is an Object/Relational Mapping solution for Java environments. **The term Object/Relational Mapping refers to the technique of mapping data between an object model representation to a relational data model representation.** See http://en.wikipedia.org/wiki/Object-relational_mapping for a good high-level discussion.

Hibernate is concerned with helping your application to achieve persistence. So what is persistence? Persistence simply means that we would like our application's data to outlive the applications process. In Java terms, we would like the state of (some of) our objects to live beyond the scope of the JVM so that the same state is available later.

Relational Databases

Specifically, Hibernate is concerned with data persistence as it applies to **relational databases** (RDBMS). In the world of Object-Oriented applications, there is often a discussion about using an **object database** (ODBMS) as opposed to a RDBMS. We are not going to explore that discussion here. Suffice it to say that RDBMS remain a very popular persistence mechanism and will so for the foreseeable future

The Object-Relational Impedence Mismatch

'Object-Relational Impedence Mismatch' (sometimes called the 'paradigm mismatch') is just a fancy way of saying that object models and relational models do not work very well together. RDBMSs represent data in a tabular format (a spreadsheet is a good visualization for those not familiar with RDBMSs), whereas object-oriented languages, such as Java, represent it as an interconnected graph of objects. Loading and storing graphs of objects using a tabular relational database exposes us to 5 mismatch problems...

1. Granularity

Sometimes you will have an object model which has more classes than the number of corresponding tables in the database (we say the object model is more granular than the relational model). Take for example the notion of an Address...

2. Identity

A RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (`a==b`) and object equality (`a.equals(b)`).

3. Associations

Associations are represented as unidirectional references in Object Oriented languages whereas RDBMSs use the notion of foreign keys. If you need bidirectional relationships in Java, you must define the association twice.

Likewise, you cannot determine the multiplicity of a relationship by looking at the object domain model.

4. Data Navigation

The way you access data in Java is fundamentally different than the way you do it in a relational database. In Java, you navigate from one association to an other walking the object network.

This is not an efficient way of retrieving data from a relational database. You typically want to minimize the number of SQL queries and thus load several entities via JOINS and select the targeted entities before you start walking the object network.

5. Inheritance(Subtypes)

Inheritance is a natural paradigm in object-oriented programming languages. However, RDBMSs do not define anything similar on the whole (yes some databases do have subtype support but it is completely non-standardized)...

Why Hibernate?

- Natural Programming Model:** Hibernate lets you develop persistent classes following natural Object-oriented idioms including inheritance, polymorphism, association, composition, and the Java collections framework.

- Transparent Persistence:** Hibernate requires no interfaces or base classes for persistent classes and enables any class or data structure to be persistent. Furthermore, Hibernate enables faster build procedures since it does not introduce build-time source or byte code generation or processing.

- High Performance:** Hibernate supports lazy initialization, many fetching strategies, and optimistic locking with automatic versioning and time stamping. Hibernate requires no special database tables or fields and generates much of the SQL at system initialization time instead of runtime. Hibernate consistently offers superior performance over straight JDBC coding.

- Reliability and Scalability:** Hibernate is well known for its excellent stability and quality, proven by the acceptance and use by tens of thousands of Java developers. Hibernate was designed to work in an application server cluster and deliver a highly scalable architecture. Hibernate scales well in any environment: Use it to drive your in-house Intranet that serves hundreds of users or for mission-critical applications that serve hundreds of thousands.

- Extensibility:** Hibernate is highly customizable and extensible.

•**Comprehensive Query Facilities:** Including support for Hibernate Query Language (HQL), Java Persistence Query Language (JPQL), Criteria queries, and "native SQL" queries; all of which can be scrolled and paginated to suit your exact performance needs.

March 14, 2012 11:50:58 AM 28 C

Using Hibernate as the Java Persistence API (JPA) provider

In using hibernate as a JPA provider most important thing is **persistence.xml** file. In JPA specification **persistence.xml** file plays an important role in the bootstrap process.

*"In Java™ SE environments the **persistence provider** (Hibernate in this case) is required to locate all **JPA configuration files** by **class-path look-up** of the **META-INF/persistence.xml** resource name"*



An Image Of A Sample persistence.xml File

persistence.xml files should provide a unique name for each persistence unit. Applications use this name to reference the configuration when obtaining an **javax.persistence.EntityManagerFactory** reference.

SQL UNIQUE Constraint

The UNIQUE constraint uniquely identifies each record in a database table.

The **UNIQUE** and **PRIMARY KEY** constraints both provide a guarantee for uniqueness for a column or set of columns.

A **PRIMARY KEY** constraint automatically has a **UNIQUE** constraint defined on it.

Note that you **can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table**.

```

1 package sampleJAVAEE.project.model;
2
3 import java.io.Serializable;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20 @Entity
21 @XmlRootElement
22 @Table(uniqueConstraints = @UniqueConstraint(columnNames = "email"))
23 public class Member implements Serializable {
24     /** Default value included to remove warning. Remove or modify at will. */
25     private static final long serialVersionUID = 1L;
26
27     @Id
28     @GeneratedValue
29     private Long id;
30
31     @NotNull
32     @Size(min = 1, max = 25)
33     @Pattern(regexp = "[A-Za-z ]*", message = "must contain only letters and spaces")
34     private String name;
35
36     @NotNull
37     @NotEmpty
38     @Email
39     private String email;
40
41     @NotNull
42     @Size(min = 10, max = 12)
43     @Digits(fraction = 0, integer = 12)
44     @Column(name = "phone_number")
45     private String phoneNumber;
46
47     public Long getId() {
48         return id;
49     }
50
51     public void setId(Long id) {
52         this.id = id;
53     }
54
55     public String getName() {
56         return name;
57     }
58
59     public void setName(String name) {
60         this.name = name;
61     }
62
63     public String getEmail() {
64         return email;
65     }
66
67     public void setEmail(String email) {
68         this.email = email;
69     }
70
71     public String getPhoneNumber() {
72         return phoneNumber;
73     }
74
75     public void setPhoneNumber(String phoneNumber) {
76         this.phoneNumber = phoneNumber;
77     }
78 }

```

@UniqueConstraints annotation defines the unique constraints in an entity

Marshalling

In computer science, **marshalling** (sometimes spelled **marshaling**, similar to **serialization**) is the **process of transforming the memory representation of an object to a data format suitable for storage or transmission**. It is typically used when data must be moved between different parts of a computer program or from one program to another.

Marshalling is a process that is used to communicate to remote objects with an object (in this case a **serialized object**). It simplifies complex communication, using custom/complex objects to communicate instead of **primitives**.

The opposite, or reverse, of marshalling is called *unmarshalling* (or *demarshalling*, similar to *deserialization*).

Comparison with serialization

[\[edit\]](#)

The term "marshal" is considered to be synonymous with "serialize" in the Python standard library^[3], but the terms are not synonymous in the Java-related RFC 2713^[4]:

To "marshal" an object means to record its state and **codebase(s)** in such a way that when the marshalled object is "unmarshalled", a copy of the original object is obtained, possibly by automatically loading the class definitions of the object. You can marshal any object that is serializable or remote. Marshalling is like serialization, except marshalling also records **codebases**. Marshalling is different from serialization in that marshalling treats remote objects specially. (RFC 2713^[4])

To "serialize" an object means to convert its state into a byte stream in such a way that the byte stream can be converted back into a copy of the object. The marshalling will be of more advantage.

The term **code base**, is used in [software development](#) to mean the whole collection of [source code](#) used to build a particular [application](#) or [component](#). i.e. the **source code** is called **code base**.

Example 4.2. Obtaining the javax.persistence.EntityManagerFactory

```
protected void setUp() throws Exception {
    entityManagerFactory = Persistence.createEntityManagerFactory( "org.hibernate.tutorial.jpa" );
}
```

Notice again that the persistence unit name is org.hibernate.tutorial.jpa, which matches [Example 4.1](#), "[persistence.xml](#)"

Example 4.3. Saving (persisting) entities

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
entityManager.persist( new Event( "Our very first event!", new Date() ) );
entityManager.persist( new Event( "A follow up event", new Date() ) );
entityManager.getTransaction().commit();
entityManager.close();
```

The code is similar to [Example 2.5](#), "[Saving entities](#)". An javax.persistence.EntityManager interface is used instead of a org.hibernate.Session interface. JPA calls this operation persist instead of save.

Example 4.4. Obtaining a list of entities

```
entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
List<Event> result = entityManager.createQuery( "from Event", Event.class ).getResultList();
for ( Event event : result ) {
    System.out.println( "Event ( " + event.getDate() + " ) : " + event.getTitle() );
}
entityManager.getTransaction().commit();
entityManager.close();
```

March 18, 2012 04:21:17 PM 30 C

Hibernate Persistence Contexts

Both **org.hibernate.Session API** & **javax.persistence.EntityManager API** represents a context for dealing with persistent data. This concept is called a **persistent context**. **Persistent data has a state in relation to both, the persistence context & the underlying data base.**

Much of the **org.hibernate.Session** and **javax.persistence.EntityManager** methods deal with **moving entities between the following four states**.

Entity states

- **new, or transient** - the entity has just been instantiated and is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- **managed, or persistent** - the entity has an associated identifier and is associated with a persistence context.
- **detached** - the entity has an associated identifier, but is no longer associated with a persistence context (usually because the persistence context was closed or the instance was evicted from the context)
- **removed** - the entity has an associated identifier and is associated with a persistence context, however it is scheduled for removal from the database.

Making entities persistent

Once you've created a new entity instance (using the standard `new` operator) it is in `new` state. **You can make it persistent by associating it to either a `org.hibernate.Session` or `javax.persistence.EntityManager`.**

Example 3.1. Example of making an entity persistent

```
DomesticCat fritz = new DomesticCat();
fritz.setColor( Color.GINGER );
fritz.setSex( 'M' );
fritz.setName( "Fritz" );
session.save( fritz );
```

```
DomesticCat fritz = new DomesticCat();
fritz.setColor( Color.GINGER );
fritz.setSex( 'M' );
fritz.setName( "Fritz" );
entityManager.persist( fritz );
```

org.hibernate.Session also has a method named `persist` which follows the exact semantic defined in the **JPA** specification for the **`persist`** method. **It is this method on `org.hibernate.Session` to which the `Hibernatejavax.persistence.EntityManager` implementation delegates.**

If the `DomesticCat` entity type has a generated identifier, the value is associated to the instance when the **`save`** or **`persist`** is called. If the identifier is not automatically generated, the application-assigned (usually natural) key value has to be set on the instance before **`save`** or **`persist`** is called.

Deleting entities

Entities can also be deleted.

Example 3.2. Example of deleting an entity

```
session.delete( fritz );  
entityManager.remove( fritz );
```

Obtain an entity reference without initializing its data

Sometimes referred to as **lazy loading**, the ability to obtain a reference to an entity without having to load its data is hugely important. **The most common case being the need to create an association between an entity and another, existing entity.**

Example of obtaining an entity reference without initializing its data

```
Book book = new Book();  
book.setAuthor( session.load( Author.class, authorId ) );  
  
Book book = new Book();  
book.setAuthor( entityManager.getReference( Author.class, authorId ) );
```

Obtain an entity with its data initialized

It is also quite common to want to obtain an entity along with its data, for display for example. **Example 3.4. Example of obtaining an entity reference with its data initialized**

```
session.get( Author.class, authorId );  
entityManager.find( Author.class, authorId );
```

Refresh entity state

You can reload an entity instance and its collections at any time.

Example 3.5. Example of refreshing entity state

```
Cat cat = session.get( Cat.class, catId );  
...  
session.refresh( cat );  
  
