**Hibernate**

*What is Hibernate ORM?*

Object Relational Mapping tool for Java (Converts data between incompatible "type-systems" in object-oriented programming languages.

Java Object - - - Hibernate - - - DB

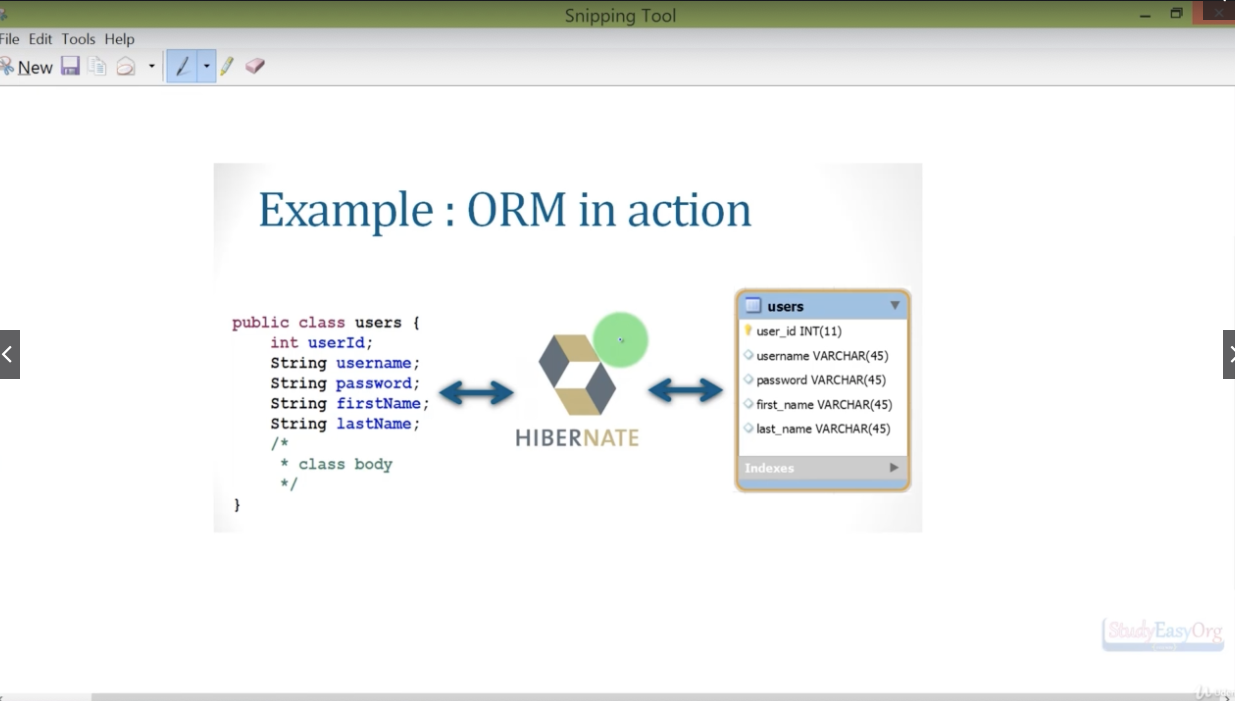
Database Independent, Handles low level SQL, Minimizes JDBC code in our App.

*Session Factory*

Creates Session Objects, Mostly created during application startup

*Session*

It is an Actual Object used to get Physical connection with a database. Not thread safe.



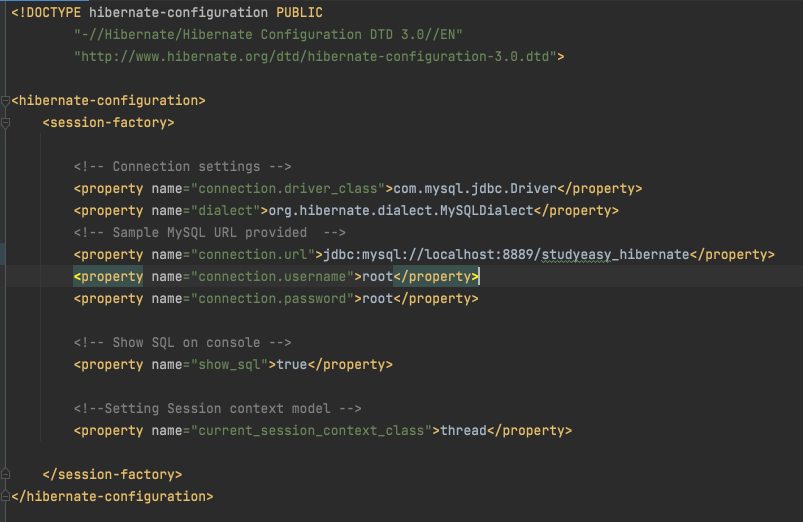
**Before Starting**

Add "Required" folder into a "LIB" created inside our project. There we paste all the content of Required Folder.

We download MySql Connector and paste it there too.

