What is JDBC?

JDBC stands for **Java Database Connectivity**. It provides a set of Java API for accessing the relational databases from Java program. These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant database.

JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.

Pros and Cons of JDBC

|  |  |
| --- | --- |
| **Pros of JDBC** | **Cons of JDBC** |
| Clean and simple SQL processing  Good performance with large data  Very good for small applications  Simple syntax so easy to learn | Complex if it is used in large projects  Large programming overhead  No encapsulation  Hard to implement MVC concept  Query is DBMS specific |

Why Object Relational Mapping (ORM)?

When we work with an object-oriented system, there is a mismatch between the object model and the relational database. RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C# represent it as an interconnected graph of objects.

Consider the following Java Class with proper constructors and associated public function −

public class Employee {

private int id;

private String first\_name;

private String last\_name;

private int salary;

public Employee() {}

public Employee(String fname, String lname, int salary) {

this.first\_name = fname;

this.last\_name = lname;

this.salary = salary;

}

public int getId() {

return id;

}

public String getFirstName() {

return first\_name;

}

public String getLastName() {

return last\_name;

}

public int getSalary() {

return salary;

}

}

Consider the above objects are to be stored and retrieved into the following RDBMS table −

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

What is ORM?

ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

An ORM system has the following advantages over plain JDBC −

|  |  |
| --- | --- |
| **Sr.No.** | **Advantages** |
| 1 | Let’s business code access objects rather than DB tables. |
| 2 | Hides details of SQL queries from OO logic. |
| 3 | Based on JDBC 'under the hood.' |
| 4 | No need to deal with the database implementation. |
| 5 | Entities based on business concepts rather than database structure. |
| 6 | Transaction management and automatic key generation. |
| 7 | Fast development of application. |

An ORM solution consists of the following four entities −

|  |  |
| --- | --- |
| **Sr.No.** | **Solutions** |
| 1 | An API to perform basic CRUD operations on objects of persistent classes. |
| 2 | A language or API to specify queries that refer to classes and properties of classes. |
| 3 | A configurable facility for specifying mapping metadata. |
| 4 | A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions. |

Java ORM Frameworks

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.

* Enterprise JavaBeans Entity Beans
* Java Data Objects
* Castor
* TopLink
* Spring DAO
* Hibernate
* And many more

Hibernate is an **O**bject-**R**elational **M**apping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.



Hibernate Advantages

* Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
* Provides simple APIs for storing and retrieving Java objects directly to and from the database.
* If there is change in the database or in any table, then you need to change the XML file properties only.
* Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.
* Hibernate does not require an application server to operate.
* Manipulates Complex associations of objects of your database.
* Minimizes database access with smart fetching strategies.
* Provides simple querying of data.

Supported Databases

Hibernate supports almost all the major RDBMS. Following is a list of few of the database engines supported by Hibernate −

* HSQL Database Engine
* DB2/NT
* MySQL
* PostgreSQL
* FrontBase
* Oracle
* Microsoft SQL Server Database
* Sybase SQL Server
* Informix Dynamic Server

Supported Technologies

Hibernate supports a variety of other technologies, including −

* XDoclet Spring
* J2EE
* Eclipse plug-ins
* Maven

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

Following is a very high level view of the Hibernate Application Architecture.



Following is a detailed view of the Hibernate Application Architecture with its important core classes.



Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

Following section gives brief description of each of the class objects involved in Hibernate Application Architecture.

Configuration Object

The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate.

The Configuration object provides two keys components −

* **Database Connection** − This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.
* **Class Mapping Setup** − This component creates the connection between the Java classes and database tables.

SessionFactory Object

Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

Session Object

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

Transaction Object

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

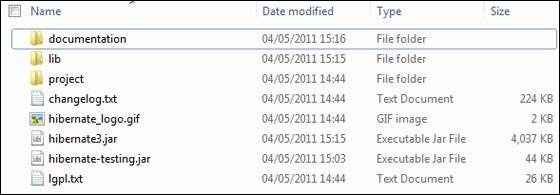
Query Object

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Downloading Hibernate

It is assumed that you already have the latest version of Java installed on your system. Following are the simple steps to download and install Hibernate on your system −

* Make a choice whether you want to install Hibernate on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.
* Download the latest version of Hibernate from <http://www.hibernate.org/downloads>.
* At the time of writing this tutorial, I downloaded **hibernate-distribution3.6.4.Final** and when you unzip the downloaded file, it will give you directory structure as shown in the following image



Installing Hibernate

Once you downloaded and unzipped the latest version of the Hibernate Installation file, you need to perform following two simple steps. Make sure you are setting your CLASSPATH variable properly otherwise you will face problem while compiling your application.