Cat cat = entityManager.find( Cat.class, catId );  
...  
entityManager.refresh( cat );
```

One case where this is useful is when it is known that the database state has changed since the data was read. Refreshing allows the current database state to be pulled into the entity instance and the persistence context.

Checking persistent state

An application can verify the state of entities and collections in relation to the persistence context.

Example 3.10. Examples of verifying managed state

```
assert session.contains( cat );
assert entityManager.contains( cat );
```

March 26, 2012 10:59:06 AM 26 C

Bean Validation

Bean Validation is a new specification in Java EE 6, inspired by **Hibernate Validator**. It allows application developers to specify constraints once (often in their domain model), and have them applied in all layers of the application, protecting data and giving useful feedback to users.

Dialects

Although SQL is relatively standardized, each database vendor uses a subset of supported syntax. This is referred to as a **dialect**. Hibernate handles variations across these dialects through its **org.hibernate.dialect.Dialect** class and the various subclasses for each vendor dialect.

Database	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
Firebird	org.hibernate.dialect.FirebirdDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Ingres	org.hibernate.dialect.IngresDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect

Database	Dialect
Oracle 8i	org.hibernate.dialect.Oracle8iDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
Progress	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase ASE 15.5	org.hibernate.dialect.SybaseASE15Dialect
Sybase ASE 15.7	org.hibernate.dialect.SybaseASE157Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect

Specifying the Dialect to use

The developer may manually specify the Dialect to use by setting the [hibernate.dialect](#) configuration property to the name of a specific ***org.hibernate.dialect.Dialect*** class to use.

March 27, 2012 06:12:28 PM 33 C

Persistent Classes

Persistent classes are classes in an application that implement the **entities** of the business problem (e.g. **Customer and Order in an E-commerce application**). The term "persistent" here means that the classes are able to be persisted, not that they are in the persistent state.

Hibernate works best if these classes follow some simple rules, also known as the **Plain Old Java Object (POJO)** programming model. However, none of these rules are hard requirements. Indeed, Hibernate assumes very little about the nature of your persistent objects. You can express a domain model in other ways (using trees of java.util.Map instances, for example).

Einer simple POJO example

There are four main rules for hibernate-persistent classes.

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
```

```
private Set kittens = new HashSet();

private void setId(Long id) {
    this.xml:id=id;
}
public Long getId() {
    return id;
}

void setBirthdate(Date date) {
    birthdate = date;
}
public Date getBirthdate() {
    return birthdate;
}

void setWeight(float weight) {
    this.weight = weight;
}
public float getWeight() {
    return weight;
}

public Color getColor() {
    return color;
}
void setColor(Color color) {
    this.color = color;
}

void setSex(char sex) {
    this.sex=sex;
}
public char getSex() {
    return sex;
}

void setLitterId(int id) {
    this.litterId = id;
}
public int getLitterId() {
    return litterId;
}

void setMother(Cat mother) {
    this.mother = mother;
}
public Cat getMother() {
```

```

        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
        kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}

```

1. Implement a no-argument constructor

Cat has a no-argument constructor. All persistent classes must have a default constructor (which can be non-public) so that Hibernate can instantiate them using **java.lang.reflect.Constructor.newInstance()**. It is recommended that this constructor be defined with at least **package visibility** in order for **runtime proxy generation** to work properly.

2. Provide an identifier property

Note

Historically this was considered option. While still not (yet) enforced, this should be considered a deprecated feature as it will be completely required to provide an identifier property in an upcoming release.

Cat has a property named **id**. **This property maps to the primary key column(s) of the underlying database table.** The type of the identifier property can be any "basic" type

Note

Identifiers do not necessarily need to identify column(s) in the database physically defined as a primary key. They should just identify columns that can be used to uniquely identify rows in the underlying table.

We recommend that you declare consistently-named identifier properties on persistent classes and that you use a nullable (i.e., non-primitive) type.

3. Prefer non-final classes (semi-optional)

A central feature of Hibernate, proxies (lazy loading), depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods. You can persist final classes that do not implement an interface with Hibernate; you will not, however, be able to use proxies for lazy association fetching which will ultimately limit your options for performance tuning. To persist a final class which does not implement a "full" interface you must disable proxy generation.

Disabling proxies in hbm.xml

```
<class name="Cat" lazy="false"...>...</class>
```

Disabling proxies in annotations

```
@Entity @Proxy(lazy=false) public class Cat { ... }
```

If the final class does implement a proper interface, you could alternatively tell Hibernate to use the interface instead when generating the proxies

Proxying an interface in **hbm.xml**

```
<class name="Cat" proxy="ICat"...>...</class>
```

Proxying an interface in annotations

```
@Entity @Proxy(proxyClass=ICat.class) public class Cat implements ICat { ... }
```

You should also avoid declaring public final methods as this will again limit the ability to generate proxies from this class. If you want to use a class with public final methods, you must explicitly disable proxying

4. Declare accessors and mutators for persistent fields (optional)

Cat declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. It is better to provide an indirection between the relational schema and internal data structures of the class. By default, Hibernate persists JavaBeans style properties and recognizes method names of the form getFoo, isFoo and setFoo. If required, you can switch to direct field access for particular properties.

Properties need not be declared public. Hibernate can persist a property declared with package, protected or private visibility as well.

April 3, 2012 11:45:05 AM 35 C

Study Of IBM Portals

Sign-On: Single sign-on (SSO) is a property of [access control](#) of multiple related, but independent software systems. With this property a user [logs in](#) once and gains access to all systems without being prompted to log in again at each of them. **Single sign-off** is the reverse property whereby a single action of signing out terminates access to multiple software systems.

As different applications and resources support different authentication mechanisms, single sign-on has to internally translate to and store different credentials compared to what is used for initial authentication.

JSR-168: is the JAVA specification for portlets, it defines portlets as Java-based Web components, managed by a portlet container, that process requests and generate dynamic content. Portals use portlets as pluggable user interface components that provide a presentation layer to information systems.

Basic Definitions

Portal: *A portal is a Web-based application that provides personalization, single sign-on, and content aggregation from different sources, and hosts the presentation layer of information systems.* Aggregation is the process of integrating content from different sources within a Webpage. A portal may have sophisticated personalization features to provide customized content to users. *Portal pages may have different sets of portlets creating content for different users.*

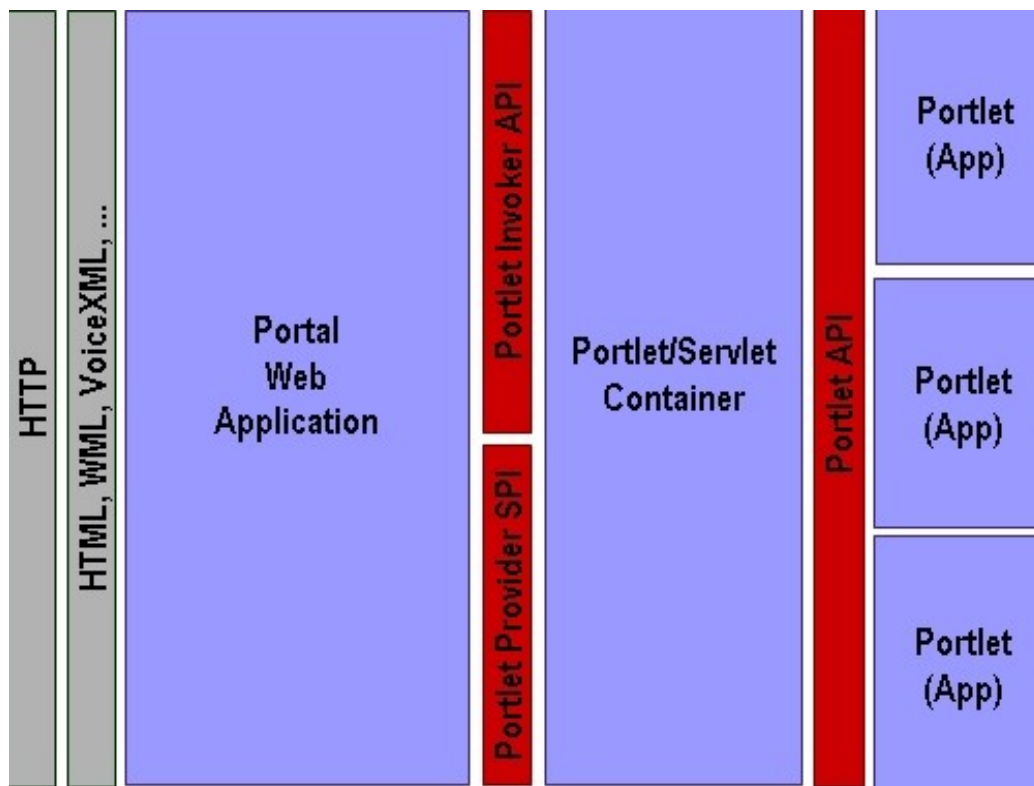
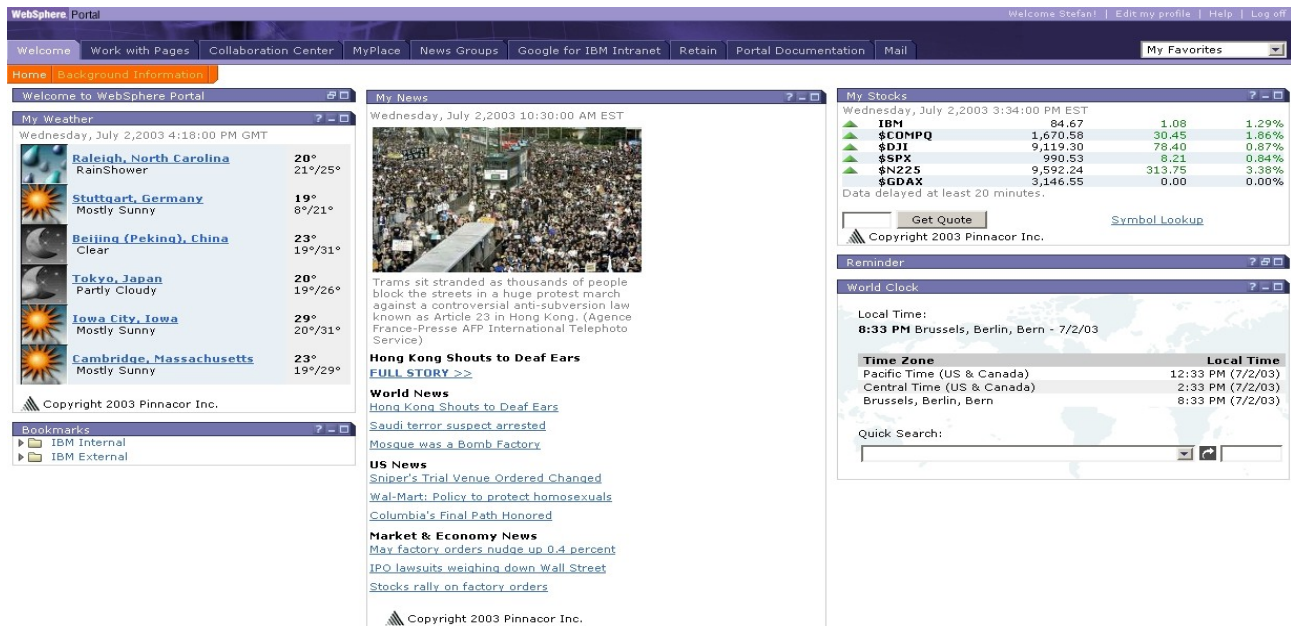


Figure above depicts a portal's basic architecture. The portal Web application processes the client request, retrieves the portlets on the user's current page, and then calls the portlet container to retrieve each portlet's content. The portlet container provides the runtime environment for the portlets and calls the portlets via the Portlet API. The portlet container is called from the portal via the Portlet Invoker API; the container retrieves information about the portal using the Portlet Provider SPI (Service Provider Interface).

