# Exercise: Creating Custom ORM Part 3

This document defines the exercise assignments for the ["DB Frameworks" course @ Software](https://softuni.bg/courses/databases-advanced-hibernate) University.

We have achieved an implementation of basic **CRUD**(CREATE, READ, UPDATE, DELETE) operations and mapping of the retrieved rows from the ResultSet to java objects.

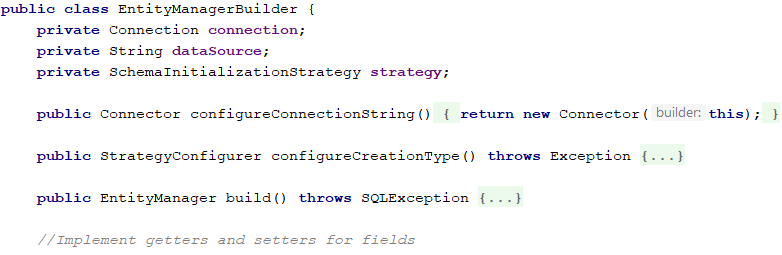
Due our current logic of the **alter** and **create** methods, changes in the database are made **only if we try to persist an entity**. And not only this, but the database modification operations are done one model at a time, **which is a potential threat of a database that is not up to date.**

Thus, we have to achieve a database actualization **on project build**. Our ORM must **update existing** **entity tables** or **create a database, if such is not present**, **defined by implemented models in the application.**

## Building the EntityManager

One approach is **instantiation** of the EntityManager with information that will define future actions. Very convenient and an easy-to-do way is an implementation of the [Builder pattern](https://en.wikipedia.org/wiki/Builder_pattern)– an object that will build our EntityManager and assign its behavior..

We can do that by creating such object:



Take a close look at the configureConnectionString() method.

We will no longer use the Connector class directly. Therefore we have to do some changes there as well:

1. Extend its functionality so that custom adapter, driver, host, port are used in the createConnection() method. You can do that by setting them as fields in the Connector class, along with corresponding **getters** and **setters**
2. The class(EntityManagerBuilder) should set the **EntityManager’s connection,** not instantiate such globally and most importantly **make a connection only to server instance**. What is the advantage of this?

* The Connector class is instantiated **once**, only when creation of a connection is needed.
* The database is not strictly set, which will allow us work with other db’s and also drop it

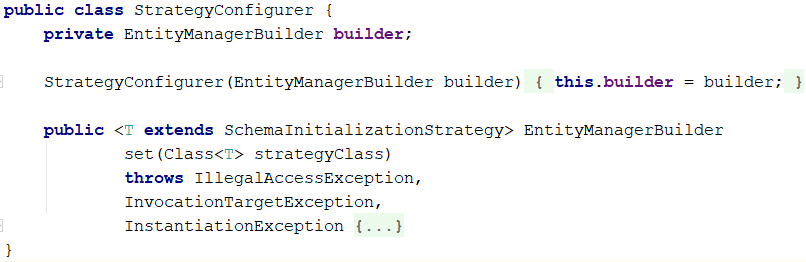
What mistake can we make if in a **bigger**, **complex** **application**, changes to the Connector are made? Let’s say we’ve found insecurities or bugs in the class. A natural way to overcome them is to fix them directly. This is actually **wrong**, since the class is probably used in many other methods and classes, that might not be implemented by us also, which means fixing those bugs will make us **rewrite its usage everywhere**. In this case, we do a [deprecation](https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/deprecation.html) of the class and create a new one that is going to be further used. Such deprecated class in the java language is [Date](https://docs.oracle.com/javase/8/docs/api/java/util/Date.html).

For now, we will not deprecate our Connector, only remove its usage in the Main class and modify its createConnection() method.

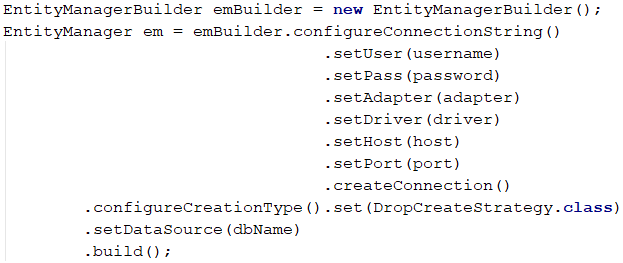
Second important thing is the appearance of a Strategy for the creation type. It will provide us information about our two options – to **create a new database in the server instance**, or **modificate already existing one**. It will be responsible for updating the current state of our datasource.