* Now, copy all the library files from **/lib** into your CLASSPATH, and change your classpath variable to include all the JARs −
* Finally, copy **hibernate3.jar** file into your CLASSPATH. This file lies in the root directory of the installation and is the primary JAR that Hibernate needs to do its work.

Hibernate requires to know in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

I will consider XML formatted file **hibernate.cfg.xml** to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Properties

Following is the list of important properties, you will be required to configure for a databases in a standalone situation −

|  |  |
| --- | --- |
| **Sr.No.** | **Properties & Description** |
| 1 | **hibernate.dialect**  This property makes Hibernate generate the appropriate SQL for the chosen database. |
| 2 | **hibernate.connection.driver\_class**  The JDBC driver class. |
| 3 | **hibernate.connection.url**  The JDBC URL to the database instance. |
| 4 | **hibernate.connection.username**  The database username. |
| 5 | **hibernate.connection.password**  The database password. |
| 6 | **hibernate.connection.pool\_size**  Limits the number of connections waiting in the Hibernate database connection pool. |
| 7 | **hibernate.connection.autocommit**  Allows autocommit mode to be used for the JDBC connection. |

# Hibernate - Sessions

Advertisements

[Previous Page](https://www.tutorialspoint.com/hibernate/hibernate_configuration.htm)

[Next Page](https://www.tutorialspoint.com/hibernate/hibernate_persistent_classes.htm)

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.

Instances may exist in one of the following three states at a given point in time −

* **transient** − A new instance of a persistent class, which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.
* **persistent** − You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
* **detached** − Once we close the Hibernate Session, the persistent instance will become a detached instance.

A Session instance is serializable if its persistent classes are serializable. A typical transaction should use the following idiom −

Session session = factory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

// do some work

...

tx.commit();

}

catch (Exception e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

} finally {

session.close();

}

The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. A mapping document helps Hibernate in determining how to pull the values from the classes and map them with table and associated fields.

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate. Hibernate works best if these classes follow some simple rules, also known as the **Plain Old Java Object** (POJO) programming model.

There are following main rules of persistent classes, however, none of these rules are hard requirements −

* All Java classes that will be persisted need a default constructor.
* All classes should contain an ID in order to allow easy identification of your objects within Hibernate and the database. This property maps to the primary key column of a database table.
* All attributes that will be persisted should be declared private and have **getXXX** and **setXXX** methods defined in the JavaBean style.
* A central feature of Hibernate, proxies, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.
* All classes that do not extend or implement some specialized classes and interfaces required by the EJB framework.

The POJO name is used to emphasize that a given object is an ordinary Java Object, not a special object, and in particular not an Enterprise JavaBean.

## Simple POJO Example

Based on the few rules mentioned above, we can define a POJO class as follows −

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

public Employee() {}

public Employee(String fname, String lname, int salary) {

this.firstName = fname;

this.lastName = lname;

this.salary = salary;

}

public int getId() {

return id;

}

public void setId( int id ) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName( String first\_name ) {

this.firstName = first\_name;

}

public String getLastName() {

return lastName;

}

public void setLastName( String last\_name ) {

this.lastName = last\_name;

}

public int getSalary() {

return salary;

}

public void setSalary( int salary ) {

this.salary = salary;

}

}

# Hibernate - Mapping Files

Advertisements

[Previous Page](https://www.tutorialspoint.com/hibernate/hibernate_persistent_classes.htm)

[Next Page](https://www.tutorialspoint.com/hibernate/hibernate_mapping_types.htm)

An Object/relational mappings are usually defined in an XML document. This mapping file instructs Hibernate — how to map the defined class or classes to the database tables?

Though many Hibernate users choose to write the XML by hand, but a number of tools exist to generate the mapping document. These include **XDoclet, Middlegen** and **AndroMDA** for the advanced Hibernate users.

Let us consider our previously defined POJO class whose objects will persist in the table defined in next section.