Portal Page

Figure below depicts the basic portal page components. The portal page itself represents a complete markup document and aggregates several portlet windows. In addition to the portlets, the page may also consist of navigation areas and banners. A portlet window consists of a title bar with the portlet's title, decorations, and the content produced by the portlet. The decorations can include buttons to change the portlet's window state and mode (we explain these concepts later).



Portlet

As mentioned above, a **portlet is a Java-based Web component that processes requests and generates dynamic content**. The content generated by a portlet is called a **fragment**, a piece of markup (e.g., HTML, XHTML, or WML (Wireless Markup Language)) adhering to certain rules. A fragment can be aggregated with other fragments to form a complete document, as shown in Figure below portlet's content normally aggregates with the content of other portlets to form the portal page. **A portlet container manages a portlet's life cycle**.

*"In the words of Uzair Mustafa **a portlet comprises of a single war file.**"*

Portlet Container

A portlet container runs portlets and provides them with the required run-time environment. A portlet container contains portlets **and manages their life cycles**. It also **provides persistent storage mechanisms for the portlet preferences**. A portlet container receives requests from the portal to execute requests on the portlets hosted by it. **A portlet container is not responsible for aggregating the content produced by the portlets; the portal itself handles aggregation**.

A portal and a portlet container can be built together as a single component of an application suite or as two separate components of a portal application.

Sunday, June 24, 2012 04:29:25 PM 32 C

STATIC BLOCKS IN JAVA

A static block is called only once when the class is first loaded into the **JVM** no matter how many objects of the class you have created.

```
static {
//.....
}
```

we can also use the **constructor block**:

```
{
    //.....
}
```

in the above case the block will be executed whenever the instance of the class will be created and the constructor block is executed before any of the constructors of the class.

Java EE Interceptors

Interceptors are used in conjunction with Java EE managed classes to allow developers to invoke interceptor methods in conjunction with **method invocations or life-cycle events** on an associated **target class**. Common uses of interceptors are **logging, auditing, or profiling**.

Interceptors can be defined within a **target class as an interceptor method**, or in an associated class called an **interceptor class**. Interceptor classes contain methods that are invoked in conjunction with the methods or life-cycle events of the target class.

Interceptor classes and **interceptor methods** are defined using the **meta data annotations** or in the **deployment descriptors**.

Remember: Interceptors defined using the deployment descriptors are not portable across the application servers.

Interceptor methods within the target class or in an interceptor class are annotated with one of the meta data annotations defined in following table.

Interceptor Metadata Annotation	Description
<code>javax.interceptor.AroundInvoke</code>	Designates the method as an interceptor method.
<code>javax.interceptor.AroundTimeout</code>	Designates the method as a timeout interceptor, for interposing on timeout methods for enterprise bean timers.
<code>javax.annotation.PostConstruct</code>	Designates the method as an interceptor method for post-construct lifecycle events.
<code>javax.annotation.PreDestroy</code>	Designates the method as an interceptor method for pre-destroy lifecycle events.

Interceptors interposes(intervene) the execution of methods and the life cycle events of the classes; if there is a post-construct event then first the `@PostConstruct` method will execute and then comes the post-construct state of a class.

INTERCEPTOR CLASSES

Interceptor classes may be designated with the optional **`javax.interceptor.Interceptor`** annotation, **but interceptor classes aren't required to be so annotated**. Interceptor classes must have a **public, no-argument constructor**. The target class can have any number of interceptor classes associated with it. The order in which the interceptor classes are invoked is determined by the order in which the interceptor classes are defined in the `javax.interceptor.Interceptors` annotation. **This order can be overridden in the deployment descriptor.**

Interceptor classes may be targets of dependency injection. Dependency injection occurs when the interceptor class instance is created, using the naming context of the associated target class, and before any `@PostConstruct` callbacks are invoked.

Life Cycle Of An Interceptor Class

Interceptor classes have the same life-cycle as their associated target class. When a target class instance is created, an interceptor class instance is also created for each declared interceptor class in the target class. That is, if the target class declares multiple interceptor classes, an instance of each class is created when the target class instance is created. The target class instance and all interceptor class instances are fully instantiated before any **@PostConstruct** callbacks are invoked, and any **@PreDestroy** callbacks are invoked before the target class and interceptor class instances are destroyed.

USING INTERCEPTORS

Interceptors are defined using one of the interceptor meta data annotations, within the **target class**, or in a separate **interceptor class**. The following code declares an **@AroundTimeout** interceptor method within a target class.

```
@Stateless
public class TimerBean {
...
@Schedule(minute="*/1", hour="*")
public void automaticTimerMethod() { ... }
@AroundTimeout
public void timeoutInterceptorMethod(InvocationContext ctx) { ... }
...
}
```

If **interceptor classes** are used, use the **javax.interceptor.Interceptors** annotation to declare one or more interceptors at the class or method level of the target class. The following code declares interceptors at the **class level**.

```
@Stateless
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
public class OrderBean { ... }
```

i.e. the life-cycle call back events and method-level invocations of the class **OrderBean** are intercepted using methods defined in the **PrimaryInterceptor.class** and **SecondaryInterceptor.class**

The following code declares a method-level interceptor class.

```
@Stateless
public class OrderBean {
...
@Interceptors(OrderInterceptor.class)
public void placeOrder(Order order) { ... }
...
}
```

Intercepting Method Invocations

The **@AroundInvoke** annotation is used to designate interceptor methods for **managed object** methods. Only one around-invoke interceptor method per class is allowed. Around-invoke interceptor methods have the following form:

```
@AroundInvoke
<visibility> Object <Method name>(InvocationContext) throws Exception { ... }
```

For example:

@AroundInvoke

```
public void interceptOrder(InvocationContext ctx) { ... }
```

Around-invoke interceptor methods can have **public**, **private**, **protected**, or **package-level** access, and must not be declared **static** or **final**. Around-invoke interceptors can call any component or resource callable by the target method on which it interposes, *have the same security and transaction context as the target method, and run in the same Java virtual machine call-stack as the target method.*

Around-invoke interceptors can throw any exception allowed by the throws clause of the target method. They may catch and suppress exceptions, and then recover by calling the **InvocationContext.proceed** method.

Using Multiple Method Interceptors

Use the **@Interceptors** annotation to declare multiple interceptors for a target method or class.

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class, LastInterceptor.class})
```

```
public void updateInfo(String info) { ... }
```

The order of the interceptors in the **@Interceptors** annotation is the order in which the interceptors are invoked. Multiple interceptors may also be defined in the deployment descriptor. The order of the interceptors in the deployment descriptor is the order in which the interceptors will be invoked.

...

```
<interceptor-binding>
```

```
<target-name>myapp.OrderBean</target-name>
```

```
<interceptor-class>myapp.PrimaryInterceptor.class</interceptor-class>
```

```
<interceptor-class>myapp.SecondaryInterceptor.class</interceptor-class>
```

```
<interceptor-class>myapp.LastInterceptor.class</interceptor-class>
```

```
<method-name>updateInfo</method-name>
```

```
</interceptor-binding>
```

...

To explicitly pass control to the next interceptor in the chain, call the *InvocationContext.proceed* method.

Sharing Data Across Interceptors

The same **InvocationContext** instance is passed as an input parameter to each interceptor method in the interceptor chain for a particular target method. The **InvocationContext** instance's **contextData** property is used to pass data across interceptor methods. The **contextData** property is a **java.util.Map<String, Object>** object. Data stored in **contextData** is accessible to interceptor methods further down the interceptor chain. *The data stored in **contextData** is not sharable across separate target class method invocations.* That is, a different **InvocationContext** object is created for each invocation of the method in the target class.

Accessing Target Method Parameters From an Interceptor Class

The **InvocationContext** instance passed to each around-invoke method may be used to access and modify the **parameters** of the **target method**. **The parameters property of *InvocationContext* is an array of *Object* instances that corresponds to the parameter order of the target method.** For example, for the following target method:

```
@Interceptors(PrimaryInterceptor.class)
```

```
public void updateInfo(String firstName, String lastName, Date date) { ... }
```

The **parameters** property, in the **InvocationContext** instance passed to the **around-invoke interceptor** method in **PrimaryInterceptor**, is an Object array containing a String object (firstName), a String object (lastName), and a Date object (date).

The parameters can be accessed and modified using the **InvocationContext.getParameters** and **InvocationContext.setParameters** methods, respectively.

Intercepting Lifecycle Callback Events

Interceptors for life-cycle callback events (post-create and pre-destroy) may be defined in the target class or in interceptor classes. The **@PostCreate** annotation is used to designate a method as a post-create lifecycle event interceptor. The **@PreDestroy** annotation is used to designate a method as a pre-destroy lifecycle event interceptor. Lifecycle event interceptors defined within the target class have the following form:

```
void <Method name>() { ... }
```

For example:

```
@PostCreate
```

```
void initialize() { ... }
```

Lifecycle event interceptors defined in an interceptor class have the following form:

```
void <Method name>(InvocationContext) { ... }
```

For example:

```
@PreDestroy
```

```
void cleanup(InvocationContext ctx) { ... }
```

Lifecycle interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Lifecycle interceptor methods are called in an unspecified security and transaction context. That is, portable Java EE applications should not assume the lifecycle event interceptor method has access to a security or transaction context. Only one interceptor method for each lifecycle event (post-create and pre-destroy) is allowed per class.

Using Multiple Lifecycle Callback Interceptors

Multiple lifecycle interceptors may be defined for a target class by specifying the interceptor classes in the **@Interceptors** annotation:

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class, LastInterceptor.class})
```

```
@Stateless
```

```
public class OrderBean { ... }
```

The order in which the interceptor classes are listed in the **@Interceptors** annotation defines the order in which the interceptors are invoked.

Data stored in the contextData property of InvocationContext is not sharable across different lifecycle events (Like in @AroundInvoke method). But is shareable within the same life-cycle event.

The interceptor Example Application

The interceptor example demonstrates how to use an interceptor class, containing an **@AroundInvoke** interceptor method, with a stateless session bean.

The **HelloBean stateless session bean** is a simple enterprise bean with a two business methods, **getName** and **setName** to retrieve and modify a string. The **setName** business method has an **@Interceptors** annotation that specifies an interceptor class, **HelloInterceptor**, for that method.

```
@Interceptors(HelloInterceptor.class)
public void setName(String name) {
    this.name = name;
}
```

The **HelloInterceptor** class defines an **@AroundInvoke** interceptor method, **modifyGreeting**, that converts the string passed to **HelloBean.setName** to lower case.

```
@AroundInvoke
public Object modifyGreeting(InvocationContext ctx) throws Exception {
    Object[] parameters = ctx.getParameters();
    String param = (String) parameters[0];
    param = param.toLowerCase();
    parameters[0] = param;
    ctx.setParameters(parameters);
    try {
        return ctx.proceed();
    } catch (Exception e) {
        logger.warning("Error calling ctx.proceed in modifyGreeting()");
        return null;
    }
}
```

The parameters to **HelloBean.setName** are retrieved and stored in an Object array by calling the **InvocationContext.getParameters** method. Because **setName** only has one parameter, it is the first and only element in the array. The string is set to lower case, and stored in the parameters array, then passed to **InvocationContext.setParameters**. To return control to the session bean, **InvocationContext.proceed** is called.

Saturday, June 30, 2012 11:43:59 AM 31 C

TICKET MONSTER CASE STUDY

[TicketMonster](#) is an example eCommerce application that demonstrates the features of Java EE 6, [JBoss Enterprise Application Platform 6](#) or [JBoss AS7](#). [TicketMonster](#) provides a reference for you, both when you want to learn new technologies, and when you are building your own application. The application itself demonstrates the multiple view framework choices whether HTML5 + RESTful endpoints, JSF2 + [RichFaces](#) or GWT + [Errai](#). The tutorials below walk you through the steps of: setting up a development environment, using [JBoss Forge](#) & Maven, defining JPA2 entities, HTML5 mobile development, [RichFaces](#) & JSF2, and using GWT & [Errai](#).