Add everything to the BUILD PATH.(proj structure - modules - dependencies - +Jar).

Create a Hibernate configuration file (hibernate.cfg.xml)



**Entity Class**

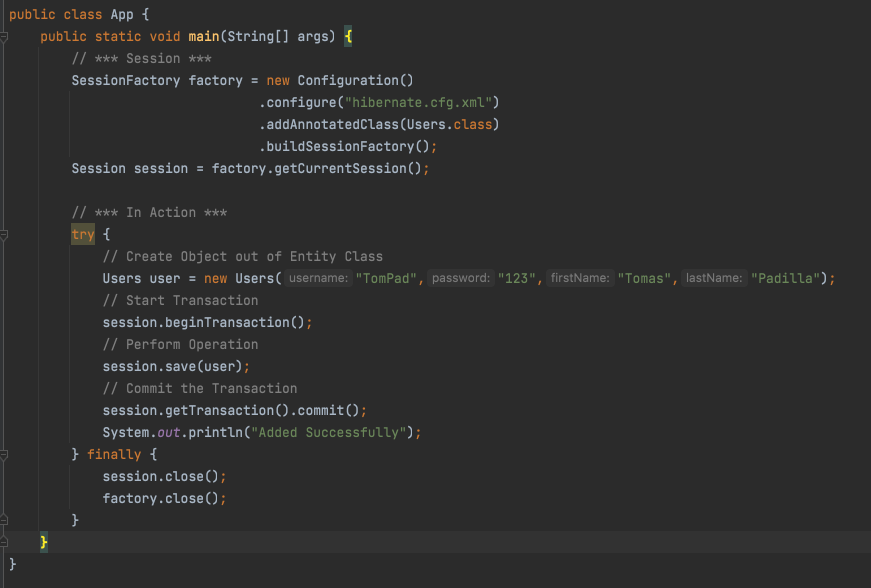
Let's create a Java Class with annotations to specify:

@Entity  
@Table(name="")  
@Id

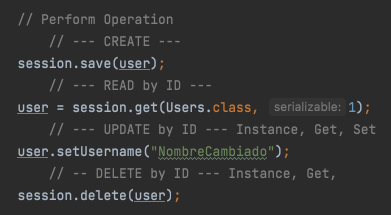
@Column(name="")

**Hibernate in Action**

If the CFG file is correctly made it will start SAVING(user) into the database.

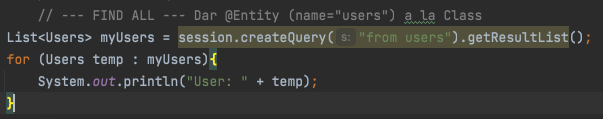


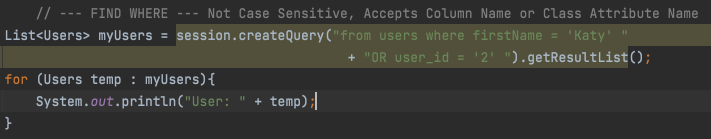
**CRUD**

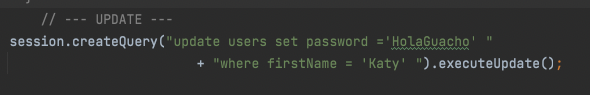


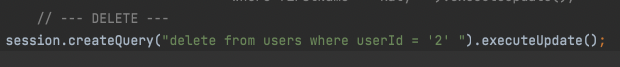
**Using HQL to Query**

Used to create custom Querys to Database. We should add a name to the User Class: @Entity(name="users")









**Integrating Hibernate to JSP-Servlet**

Ver Videos tranqui.

**Spring 5**

We use Spring Tool Suit to create our project.

Basically we have a Maven Project, where we add dependencies to have Spring 5:

* Spring Web
* Spring Core
* Spring Context
* Java Servlet API
* JavaServlet(TM) Specification
* JavaServer Pages(TM) API

For last we must update the Build Path to use JDK 16.

* Order & Export: select the JDK shown
* Libraries: JRE edit then choose "default 16"

We are all set to Learn Spring.

**Dependency Injection**

Normally, whenever we want to use methods of a CLass, we mas instance it vefore invoke them.

*Example: Car myCar = new Swift();*

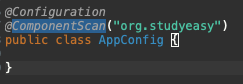
If we want to change the "type" of MyCar, we then need to change the code itself.

*Example: Car myCar = new Corolla();*

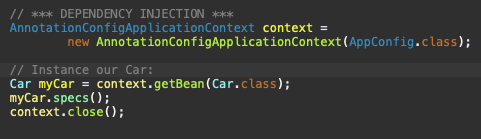
**DI** uses Components to avoid "hardcoding".

We need to tell Spring that we will be using **Components** and we it needs to **Scan** a certain Package.

AppConfig:



Main App:



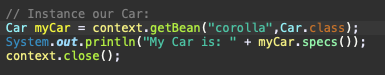
This will Crash, as we need to Specify which are Beans for the Car Class. Beans are jsut an instance/object of a Car Class.