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

public Employee() {}

public Employee(String fname, String lname, int salary) {

this.firstName = fname;

this.lastName = lname;

this.salary = salary;

}

public int getId() {

return id;

}

public void setId( int id ) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName( String first\_name ) {

this.firstName = first\_name;

}

public String getLastName() {

return lastName;

}

public void setLastName( String last\_name ) {

this.lastName = last\_name;

}

public int getSalary() {

return salary;

}

public void setSalary( int salary ) {

this.salary = salary;

}

}

There would be one table corresponding to each object you are willing to provide persistence. Consider above objects need to be stored and retrieved into the following RDBMS table −

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

Based on the two above entities, we can define following mapping file, which instructs Hibernate how to map the defined class or classes to the database tables.

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

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name = "Employee" table = "EMPLOYEE">

<meta attribute = "class-description">

This class contains the employee detail.

</meta>

<id name = "id" type = "int" column = "id">

<generator class="native"/>

</id>

<property name = "firstName" column = "first\_name" type = "string"/>

<property name = "lastName" column = "last\_name" type = "string"/>

<property name = "salary" column = "salary" type = "int"/>

</class>

</hibernate-mapping>

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml.

Let us see understand a little detail about the mapping elements used in the mapping file −

* The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains all the **<class>** elements.
* The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database **table** name is specified using the table attribute.
* The **<meta>** element is optional element and can be used to create the class description.
* The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.
* The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence**, or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.
* The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

# Spring - MVC Framework

Advertisements

[Previous Page](https://www.tutorialspoint.com/spring/spring_transaction_management.htm)

[Next Page](https://www.tutorialspoint.com/spring/logging_with_log4j.htm)

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

## The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram −



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* −

* After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
* The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
* The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
* Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* w which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

## Required Configuration

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file. The following is an example to show declaration and mapping for **HelloWeb** *DispatcherServlet* example −

<web-app id = "WebApp\_ID" version = "2.4"

xmlns = "http://java.sun.com/xml/ns/j2ee"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>\*.jsp</url-pattern>

</servlet-mapping>

</web-app>

The **web.xml** file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of **HelloWeb** DispatcherServlet, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INFdirectory. In this case, our file will be **HelloWebservlet.xml**.

Next, <servlet-mapping> tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

If you do not want to go with default filename as *[servlet-name]-servlet.xml* and default location as *WebContent/WEB-INF*, you can customize this file name and location by adding the servlet listener *ContextLoaderListener* in your web.xml file as follows −

<web-app...>

<!-------- *DispatcherServlet* definition goes here----->

....

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

</web-app>

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory −

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:context = "http://www.springframework.org/schema/context"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package = "com.tutorialspoint" />

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value = ".jsp" />

</bean>

</beans>

Following are the important points about **HelloWeb-servlet.xml** file −

* The *[servlet-name]-servlet.xml* file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
* The *<context:component-scan...>* tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
* The *InternalResourceViewResolver* will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at */WEB-INF/jsp/hello.jsp* .

The following section will show you how to create your actual components, i.e., Controller, Model, and View.

## Defining a Controller

The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The **@Controller**annotation indicates that a particular class serves the role of a controller. The **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

@Controller

@RequestMapping("/hello")

public class HelloController {

@RequestMapping(method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation**@RequestMapping(method = RequestMethod.GET)** is used to declare theprintHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write the above controller in another form where you can add additional attributes in *@RequestMapping* as follows −

@Controller

public class HelloController {

@RequestMapping(value = "/hello", method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request. The following important points are to be noted about the controller defined above −

* You will define required business logic inside a service method. You can call another method inside this method as per requirement.
* Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
* A defined service method can return a String, which contains the name of the **view** to be used to render the model. This example returns "hello" as logical view name.

## Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

Let us write a simple **hello** view in /WEB-INF/hello/hello.jsp −

<html>

<head>

<title>Hello Spring MVC</title>

</head>

<body>

<h2>${message}</h2>

</body>

</html>

Here **${message}** is the attribute which we have set up inside the Controller. You can have multiple attributes to be displayed inside your view.

## Hibernate Single Table Strategy

In case of single table strategy, there is a single table created per inheritance hierachy. For example, we have Employee class being extended by 2 others classes but when it comes to single table strategy a single table will be created representing all the classes per inheritance hieracy and this table will contain all the data related to either Employee or ContractEmployee or PermanentEmployee.

So, the question arises as if all the entries are made in a single table then how can we identify those rows from object perspective. For this, hbernate provides a Discriminator Type(DType) column which helps to differentiate between these records. This configuration is completely annotation based. So let us define our entities and implement hibernate inheritance with Single Table Strategy.