Web Tools Project (WTP): WTP compliant application servers allows us to use an IDE for performing the tasks such as deploying an application to the application server.

Hibernate Validator

Hibernate Validator 4.x is the reference implementation for [JSR 303 - Bean Validation](#) of which Red Hat is the specification lead.

JSR 303 - Bean Validation defines a meta data model and API for JAVA Bean validation. The default meta data source is annotations, with the ability to override and extend the meta-data through the use of XML validation descriptors. **The API is not tied to a specific application tier or programming model. It is specifically not tied to either the web tier or the persistence tier, and is available for both server-side application programming, as well as rich client Swing application developer.**

Creating A new Entity

There are several ways to add a new **JPA** entity to your project:

Starting from scratch

Right-click on the .model package and select New → Class. JPA entities are annotated POJOs so starting from a simple class is a common approach.

Reverse Engineering

Right-click on the "model" package and select New → JPA Entities from Tables.

Using Forge

To create a new entity for your project using a CLI (we will explore this in more detail below) Reverse Engineering with Forge Forge has a Hibernate Tools plug-in that allows you to script the conversion of RDBMS schema into JPA entities. For more information on this technique see this video.

Discussions about **hashCode()** and **equals()** methods:

hashCode():

- **In the java, every class must provide a hashCode() method which digests the data stored in an instance of the class into a single hash value (a 32- bit signed Integer).** This hash is used by other code when storing or manipulating the instance - the values are intended to be evenly distributed for varied inputs in order to avoid clustering. This property is important to the performance of **hash tables** and other **data structures** that store objects in groups ("buckets") based on their computed hash values.
- **Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce same hash codes.**

"If in case we override the equals method inside a class then we must also override the hashCode method of that class"

@Enumerated(STRING) or @Enumerated(ORDINAL)?

JPA can store an **enum** value using it's ordinal (position in the list of declared enums) or it's **STRING** (the name it is given). If you choose to store an ordinal, you must not alter the order of the list. If you choose to store the name, you must not change the **enum** name. The choice is yours!

Natural Identity

Using an ORM introduces additional constraints on object identity. Defining the properties that make up an entity's natural identity can be tricky, but is very important. Using the object's identity, or the synthetic identity (database generated primary key) identity can introduce unexpected bugs into your application, so you should always ensure you use a natural identity. We denote the natural identity of an entity by annotating it with `@Column(unique = true)` annotation.

```
* <p>
* The url of the media item forms it's natural id and cannot be shared
  between event categories
* </p>

@URL
@Column(unique = true)
private String url;
```

Synthetic Identity

The synthetic identity if an entity is the data base generated primary key.

```
/**
 * The synthetic id of the object.
 */
@Id
@GeneratedValue(strategy = IDENTITY)
private Long id;
```

Mapping identifier properties

The `@Id` annotation lets you define which property is the identifier of your entity. This property can be set by the application itself or be generated by Hibernate (preferred). You can define the identifier generation strategy thanks to the `@GeneratedValue` annotation.

Generating the identifier property

JPA defines five types of identifier generation strategies:

- **AUTO** - either identity column, sequence or table depending on the underlying DB. It makes an application more portable.
- **TABLE** - table holding the id
- **IDENTITY** - identity column
- **SEQUENCE** - sequence
- **identity copy** - the identity is copied from another entity

Hibernate provides more id generators than the basic JPA ones.

July 15, 2012 06:49:45 PM 30 C

Relationships supported by JPA

JPA can model four types of relationship between entities - **one-to-one**, **one-to-many**, **many-to-one** and **many-to-many**. A relationship may be bi-directional (both sides of the relationship know about each other) or uni-directional (only one side knows about the relationship).

Bi-Directional Relation Ship

```
public class Show implements Serializable {
    @OrderBy("date")
    @OneToMany(fetch = EAGER, mappedBy = "show", cascade = ALL)
    private Set<Performance> performances = new HashSet<Performance>();
}
```

where 'show' and 'date' are the names of properties in the **Performance** entity

July 15, 2012 08:09:59 PM 29 C

Lazy Loading In Hibernate

Say you have a parent and that parent has a collection of children. Hibernate now can "lazy-load" the children, which means that it does not actually load all the children when loading the parent. Instead, it loads them when requested to do so. You can either request this explicitly or, and this is far more common, hibernate will load them automatically when you try to access a child.

Lazy-loading can help improve the performance significantly since often you won't need the children and so they will not be loaded.

Also beware of the n+1-problem. Hibernate will not actually load all children when you access the collection. Instead, it will load each child individually. When iterating over the collection, this causes a query for every child. In order to avoid this, you can trick hibernate into loading all children simultaneously, e.g. by calling `parent.getChildren().size()`.

Fetch EAGER in HIBERNATE (Vice Versa Of Lazy Loading)

When retrieving a show, we will also retrieve its associated performances by adding the **fetch = EAGER** attribute to the **@OneToMany** annotation. This is a design decision which required careful consideration. In general, you should favour the default lazy initialization of collections: their content should be accessible on demand. However, in this case we intend to marshal the contents of the collection and pass it across the wire in the **JAX-RS** layer, after the entity has become detached, and cannot initialize its members on demand.

```
public class Show implements Serializable {
    @OneToMany(fetch = EAGER, mappedBy = "show", cascade = ALL)
    private Set<Performance> performances = new HashSet<Performance>();
}
```

As Show is the owner of Performance (*and without a show, a performance cannot exist*), we add the **cascade = ALL** attribute to the **@OneToMany** annotation. As a result, any persistence operation that occurs on a show, will be propagated to it's performances. For example, if a show is removed, any associated performances will be removed as well.

July 31, 2012 12:06:57 PM 30 C

Cascading In JPA

"Cascade means a stream of water which emerges from a single source and further divides into several branches. (So if we close the emerging source of stream then all of the branches will spontaneously be vanished)"

"Cascade may also refer to a chain of spontaneous processes which initiates when we perform a single action"

Consider two entities **Show** and **Performance** Show has **@OneToMany** relationship with the **Performance** entity. **Show** is the owner i.e. if a **Show** does not exist then it's performances will no longer be needed to exist also. So we can mark their relationships as **cascade = ALL** so any persistence related operation (REMOVE, DETACHMENT, MERGE, PERSIST, REFRESH) performed on **Show** will be reflected to it's respected performances.

A cascade has five types:

1. REMOVE
2. REFRSH
3. DETACH
4. MERGE
5. PERSIST

July 17, 2012 11:49:14 AM 31 C

JSF Is MVC Based

JSF applications adhere to the **Model-View-Controller** (MVC) architecture, which **separates an application's data (contained in the model) from the graphical presentation (the view) and the processing logic (the controller)**. Following figure shows the relationships between components in MVC.



Model-View-Controller architecture.

In JSF, **the controller is the JSF framework** and is responsible for coordinating interactions between the view and the model. The **model contains the application's data (typically in a database)**, and the **view presents the data stored in the model (typically as web pages)**. When a user interacts with a JSF web application's view, the JSF framework (**controller**) interacts with the model to store and/or retrieve data. When the model changes, the view is updated with the changed data.

@Temporal Annotation

`javax.persistence.Temporal` annotation is used on fields & properties (*getters/setters*) of types `java.util.Date` and `java.util.Calendar`

"It may only be specified for fields or properties of these types i.e. Date & Calendar."

example usage:

```

@NotNull
@Temporal(TIMESTAMP)
private Date date;
  
```

A Java `Date` object represents both a date and a time, whilst **an RDBMS splits out Date, Time and Timestamp**. Therefore we instruct JPA to store this date as a **timestamp** using the `@Temporal(TIMESTAMP)` annotation. The date and time of the performance is required, and the Bean Validation constraint `@NotNull` enforces this.

A Java `Date` represents "a specific instance in time, with millisecond precision" and is the recommended construct for representing date and time in the JDK. A RDBMS's `DATE` type typically has day precision only, and uses the `DATETIME` or `TIMESTAMP` types to represent an instance in time, and often only to second precision.

As the mapping between Java date and time, and database date and time isn't straightforward, **JPA requires us to use the `@Temporal` annotation on any property of type `Date`, and to specify whether the `Date` should be stored as a date, a time or a timestamp (date and time).**

July 21, 2012 06:35:25 AM 29 C

Business Services Where to draw the line?

A business service is an encapsulated, reusable logical component that groups together a number of well-defined cohesive business operations. Business services perform business operations, and may coordinate infrastructure services such as persistence units, or even other business services as well. The boundaries drawn between them should take into account whether the newly created services represent, potentially reusable components.

Static Blocks

In a **JAVA** class a static block is executed only once when the class is first time loaded into the **JVM** (and when no instance of the class exists).

A static block is called is executed before the constructor of a class.

```

@Named
@RequestScoped
public class MediaManager {

    /**
     * Locate the tmp directory for the machine
     */
    private static final File tmpDir;

    static {
        tmpDir = new File(System.getProperty("java.io.tmpdir"),
  
```



```

"org.jboss.jdf.examples.ticket-monster");
    if (tmpDir.exists()) {
        if (tmpDir.isFile())
            throw new IllegalStateException(tmpDir.getAbsolutePath() + "
already exists, and is a file. Remove it.");
        else {
            tmpDir.mkdir();
        }
    }
}

```

July 22, 2012 11:20:52 AM 32 C

Study Of Generic Classes And Methods

Motivation For The Generics

- It would be nice if we could write a single sort method to sort the elements in an Integer array, a String array or an array of any type that supports ordering (i.e., its elements can be compared).
- It would also be nice if we could write a single Stack class that could be used as a Stack of integers, a Stack of floating-point numbers, a Stack of Strings or a Stack of any other type.
- It would be even nicer if we could detect type mismatches at compile time—known as compile-time type safety. For example, if a Stack stores only integers, attempting to push a String onto that Stack should issue a compile-time error.

generics, provide the means to create the general models mentioned above .

Generics in JAVA is among the most powerful capabilities for software reuse with compile-time type safety

In JAVA generics is the implementation of the DRY(Do not repeat yourself) principle.

- Only reference types can be used with the generic classes and methods.
- If the operations performed by several overloaded methods are identical for each argument type, the overloaded methods can be more compactly and conveniently coded using a generic method.
- We can write a single generic method declaration that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Generic Methods

- All generic methods declarations have a **type parameter section** delimited by the angled brackets <> and precedes the method's return type.

```

// generic method printArray
public static < T > void printArray( T[] inputArray ) {
    ...
}

```

- A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the **return type, parameter types** and **local variable types** in a generic method declaration, and they act as **placeholders for the types of the arguments** passed to the generic method, which are known as actual type arguments.
- Type parameters can represent only reference types **not primitive types** (like int, double and

char) .

- A type parameter can be declared only once in the **type-parameter section** but can appear more than once in the method's parameter list. For example, the type-parameter name **T** appears twice in the following method's parameter list:

```
// generic method printTwoArrays
public static < T > void printTwoArrays( T[] array1, T[] array2 ) {
    ...
}
```

- When declaring a generic method, failing to place a **type-parameter section** before the return type of a method is a syntax error.

Erasure at compile-time in generic methods

Consider following generic method:

```
// generic method printArray
public static < T > void printArray( T[] inputArray ) {
    ...
}
```

- When the compiler translates generic method `printArray` into Java byte codes, it removes the type parameter section and replaces the type parameters with actual types. This process is known as **erasure**.
- By default all generic types are replaced with type **Object**. So the compiled version of method `printArray` appears as shown :

```
// generic method printArray
public static void printArray( Object[] inputArray ) {
    ...
}
```

- In a generic method, the benefits become apparent when the method also uses a type parameter as the method's return type.
- We cannot use relational operators such as `'=='` or `'>'` with the reference types.