We just need to add a *@Component* annotation in our Class that implements Car Interface.



If we have multiple Types of cars we must specify the name of each Bean , as to implement it to our Car. By **Default** it will take the name of the component, transformed to lowerCase.



**Inversion of Control**

Normally, we. have a Class that we can make an Instance out of it. This way we create Objects "by hand" , that have some Attributes and Methods.

*Class Test{ void display(){} }*

*Test test = new Test();*

*t.display();*

If we like to have a Component based programming, where we can remove, update, use any object as we need, we have a Spring solution: **Inversion of Control.**

We give the Control to Spring, rather than having it ourselves.

Bean Factory: Is where Spring keeps all the objects it creates.

getBean(): Will extract an Object from the Bean Factory. Saving us the time to Instance it (new keyword).

**Autowire**

Using IOC (inversion of control), normally, if we have nested objects, we would need to instanciate the nested one "by hand".

*Car has an attribute Engine engine = new Engine();*

It is a **bad practice** to give only "Partial" control if we use IOC.

Solution: Engine will be a component, Car will have the Attribute Engine "Autowired". **No need to instanciate it.**

One of the most important things about Autowiring is using correct Error Handling, proper ConsoleLogs.

**Scenarios**

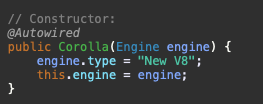
Not Using Engine:

If we do not require an object, Spring is smart to not waste time on this, not creating the object.

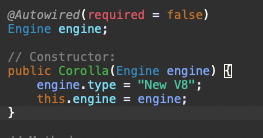


Need to use Constructor for Corolla:

Autowire will pass the object Engine to the constructor, allowing us to pass the values.



This will also work if we set Autowire to the Object itself. Because it affects the object.



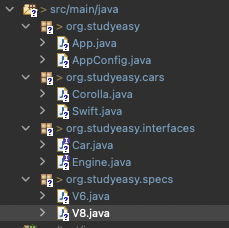
**Qualifier Annotation:**

So far: We know if we use Autowire, Spring will instance the Object, allowing us to acces it's attributes and methods.

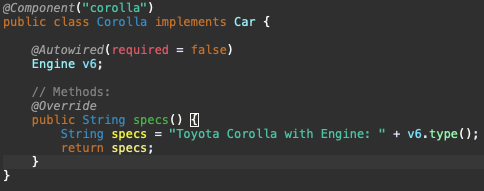
What if we have something more ambiguous?

Engine Interface, 2 more Classes that implements Engine.

It will crash, since we do not specify which Engine is our Corolla using.



But if we name our Engine something similar to the Class, it will be instanciated by Spring and we can access all it's properties:

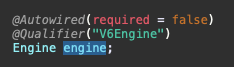




This is good, but not the best. We are **STUCK** to use something similar to the Class.

Therefore we use  **QUALIFIER.**

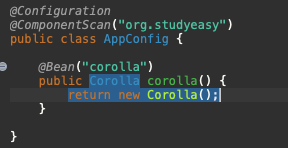
****

****

**Beans**

Components are outdated, we now use Beans Approach.

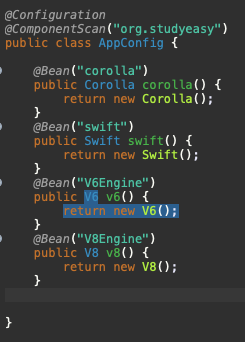
For this we will use our AppConfig.



When using a *@Bean* there is no need anymore to use *@Component.* We remove the Component and the Import.



This approach is more debug friendly, best practice.



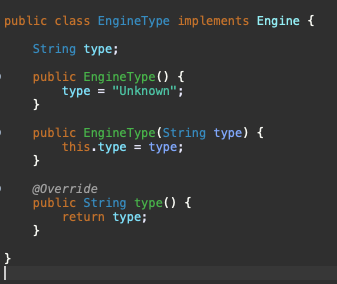


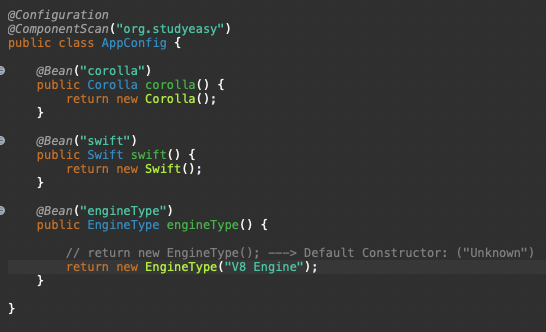
**Constructor Injection:**

We still are "hardcoding" the Engine into each Object / Class.

*Ex: Corolla -> Engine engine -> we passed the V6Engine*

Now we can generate a default value, and a constructor. Allowing us to construct immediately or leave the default values.





By using this, we have all the configs available inside our AppConfig, being easy to mantain, debug and change.

**Spring 5 MVC**