**Other Interesting Posts**

[Spring Hibernate Integration Example with JavaConfig](https://www.devglan.com/spring-mvc/spring-hibernate-integration-example-javaconfig)

[Object Relational Mapping in Java](https://www.devglan.com/hibernate/object-relational-mapping-in-java)

[Hibernate Different Annotations Example](https://www.devglan.com/hibernate/hibernate-annotations-example)

[Hibernate One to Many Mapping Example](https://www.devglan.com/hibernate/hibernate-one-to-one-mapping-example)

[Hibernate One to Many Relationship Example](https://www.devglan.com/hibernate/hibernate-one-to-many-relationship-example)

[Hibernate Many to Many Relationship Example](https://www.devglan.com/hibernate/hibernate-many-to-many-mapping-example)

**Employee.java**

@Entity

@Table(name = "EMPLOYEE")

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "EMP\_ID")

private int empId;

@Column(name ="name")

private String name;

**PermanentEmployee.java**

@Entity

@Table(name="PERMANENT\_EMPLOYEE")

@DiscriminatorValue("PERMANENT\_EMP")

public class PermanentEmployee extends Employee {

@Column(name="TYPE")

private String type;

**ContractEmployee.java**

@Entity

@Table(name="CONTRACT\_EMPLOYEE")

@DiscriminatorValue("CONTRACT\_EMP")

public class ContractEmployee extends Employee {

@Column(name="TYPE")

private String type;

@Inheritance - It is used to define the type of inheritance used in hibernate and it is defined in the parent class. If the Inheritance annotation is not specified or if no inheritance type is specified for an entity class hierarchy, the SINGLE\_TABLE mapping strategy is used.

@DiscriminatorValue - This annotation is used to specify the DType column name. Here we have defined it as PERMANENT\_EMP in case of PermanentEmployee.java and CONTRACT\_EMPLOYEE in case of ContractEmployee.java. The DiscriminatorValue annotation can only be specified on a concrete entity class. If the DiscriminatorType is STRING, the discriminator value default is the entity name.

## Defining hibernate.cfg.xml

<hibernate-configuration>

<session-factory>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost/test</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">root</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<property name="show\_sql">false</property>

<mapping class="com.devglan.model.Employee"/>

<mapping class="com.devglan.model.ContractEmployee"/>

<mapping class="com.devglan.model.PermanentEmployee"/>

</session-factory>

</hibernate-configuration>

## Testing Single Table Strategy

Let us define Application.java having a main method inside it to run the example and see the entries it created in the DB

**Application.java**

public class Application {

public static void main(String[] args) {

createEmployee();

}

public static SessionFactory getSessionFactory() {

Configuration configuration = new Configuration().configure();

StandardServiceRegistryBuilder builder = new StandardServiceRegistryBuilder()

.applySettings(configuration.getProperties());

SessionFactory sessionFactory = configuration

.buildSessionFactory(builder.build());

return sessionFactory;

}

public static void createEmployee() {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Creating Employee\*\*\*\*\*\*\*\*\*\*\*\*\*");

Employee emp = new Employee();

emp.setName("John");

ContractEmployee contEmp = new ContractEmployee();

contEmp.setName("Mike");

contEmp.setType("CONTRACT");

PermanentEmployee perEmp = new PermanentEmployee();

perEmp.setName("Jordan");

perEmp.setType("PERMANENT");

Session session = getSessionFactory().openSession();

session.beginTransaction();

session.save(emp);

session.save(contEmp);

session.save(perEmp);

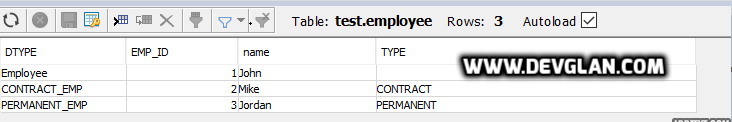
session.getTransaction().commit();

session.close();

}

}

If you run above class a java application, then you can see entries created in the DB. Since, we did not define any specific value for DTYPE column in Employee.java, by default hibernate named it as entity class name.

[](https://imgur.com/vGU6aZV)

## Advantages of Single Table Strategy

Simplest to implement.

Only one table to deal with.

Performance wise better than all strategies because no joins or sub-selects need to be performed.

## Disadvantages of Single Table Strategy

Most of the column of table are nullable so the NOT NULL constraint cannot be applied.

Tables are not normalized.