Interface java.lang.Comparable

- It is possible to compare two objects of the same class if that class implements the generic interface **Comparable<T>** .
- All the type-wrapper classes for **primitive** types implement this interface .
- For example, if we have two Integer objects, `integer1` and `integer2`, they can be compared with the expression:

```
integer1.compareTo( integer2 )
```

- It is our responsibility when you declare a class that implements **Comparable<T>** to declare method **compareTo** such that it compares the contents of two objects of that class and returns the comparison results .
- As specified in interface **Comparable<T>** documentation, **compareTo** must return 0 if the objects are equal, a **negative integer** if **object1** is less than **object2** or a **positive integer** if **object1** is greater than **object2**.

Upper Bound Of a type parameter

```
public static < T extends Comparable< T > > T maximum( T x, T y, T z ) {
    ...
}
```

In the method above:

- The type-parameter section specifies that **T** extends **Comparable<T>** interface and **only objects** of classes that implement interface **Comparable<T>** can be used with this method.
- In this case, **Comparable** is known as the **upper bound** of the type parameter.

- By default, `Object` is the upper bound.
- Type-parameter declarations that bound the parameter **always use keyword `extends`** regardless of whether the type parameter extends a class or implements an interface.
- When the compiler replaces the type-parameter information with the upper-bound type in the method declaration, it also inserts explicit cast operations in front of each method call to ensure that the returned value is of the type expected by the caller.

Overloading Generic Methods

- A generic method may be overloaded.
- A class can provide two or more generic methods that specify the same method name but different method parameters.
- A generic method can also be overloaded by non generic methods.
- When the compiler encounters a method call, it searches for the method declaration that most precisely matches the method name and the argument types specified in the call.
- The compiler tries to find and use a precise match in which the method name and argument types of the method call match those of a specific method declaration.
- If there's no such method, the compiler attempts to find a method with compatible (*such as of type `Object` or Base Classes*) types or a matching generic method.

Overriding Generic Methods

- A generic method in a subclass can override a generic method in a super class if both methods have the same signatures.

Generic Classes

- The concept of a data structure, such as a stack, can be understood independently of the element type it manipulates.
- Generic classes provide a means for describing the concept of a stack (or any other class) in a type-independent manner.

Example Of A Generic Class

```
// Stack generic class declaration.
import java.util.ArrayList;

public class Stack<T> {

    // ArrayList stores stack elements
    private ArrayList<T> elements;

    // no-argument constructor creates a stack of the default size
    public Stack() {
        this(10); // default stack size
    }

    // Constructor creates a stack of the specified number of elements
    public Stack(int capacity) {
        int initCapacity = capacity > 0 ? capacity : 10; // validate
        elements = new ArrayList<T>(initCapacity); // create ArrayList
    }

    // Push element onto stack
    public void push(T pushValue) {
        elements.add(pushValue); // place pushValue on Stack
    }

    // Return the top element if not empty; else throw EmptyStackException
    public T pop() {
        // If stack is empty
        if (elements.isEmpty())
            throw new EmptyStackException("Stack is empty, cannot pop");
    }
}
```

```

        // remove and return top element of Stack
        return elements.remove(elements.size() - 1);
    }
}

```

Raw Types:

- It's also possible to instantiate generic class Stack without specifying a type argument, as follows:

```

Stack objectStack = new Stack( 5 ); // no type-argument specified

Stack rawTypeStack2 = new Stack< Double >( 5 );

Stack< Integer > integerStack = new Stack( 10 );

```

Although the third assignment is permitted, it's unsafe, because a Stack of raw type might store types other than Integer. In this case, the compiler issues a warning message which indicates the unsafe assignment.

Wild Cards

Wild cards enable you to specify method **parameters**, **return values**, **variables** or **fields**, and so on, that act as **super types** or **subtypes** of **parametrized** types.

```
ArrayList< ? extends Number >
```

July 27, 2012 04:45:33 PM 30 C

What is MiddleWare ?

Like many technology terms "middleware" can be difficult to define. One interesting definition is:

Middleware: *The kind of word that software industry insiders love to spew. Vague enough to mean just about any software program that functions as a link between two other programs, such as a Web server and a database program. Middleware also has a more specific meaning as a program that exists between a "network" and an "application" and carries out such tasks as authentication. But middleware functionality is often incorporated in application or network software, so precise definitions can get all messy. Avoid using at all costs.*

That's not a very informative definition. Let's define what middleware is in terms of where it resides in a software system's architecture and the functions that it performs.

The "where" is easy. Middleware occupies the "middle," in between the operating system and your applications.

One of its primary tasks is to connect systems, applications, and databases together in a secure and reliable way. For example, let's say you bought a sweater at a store web site last night. What happened? You looked through various sweaters' images, selected color and size, entered a charge card number, and that was it, right? Well, behind the scenes, middleware made sure that the store's inventory database showed that sweater in stock, connected to the charge card company's database to make sure that your card wasn't maxed out, and connected to the shipping company database to verify a delivery date. Additionally, it made sure that hundreds or thousands of people could all shop on that site at the same time. Also, while it looked to you like you were looking at one web site, middleware tied together many different computers, each in a different location, all running the store's e-commerce application, into a cluster. Why is this important? To make sure that you can always get to the store online, even if some of these computers are down due to maintenance or power failures.

Before we move on, let's look at middleware in terms of a real-world analogy:

Middleware is plumbing.

There are four ways that this is true.

First, it's mostly invisible.

You don't generally see much of the plumbing in your house. What you see is the water. As a consumer, you don't see middleware. You see the web sites and the information flow that middleware makes possible. This is part of why middleware is hard to define. If you live and work with software, and if you're reading this you probably do, then you're very aware of software packages at the top level in a logical view, such as e-commerce web applications, and packages that exist at the bottom level, such as databases and the operating system. The middle part, that plumbing that ties everything together, can seem less concrete and identifiable. Second, as a developer, you rely on middleware to provide a standard way of doing things.

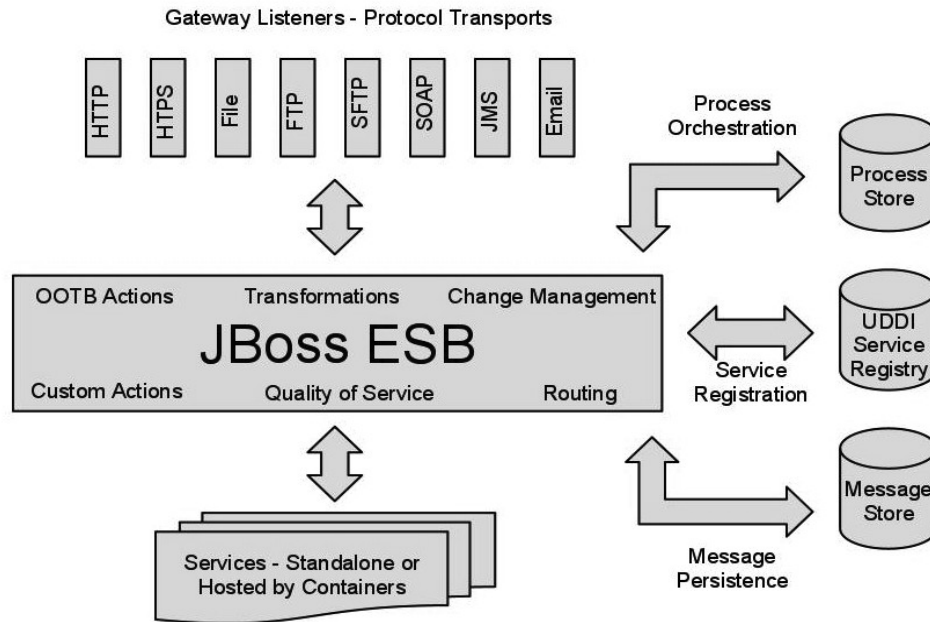
If you wanted to build your own plumbing from scratch, you could. But it's much easier to just buy plumbing fixtures. You, as a software developer, could design and build your own application servers, database connection drivers, authentication handlers, messaging systems, etc. But these wouldn't be easy to build and maintain. It's much easier to make use of middleware components that are built according to established (and especially open!) standards. In middleware, these standards take the form of libraries of functions that your applications use through well-defined application programming interfaces (APIs). You call these functions instead of having to invent your own to handle tasks such as accessing databases or executing transactions. Third, it ties together parts of complex systems. There's another similarity about your household plumbing and middleware; tying systems together. They both enable you to tie together systems that were built by different people, at different times, without your having to reconstruct everything from scratch. Think about your house for a minute. If your house is older, you probably have several generations of plumbing all working together. You didn't have to upgrade your washing machine with multiple service packs when you installed a new hot water heater. In middleware, one of the most powerful approaches is Service-oriented Architecture (SOA) based on an Enterprise Service Bus (ESB). As its name implies, an ESB provides a server, messaging, and APIs that function like a hardware bus. In order to integrate enterprise software applications developed at different times, by different organizations, and even communicating via different protocols, you don't have to rewrite them to speak one consistent language. The ESB enables you to "plug" these applications as services into the bus. The ESB takes care of transforming messages between the protocols and routing the messages to the correct services. Fourth and finally, it lets you worry about other things. When you put an addition onto a house, what do you worry about? Bathroom fixtures, kitchen appliances, flooring, colors, and how to pay for it all. It's a very stressful process. The last thing you want to worry about is whether you want 3/4-inch or half-inch pipe, copper or PVC connectors, #9 or #17 solder, etc. With middleware taking care of all the invisible functions, you, as a software developer, can concentrate on building software to solve your business problems and fulfill your customers' needs.

What is an SOA? What is an ESB?

Service Oriented Architecture (SOA) is not a single program or technology. It's really a matter of software architecture or design. Some of the basic principles of SOA are: * Based on inter-operable services: Instead of building an application from a huge monolithic program, SOA design calls for breaking systems into multiple services. These services are not rigidly coded together, but interact by sending and receiving messages.

- * Software reuse: By dividing large applications into services, individual services, or groups of services can be reused.
- * Loose coupling: Services are not rigidly coded together, but rather interact through "loose coupling" where services communicate by sending and receiving messages.
- * Abstraction: The messages sent between services follow a well-defined contract. No information on the internal implementation of the services is needed.
- * Location: Clients and services don't store the network or server location of target services. The information is made discoverable in a registry.

In hardware terms, a "bus" is a physical connector that ties together multiple systems or subsystems. Instead of having a large number of point-to-point connectors between pairs of systems, you connect each system to the bus once. An Enterprise Service Bus (ESB) does the same thing, logically, in software. Instead of passing electric current or data over the bus to and from the connections (or "endpoints") on the ESB, the ESB logically sits in the architectural layer above a messaging system. The messaging system allows for asynchronous communications between services over the ESB. In fact, when you are working with an ESB, everything is either a service (which in this context is your application software) or a message being sent between services. It's important to note that a "service" is not automatically a web service. Other types of applications, using transports such as FTP or JMS, can also be services. For more information on what an ESB is, a great resource is Enterprise Service Bus: Theory in Practice by David Chappell.



July 27, 2012 06:38:28 PM 29 C

What are qualifiers?

CDI uses a type-based resolution mechanism for **injection** and **observers**. In order to distinguish between implementations of an interface, you can use qualifiers, a type of annotations, to disambiguate. Injection points and event observers can use qualifiers to narrow down the set of candidates.

July 29, 2012 11:49:04 AM 29 C

Hypertext Transfer Protocol

The **Hypertext Transfer Protocol** (HTTP) is an [application protocol](#) for distributed, collaborative, [hypermedia](#) information systems. HTTP is the foundation of data communication for the [World Wide Web](#).

Technical overview

HTTP functions as a [request-response](#) protocol in the [client-server](#) computing model. A [web browser](#), for example, may be the *client* and an application running on a computer [hosting](#) a [web site](#) may be the *server*. The client submits an HTTP *request* message to the server. The server, which provides *resources* such as [HTML](#) files and other content, or performs other functions on behalf of the client, returns a *response* message to the client. The response contains completion status information about the request and may also contain requested content in its message body.

HTTP session

An HTTP session is a sequence of **network request-response transactions**. An HTTP client initiates a request by establishing a [Transmission Control Protocol](#) (TCP) connection to a particular [port](#) on a server (*typically port 80*). An HTTP server listening on that port waits for a client's request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own. The body of this message is typically the requested resource, although an error message or other information may also be returned.

Hypertext

Hypertext is text displayed on a [computer](#) or other electronic device with references ([hyperlinks](#)) to other text that the reader can immediately access, usually by a mouse click, key press sequence or by touching the screen. Apart from running text, hypertext may contain tables, images and other presentational devices. Hypertext is the underlying concept defining the structure of the [World Wide Web](#). It is an easy-to-use and flexible format to share information over the [Internet](#).

Request methods

HTTP defines methods (sometimes referred to as "verbs") to indicate the desired action to be performed on the identified **resource**. What this resource represents, whether pre-existing data or data that is generated dynamically, depends on the implementation of the server. Often, the resource corresponds to a file or the output of an **executable** residing on the server.

The HTTP/1.0 specification:section 8 defined the **GET**, **POST** and **HEAD** methods and the HTTP/1.1 specification:section 9 added 5 new methods: **OPTIONS**, **PUT**, **DELETE**, **TRACE** and **CONNECT**. By being specified in these documents their semantics is known and can be depended upon. Any client can use any method that they want and the server can choose to support any method it want.

Head

Asks for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.

Get

Requests a representation of the specified resource. Requests using GET should only [retrieve data](#) and should have no other effect. (This is also true of some other HTTP methods.) The [W3C](#) has published guidance principles on this distinction, saying, "[Web application](#) design should be informed by the above principles, but also by the relevant limitations."

Post

Submits data to be processed (e.g., from an HTML form) to the identified resource. **The data is included in the body of the request and the query length can be unlimited (unlike in GET)**. This may result in the creation of a new resource or the updates of existing resources or both.

Put

Uploads a representation of the specified resource.

Delete

Deletes the specified resource.

Trace

Echoes back the received request, so that a client can see what (if any) changes or additions have been made by intermediate servers.

Options

Returns the HTTP methods that the server supports for specified [URL](#). This can be used to check the functionality of a web server by requesting '*' instead of a specific resource.

Connect

Converts the request connection to a transparent [TCP/IP tunnel](#), usually to facilitate [SSL](#)-encrypted communication (HTTPS) through a non encrypted HTTP proxy.

Patch

It is used to apply partial modifications to a resource.

```

josh@blackbox:~$ telnet en.wikipedia.org 80
Trying 208.80.152.2...
Connected to rr.pmtpa.wikimedia.org.
Escape character is '^]'.
GET /wiki/Main_Page http/1.1
Host: en.wikipedia.org

HTTP/1.0 200 OK
Date: Thu, 03 Jul 2008 11:12:06 GMT
Server: Apache
X-Powered-By: PHP/5.2.5
Cache-Control: private, s-maxage=0, max-age=0, must-revalidate
Content-Language: en
Vary: Accept-Encoding, Cookie
X-Vary-Options: Accept-Encoding;list-contains=gzip, Cookie;string-contains=enwikiToken;string-contains=enwikiLoggedOut;string-contains=enwiki_session;string-contains=centralauth_Token;string-contains=centralauth_Session;string-contains=centralauth_LoggedOut
Last-Modified: Thu, 03 Jul 2008 10:44:34 GMT
Content-Length: 54218
Content-Type: text/html; charset=utf-8
X-Cache: HIT from sq39.wikimedia.org
X-Cache-Lookup: HIT from sq39.wikimedia.org:3128
Age: 3
X-Cache: HIT from sq38.wikimedia.org
X-Cache-Lookup: HIT from sq38.wikimedia.org:80
Via: 1.0 sq39.wikimedia.org:3128 (squid/2.6.STABLE18), 1.0 sq38.wikimedia.org:80 (squid/2.6.STABLE18)
Connection: close

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <meta name="keywords" content="Main Page,1778,1844,1863,1938,1980 Summer Olympics,2008,2008 Guizhou riot,2008 Jerusal
    ...
    ... This content has been removed to save space.
    ...
    "Non-profit organization">nonprofit</a> <a href="http://en.wikipedia.org/wiki/Charitable_organization" title="Charitable organization">charity</a>.<b
    r /></li>
      <li id="privacy"><a href="http://wikimediafoundation.org/wiki/Privacy_policy" title="wikimedia:Privacy policy">Privac
    y policy</a></li>
      <li id="about"><a href="/wiki/Wikipedia:About" title="Wikipedia:About">About Wikipedia</a></li>
      <li id="disclaimer"><a href="/wiki/Wikipedia:General_disclaimer" title="Wikipedia:General disclaimer">Disclaimers</a>
    </li>
    </ul>
  </div>
</div>

  <script type="text/javascript">if (window.runOnLoadHook) runOnLoadHook();</script>
<!-- Served by srv93 in 0.050 secs. --></body></html>
Connection closed by foreign host.
josh@blackbox:~$

```

An HTTP request made using telnet. The request, response headers and response body are highlighted

HTTP session state

HTTP is a [stateless protocol](#). A stateless protocol does not require the server to retain information or status about each user for the duration of multiple requests. For example, when a web server is required to customize the content of a [web page](#) for a user, the [web application](#) may have to track the user's progress from page to page.

- A common solution is the use of [HTTP cookies](#).

Other methods include

- server side sessions,
- hidden variables (when the current page contains a [form](#))
- URL-rewriting using URI-encoded parameters, e.g., `/index.php?session_id=some_unique_session_code`.

HTTP replacements or enhancements

- Representational State Transfer (REST)

JAVA API FOR RESTFUL WEB SERVICES

JAX-RS: Java API for RESTful Web Services is a [Java programming language API](#) that provides support in creating [web services](#) according to the [Representational State Transfer](#) (REST) architectural style. JAX-RS uses [annotations](#), introduced in [Java SE 5](#), to simplify the development and deployment of web service clients and endpoints.

From version 1.1 on, JAX-RS is an official part of [Java EE 6](#). A notable feature of being an official part of Java EE is that no configuration (*in the form of deployment descriptors*) is necessary to start using JAX-RS. For non-Java EE 6 environments a (small) entry in the [web.xml deployment descriptor](#) is required.

Specification

JAX-RS provides some annotations to aid in mapping a resource class (a [POJO](#)) as a web resource. The annotations include:

- `@Path` specifies the relative path for a resource class or method.
- `@GET`, `@PUT`, `@POST`, `@DELETE` and `@HEAD` specify the HTTP request type of a resource.
- `@Produces` specifies the response [MIME](#) media types.
- `@Consumes` specifies the accepted request media types.

In addition, it provides further annotations to method parameters to pull information out of the request. All the `@*Param` annotations take a key of some form which is used to look up the value required.

- `@PathParam` binds the parameter to a path segment.
- `@QueryParam` binds the parameter to the value of an HTTP query parameter.
- `@MatrixParam` binds the parameter to the value of an HTTP matrix parameter.
- `@HeaderParam` binds the parameter to an HTTP header value.
- `@CookieParam` binds the parameter to a cookie value.
- `@FormParam` binds the parameter to a form value.
- `@DefaultValue` specifies a default value for the above bindings when the key is not found.

July 31, 2012 04:11:02 PM 30 C

LOCKING IN RDBMS

- A **lock** is used when [multiple users](#) need to access a [database concurrently](#).
- This prevents data from being corrupted or invalidated when multiple users try to write to the database.
- Any single user can only modify those [database records](#) (that is, items in the database) to which they have applied a lock that gives them exclusive access to the record until the lock is released.
- Locking not only provides exclusivity to writes but also prevents (or controls) reading of unfinished modifications (AKA uncommitted data).

TWO TYPES OF LOCKING STRATEGIES

There are two mechanisms for locking data in a database:

- **Pessimistic locking:** *In pessimistic locking a record or page is locked immediately when the lock is requested. With pessimistic locking it is guaranteed that the record will be updated.*
- **Optimistic locking:** *In an optimistic lock the record or page is only locked when the changes made to that record are updated. Optimistic locking is only appropriate when there is less chance of someone needing to access the record while it is locked; otherwise it cannot be certain that the update will succeed because the attempt to update the record will fail if another user updates the record first.*

Optimistic locking in detail:

- In the field of **RDBMS**, **optimistic concurrency control (OCC)** is a concurrency control method that assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed with out locking the data resources that they affect.
- Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.
- OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions

wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods.

- However, if conflicts happen often, the cost of repeatedly restarting transactions hurts performance significantly; other concurrency control methods have better performance under these conditions.

OCC phases

More specifically, OCC transactions involve these phases:

- **Begin:** Record a time stamp marking the transaction's beginning.
- **Modify:** Read and write database values.
- **Validate:** Check whether other transactions have modified data that this transaction has used (read or wrote). Always check transactions that completed after this transaction's start time. Optionally, check transactions that are still active at validation time.
- **Commit/Rollback:** If there is no conflict, make all changes part of the official state of the database. If there is a conflict, resolve it, typically by aborting the transaction, although other resolution schemes are possible.



6.1.1. Pessimistic and optimistic locking

Transactional isolation is usually implemented by locking whatever is accessed in a transaction. There are two different approaches to transactional locking: Pessimistic locking and optimistic locking.

The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time. If most transactions simply look at the resource and never change it, an exclusive lock may be overkill as it may cause lock contention, and optimistic locking may be a better approach. With pessimistic locking, locks are applied in a fail-safe way. In the banking application example, an account is locked as soon as it is accessed in a transaction. Attempts to use the account in other transactions while it is locked will either result in the other process being delayed until the account lock is released, or that the process transaction will be rolled back. The lock exists until the transaction has either been committed or rolled back.

With optimistic locking, a resource is not actually locked when it is first accessed by a transaction. Instead, the state of the resource at the time when it would have been locked with the pessimistic locking approach is saved. Other transactions are able to concurrently access the resource and the possibility of conflicting changes is possible. At commit time, when the resource is about to be updated in persistent storage, the state of the resource is read from storage again and compared to the state that was saved when the resource was first accessed in the transaction. If the two states differ, a conflicting update was made, and the transaction will be rolled back.

In the banking application example, the amount of an account is saved when the account is first accessed in a transaction. If the transaction changes the account amount, the amount is read from the store again just before the amount is about to be updated. If the amount has changed since the transaction began, the transaction will fail itself, otherwise the new amount is written to persistent storage.

How Optimistic Locking Works

Versions

Optimistic locking requires versioning of the database objects(records). Whenever a record is modified it's version number is updated.

SELECT * FROM SECTIONALLOCATION;					
ID	ALLOCATED	OCCUPIEDCOUNT	VERSION	PERFORMANCE_ID	SECTION_ID
1	<i>null</i>	0	1	1	1

- Adding a field annotated with the `@Version` annotation automatically enables the optimistic locking.

```
@Version
@SuppressWarnings("unused")
private long version;
```

- No getters and setters are required and no setter to the version field should be provided in any case as versioning is handled by the hibernate or the underlying database.
- The version property on the class can be numeric (short, int, or long) or timestamp or calendar.

Optimistic Locking Working:

- First a record is fetched no lock is applied on that record at this time.
- Then record is modified.
- When persisting that record in database the version property of the entity is matched with the version column in the database of both matches it means the record in the database is not modified and the transaction is committed and the **version column is updated incremented** (depends on whether we are using *int*, *long*, *short* or *time stamp* as the data type of the versioned property).
- But if the version column of the record do not matches the version property of the entity then it means that the record is modified in the database and an exception **StaleObjectStateException** is thrown and the transaction is **rolled back**.

August 12, 2012 11:51:55 PM 28 C

Cloud Computing

Cloud computing is the use of computing resources such as *hardware* or *software* which are delivered as a service over a network; most probably internet.

There are three types of cloud computing:

1. Infrastructure as a service. (IaaS)
2. Platform as a service. (PaaS)
3. Software as a service. (SaaS)

Platform As A Service

In the PaaS model, cloud providers deliver a **computing platform** typically including operating system, programming language execution environment, database, and web server. Application developers can develop and run their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers. With some PaaS offers, the underlying computer and storage resources scale automatically to match application demand such that cloud user does not have to allocate resources manually (*as provided by the RedHat OpenShift*).

Examples of PaaS include: Amazon Elastic Beanstalk, [Google App Engine](#), RedHat OpenShift and [Microsoft Azure](#).

August 14, 2012 11:17:17 AM 29 C

Secure Shell

Secure Shell (SSH) is a network protocol for **secure data communication**, **remote shell services** or **command execution** and other secure network services between two networked computers that it connects via a secure channel over an insecure network: a server and a client (running SSH server and SSH client programs, respectively).

The best-known application of the protocol is for access to [shell accounts](#) on [Unix-like](#) operating systems.

It was designed as a replacement for [Telnet](#) and other [insecure](#) remote [shell](#) protocols such as the Berkeley [rsh](#) and [rexec](#) protocols, which send information, notably [passwords](#), in [plaintext](#), rendering them susceptible to interception and disclosure using [packet analysis](#). The [encryption](#) used by SSH is intended to provide confidentiality and integrity of data over an unsecured network, such as the [Internet](#).

Hence SSH is better than telnet etc.

- SSH uses [public-key cryptography](#) to [authenticate](#) the remote computer.
- Anyone can produce a matching pair of different keys (public and private). By using utilities such as `ssh-keygen` in linux.
- *The public key is placed on all computers that must allow access to the owner of the matching private key (the owner keeps the private key secret).*
- *SSH only verifies if the same person offering the public key also owns the matching private key.*
- While authentication is based on the private key, the key itself is never transferred through the network during authentication.

Key Management

- On Unix Like systems, the list of authorized keys is stored in the home folder of the user that is allowed to log in remotely. For example:

```
[Affan@AffanArbeitsplatz ~]$ ls /home/Affan/.ssh/
authorized_keys  id_rsa  id_rsa.pub  known_hosts  other_keys.seahorse
[Affan@AffanArbeitsplatz ~]$
```

Usage

- SSH is typically used to log into a remote machine and execute commands.
- But it also supports [tunneling](#), [forwarding TCP ports](#) and [X11](#) connections; it can transfer files using the associated [SSH file transfer](#) (SFTP) or [secure copy](#) (SCP) protocols. SSH uses the [client-server](#) model.
- The standard TCP port # 22 has been assigned for contacting SSH servers, though administrators frequently change it to a non-standard port, believing it adds an additional security measure.

August 21, 2012 08:15:58 AM 28 C

JPA

Mapped Super Class

Designates a class whose mapping information is applied to the entities that inherit from it. A mapped superclass has no separate table defined for it.

A class designated with the `@MappedSuperclass` annotation can be mapped in the same way as an entity except that the mappings will apply only to its subclasses since no table exists for the mapped superclass itself. When applied to the subclasses the inherited mappings will apply in the context of the subclass tables. Mapping information may be overridden in such subclasses by using the [AttributeOverride](#) and [AssociationOverride](#) annotations or corresponding XML elements.

OR

Entities may inherit from superclasses that contain persistent state and mapping information but are not entities. That is, the superclass is not decorated with the `@Entity` annotation and is not mapped as an entity by the Java Persistence provider.

These superclasses are most often used when you have state and mapping information common to multiple

entity classes.

```

@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}

```

- Mapped superclasses cannot be queried and can't be used in EntityManager or Query operations.
- You must use entity subclasses of the mapped superclass in EntityManager or Query operations.
- **Mapped superclasses can't be targets of entity relationships. Mapped superclasses can be abstract or concrete.**
- Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from the mapped superclass define the table mappings. For instance, in the preceding code sample, the underlying tables would be **FULLTIMEEMPLOYEE** and **PARTTIMEEMPLOYEE**, but there is no **EMPLOYEE** table.

Non-Entity Superclasses

- Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete.
- **The state of non-entity superclasses is nonpersistent, and any state inherited from the non-entity superclass by an entity class is nonpersistent.**
- Non-entity superclasses may not be used in **EntityManager** or **Query** operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

Orphan Removal in Relationships

- When a target entity in one-to-one or one-to-many relationship is removed from the relationship, it is often desirable to cascade the remove operation to the target entity.
- Such target entities are considered "orphans," and the orphanRemoval attribute can be used to specify that orphaned entities should be removed.
- For example, if an order has many line items and one of them is removed from the order, the removed line item is considered an orphan. If orphanRemoval is set to true, the line item entity will be deleted when the line item is removed from the order.
- The orphanRemoval attribute in @OneToMany and @OneToOne takes a Boolean value and is by default false.
- The following example will cascade the remove operation to the orphaned customer entity when it is removed from the relationship:

```
@OneToMany(mappedBy="customer", orphanRemoval="true")
```

```
public List<Order> getOrders() { ... }
```

Entity Inheritance

Entities support **class inheritance**, **polymorphic associations**, and **polymorphic queries**. Entity classes can extend **non-entity classes**, and **non-entity classes can extend entity classes**. Entity classes can be both abstract and concrete.

Abstract Entities

An abstract class may be declared an entity by decorating the class with **@Entity**. Abstract entities are like concrete entities but cannot be instantiated.

Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity:

```
@Entity
```

```
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

September 5, 2012 12:13:09 PM 31 C Broken Clouds

STUDY OF ORM IN JPA 2

Persistence Annotations

- Persistence annotations are applied to three levels **class**, **field** and **property(method)** level.
- The annotations are divided into two categories **logical annotations**, **physical annotations**.
 - **Logical annotations** deal with:
 - Describing entity model from the object modeling point of view.
 - They are tightly bound to the **domain model** and are sort of the meta data tat we might want to specify in our UML or any other modeling framework.
 - **Physical annotations** deal with the concrete data model in the database such as tables, columns, constraints & other database level artifacts.

Discussion about accessing the entity's persistent state:

We can specify how the persistence provider will access our entity's persistent state, when persisting

to database and loading data from database into entities.

There are two ways of accessing the entity's persistent state **field access**, **property(method) access**.

Field Access:

- When field access is enabled then there will be no need for the getters and setters **and the provider will access the fields using reflection**.
 - **Reflection:** In computer science the term **reflection** means the ability of a computer program to dynamically examine and modify the structure and behavior(*specifically the meta data, values, properties and functions*) of an object at the **run time**.
 - Reflection is commonly used in high level VM programming languages like **JAVA**.
- When field access is enabled a field is not required to have getters and setters, but it must be declared private, protected or package **not Public**. If other classes in the VM needs to access the persistent field they must use the getters and setters they cannot directly access a **Field access enabled** field of an entity.

Example:

@Entity

```
public class Day extends BaseEntity implements Serializable{

    private static final long serialVersionUID = -7647690467575931834L;

    @NotNull
    @Column(unique = true)
    @Temporal(TemporalType.DATE)
    private Date date;
}
```

In the above example by writing the annotations above the field we are enabling field access. The provider will access this field directly at run time using **reflection**.

Field access is most common approach, however sometimes we specifically want the container to invoke the property method to access the field such as in case if we need explicit conversion when storing/retrieving the data into database.

Property Access:

- When property access is used the same contract as for the java beans applies & there must be getters & setters.
- The mapping annotations must be on the getter method.
- The method must be either **public or protected**.

Listing 4-2. Using Property Access

```

@Entity
public class Employee {
    private int id;
    private String name;
    private long wage;

    @Id public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public long getSalary() { return wage; }
    public void setSalary(long salary) { this.wage = salary; }
}

```

- In the above we applied @Id annotation the getId() so the provider will use property access for this whole class.
- Notice that getSalary() is backed by field **wage** but the database column will be **salary**. The field name is ignored.

September 18, 2012 10:30:25 PM

Mixed Access Mode:

- It is also possible to mix field & property access modes with in an entity or in an entity hierarchy.
- It is useful in case when we need some **special transformation** while persisting to or retrieving data from the database.
- **@Access** annotation is used to denote the access type in an entity.
- Consider an employee entity that has a phone number field. Phone numbers contains the area code and we only want to store area for those numbers which are not local.
- So we use property access for the employee entity.

```

@Entity
@Access(AccessType.FIELD)
public class Employee { ... }

```

- The next step is to annotate the respected phone number property with the @Access annotation.

```

@Access(AccessType.PROPERTY) @Column(name="PHONE")
protected String getPhoneNumberForDb() { ... }

```

- We also need to mark the phone number field with the **@Transient** annotation.

```

@Transient private String phoneNum;

```

- We did so; so that the phone number will not be persisted twice due to the default **AccessType.field** behavior.

"@Transient annotation is used to mark those fields in an entity which will not be persisted to the database."

*"Java also has a keyword transient which is used to mark those fields in a class which are not supposed to be the part of **object serialization**. Fields starting with the **transient** keyword are also non persistent"*

Mapping to a table in database:

To map a table to database we use **@Table** annotation. In **RDBMS** the schema names are used to differentiate one set of table from another. We can denote the schema name in @Table annotation:

```
@Entity
@Table(name="EMP", catalog="HR")
public class Employee { ... }
```

Mapping simple types:

Simple Java types are mapped as part of the immediate state of an entity in its fields or properties. The list of persistable types is quite lengthy and includes pretty much every type that you would want to persist. They include the following:

- **Primitive Java types:** byte, int, short, long, boolean, char, float, double
- **Wrapper classes of primitive Java types:** Byte, Integer, Short, Long, Boolean, Character, Float, Double
- **Byte and character array types:** byte[], Byte[], char[], Character[]
- **Large numeric types:** java.math.BigInteger, java.math.BigDecimal
- **Strings:** java.lang.String
- **Java temporal types:** java.util.Date, java.util.Calendar
- **Enumerated types:** Any system or user-defined enumerated type
- **Serializable objects:** Any system or user-defined serializable type

We can also use **@Basic** annotation on the fields to specifically mark them as persistent. However it is only for the documentation purposes and is not a requirement for field to be persistent.

Lazy Fetching:

Generally we use the concept of lazy fetching for the collections inside entities. But we can also use lazy fetching for the individual fields in an entity; by using **@Basic** annotation.

On occasion, we know that certain portions of an entity will be seldom accessed. In these situations, we can optimize the performance when retrieving the entity by fetching only the data that we expect to be frequently accessed. We would like the remainder of the data to be fetched only when or if it is required. There are many names for this kind of feature, including lazy loading, deferred loading, lazy fetching, on-demand fetching, just-in-time reading, indirection, and others. They all mean pretty much the same thing, which is just that some data might not be loaded when the object is initially read from the database, but will be fetched only when it is referenced or accessed.

The *fetch type* of a basic mapping can be configured to be lazily or eagerly loaded by specifying the fetch element in the corresponding **@Basic** annotation. The FetchType enumerated type defines the values for this element, which can be either EAGER or LAZY. Setting the fetch type of a basic mapping to LAZY means that the provider might defer loading the state for that attribute until it is referenced. The default is to load all basic mappings eagerly. Listing 4-8 shows an example of overriding a basic mapping to be lazily loaded.

```

@Entity
public class Employee {
    // ...
    @Basic(fetch=FetchType.LAZY)
    @Column(name="COMM")
    private String comments;
    // ...
}

```

We are assuming in this example that applications will seldom access the comments in an employee record, so we mark it as being lazily fetched. Note that in this case the `@Basic` annotation is not only present for documentation purposes but also required in order to specify the fetch type for the field. Configuring the comments field to be fetched lazily will allow an `Employee` instance returned from a query to have the comments field empty. The application does not have to do anything special to get it, however. By simply accessing the comments field, it will be transparently read and filled in by the provider if it was not already loaded.

Before you use this feature, you should be aware of a few pertinent points about lazy attribute fetching. First and foremost, the directive to lazily fetch an attribute is meant only to be a hint to the persistence provider to help the application achieve better performance. The provider is not required to respect the request because the behavior of the entity is not compromised if the provider goes ahead and loads the attribute. The converse is not true, though, because specifying that an attribute be eagerly fetched might be critical to being able to access the entity state once the entity is detached from the persistence context. We will discuss detachment more in Chapter 6 and explore the connection between lazy loading and detachment.

Second, on the surface it might appear that this is a good idea for certain attributes of an entity, but in practice it is almost never a good idea to lazily fetch simple types. There is little to be gained in returning only part of a database row unless you are certain that the state will not be accessed in the entity later on. The only times when lazy loading of a basic mapping should be considered are when there are many columns in a table (for example, dozens or hundreds) or when the columns are large (for example, very large character strings or byte strings). It could take significant resources to load the data, and not loading it could save quite a lot of effort, time, and resources. Unless either of these two cases is true, in the majority of cases lazily fetching a subset of object attributes will end up being more expensive than eagerly fetching them.

*"Lazy fetching in fields is only useful when we have hundreds of columns
OR
When columns are large enough i.e. BLOBS or CLOBS
OTHERWISE
lazy fetching in fields will be expensive instead of being efficient"*

Large Objects

A common database term for a character or byte-based object that can be very large (up to the gigabyte range) is *large object*, or *LOB* for short. Database columns that can store these types of large objects require special JDBC calls to be accessed from Java. To signal to the provider that it should use the LOB methods when passing and retrieving this data to and from the JDBC driver, an additional annotation must be added to the basic mapping. The `@Lob` annotation acts as the marker annotation to fulfill this purpose and might appear in conjunction with the `@Basic` annotation, or it might appear when `@Basic` is absent and implicitly assumed to be on the mapping.

Because the `@Lob` annotation is really just qualifying the basic mapping, it can also be accompanied by a `@Column` annotation when the name of the LOB column needs to be overridden from the assumed default name.

CLOB Or BLOB can be as large up to gigabytes

LOBs come in two flavors in the database: character large objects, called *CLOBs*, and binary large objects, or *BLOBs*. As their names imply, a CLOB column holds a large character sequence, and a BLOB column can store a large byte sequence. The Java types mapped to BLOB columns are `byte[]`, `Byte[]`, and `Serializable` types, while `char[]`, `Character[]`, and `String` objects are mapped to CLOB columns. The provider is responsible for making this distinction based on the type of the attribute being mapped.

An example of mapping an image to a BLOB column is shown in Listing 4-9. Here, the `PIC` column is assumed to be a BLOB column to store the employee picture that is in the `picture` field. We have also marked this field to be loaded lazily, a common practice applied to LOBs that do not get referenced often.

Listing 4-9. Mapping a BLOB Column

```
@Entity
public class Employee {
    @Id
    private int id;
    @Basic(fetch=FetchType.LAZY)
    @Lob @Column(name="PIC")
    private byte[] picture;
    // ...
}
```

Enumerated Types:

In JPA the default storage type for the enumerations is **ORDINAL**.

Temporal Types

Temporal types are the set of time-based types that can be used in persistent state mappings. The list of supported temporal types includes the three `java.sql` types `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`, and it includes the two `java.util` types `java.util.Date` and `java.util.Calendar`.

The `java.sql` types are completely hassle-free. They act just like any other simple mapping type and do not need any special consideration. The two `java.util` types need additional metadata, however, to indicate which of the JDBC `java.sql` types to use when communicating with the JDBC driver. This is done by annotating them with the `@Temporal` annotation and specifying the JDBC type as a value of the `TemporalType` enumerated type. There are three enumerated values of `DATE`, `TIME`, and `TIMESTAMP` to represent each of the `java.sql` types.

Listing 4-12 shows how `java.util.Date` and `java.util.Calendar` can be mapped to date columns in the database.

Listing 4-12. Mapping Temporal Types

```
@Entity
public class Employee {
    @Id
    private int id;
    @Temporal(TemporalType.DATE)
    private Calendar dob;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}
```

Like the other varieties of basic mappings, the `@Column` annotation can be used to override the default column name.

Transient State

Attributes that are part of a persistent entity but not intended to be persistent can either be modified with the transient modifier in Java or be annotated with the `@Transient` annotation. If either is specified, the provider runtime will not apply its default mapping rules to the attribute on which it was specified.

Transient fields are used for various reasons. One might be the case earlier on in the chapter when we mixed the access mode and didn't want to persist the same state twice. Another might be when you want to cache some in-memory state that you don't want to have to recompute, rediscover, or reinitialize. For example, in Listing 4-13 we are using a transient field to save the correct locale-specific word for *Employee* so that we print it correctly wherever it is being displayed. We have used the transient modifier instead of the `@Transient` annotation so that if the *Employee* gets serialized from one VM to another then the translated name will get reinitialized to correspond to the locale of the new VM. In cases where the non-persistent value should be retained across serialization, the annotation should be used instead of the modifier.

*"Main difference between keyword **transient** & **@Transient** annotation is that:*

- *If we want a non persistent field ; **not to become the part of object serialization** then we use the keyword **transient**. (as the purpose of keyword **transient** is to mark a field un serializable)*
- *If we want a non persistent field **to become the part of object serialization** then we use **@Transient** annotation."*

Listing 4-13. Using a Transient Field

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    transient private String translatedName;
    // ...

    public String toString() {
        if (translatedName == null) {
            translatedName =
                ResourceBundle.getBundle("EmpResources").getString("Employee");
        }
        return translatedName + ": " + id + " " + name;
    }
}
```

September 20, 2012 12:18:56 PM 33 C

MAPPING OF THE PRIMARY KEYS

Every entity must map to the primary key column in the database. To do this we use `@Id` annotation.

Some discussion about the `@Column` annotation:

```

@NotNull
@Temporal(TemporalType.TIME)
@Column(name = "start_time")
private Date startTime;

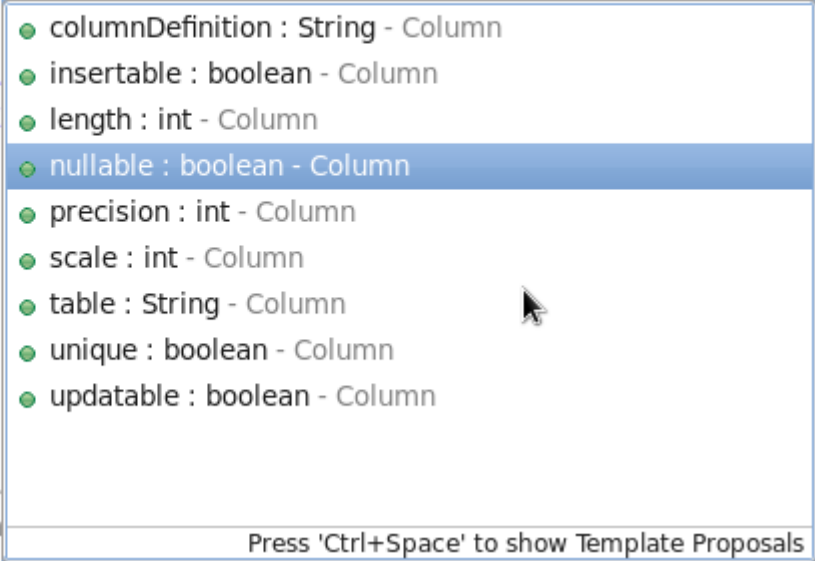
@NotEmpty(message = "Brief de")
@Size(max = 313, message = "E")
private String description;

public BaseActivity() {
    super();
}

public Date getStartTime() {
    return this.startTime;
}

public void setStartTime(Date
    this.startTime = startTim
}

```



@Column annotation is used to modify the properties of a database column at an entity's persistent field level; when no @Column annotation is provided then the default values applies.

Example 1:

```

@Column(name="DESC", nullable=false, length=512)
public String getDescription() { return description; }

```

Example 2:

```

@Column(name="DESC",
        columnDefinition="CLOB NOT NULL",
        table="EMP_DETAIL")
@Lob
public String getDescription() { return description; }

```

Example 3:

```

@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
public BigDecimal getCost() { return cost; }

```

Precision is the number of digits in a number. Scale is the number of digits to the right of the decimal point in a number. For example, the number 123.45 has a precision of 5 and a scale of 2.

Annotation Elements

String [columnDefinition](#)

(Optional) The SQL fragment that is used when generating the DDL for the column.

Defaults to the generated SQL to create a column of the inferred type.

Default value:

""

Since:

JPA 1.0

boolean [insertable](#)

(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.

Default value:

true

Since:

JPA 1.0

int [length](#)

(Optional) The column length. (Applies only if a string-valued column is used.)

Default value:

255

Since:

JPA 1.0

Question: Why to use `insertable = false` and `updatable = false` in `@Column` or `@JoinColumn` ?

insertable = false is useful when creating composite primary keys



Let me answer step by step.

1. When do you need `insertable = false, updatable = false`?

Lets look at the below mapping,

```
public class zip {  
  
    @ManyToOne  
    @JoinColumn(name = "country_code", referencedColumnName = "iso_code")  
    private Country country = null  
  
    @Column(name = "country_code")  
    private String countryCode;  
  
}
```

Here we are referring to the same column in the table using two different properties. In the below code,

```
Zip z = new zip();  
  
z.setCountry(getCountry("US"));  
x.setCountryCode("IN");  
  
saveZip(z);
```

What will hibernate do here??

To prevent these kind of inconsistency, hibernate is asking you to specify the update point of relations. Which means **you can refer to the same column in the table n number of times but only one of them can be used to update and all others will be read only.**

2. Why is hibernate complaining about your mapping?

In your `Zip` class you are referring to the Embedded id class `ZipId` that again contains the country code. As in the above scenario now you have a possibility of updating the `country_code` column from two places. Hence error given by hibernate is proper.

3. How to fix it in your case?

No. Ideally you want your `ZipId` class to generate the id, so you should not add `insertable = false, updatable = false` to the `countryCode` inside the `ZipId`. So the fix is as below modify the `country` mapping in your `zip` class as below,

```
@ManyToOne  
@JoinColumn(name = "country_code", referencedColumnName = "iso_code",  
insertable = false, updatable = false)  
private Country country;
```

Hope this helps your understanding.

int scale

(Optional) The scale for a decimal (exact numeric) column. (Applies only if a decimal column is used.)

Default value:

0

Since:

JPA 1.0

String table

(Optional) The name of the table that contains the column. If absent the column is assumed to be in the primary table.

Default value:

""

Since:

JPA 1.0

boolean unique

(Optional) Whether the column is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint corresponds to only a single column. This constraint applies in addition to any constraint entailed by primary key mapping and to constraints specified at the table level.

Default value:

false

Since:

JPA 1.0

boolean **updatable**

(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.

Default value:

true

Since:

JPA 1.0