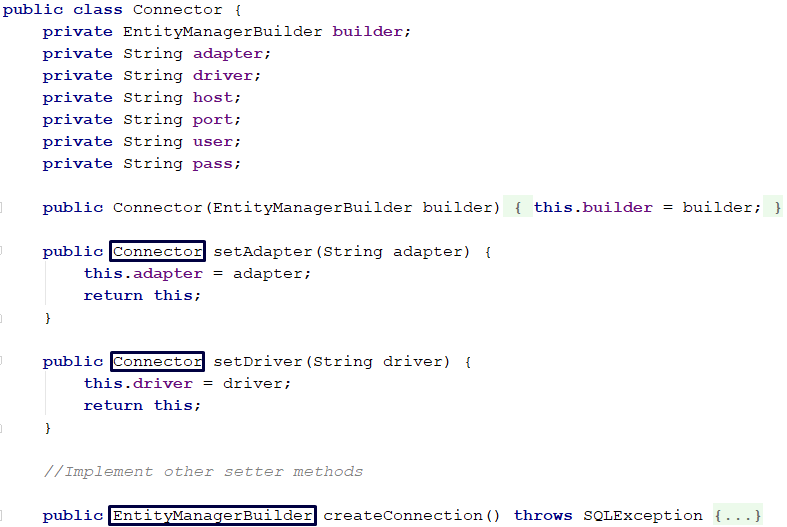
A StrategySetter class is responsible for assigning the EntityManager actions which to perform on build:



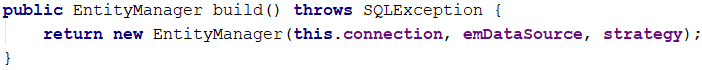
What will we achieve to this point? **Easy** and **readable** instantiation of the EntityManager:



Notice the **return type** of all setter methods – all of them return an object Connector or EntityManager so we can **chain them in instantiation**:



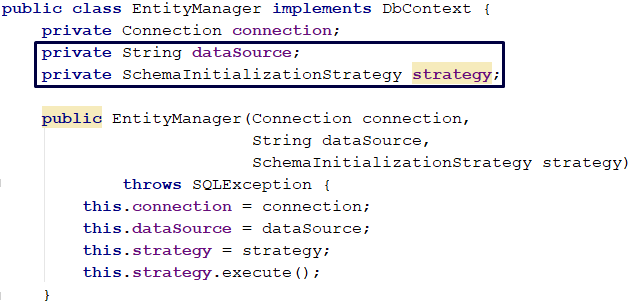
In result, the EntityManagerBuilder produces an EntityManager instance that has a connection, data source name and strategy **pre-set**, done by the **build()** method:



## Encapsulating logic in strategies

If we want to achieve a little application that keeps our database updated according to our models’ structure, we have to set up some rules for the EntityManager.

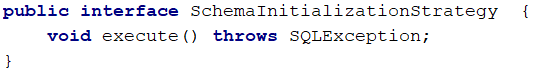
Naturally, his implementation will go through some changes too. To start, add proper **fields for his data source’s name and strategy**, that will be executed on creation in the constructor:



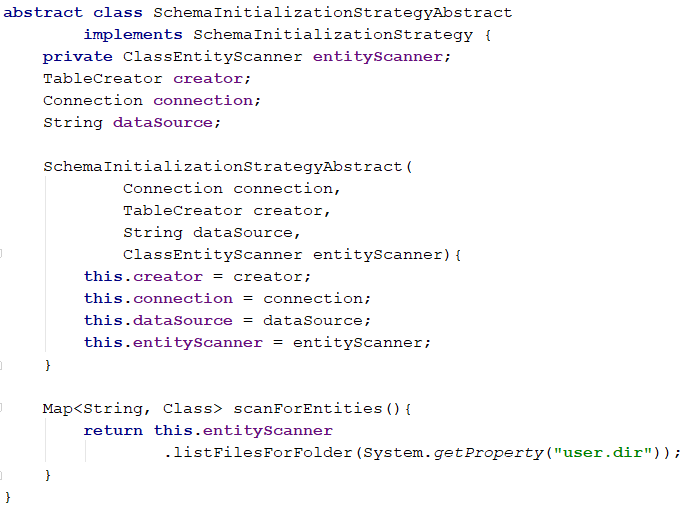
As you probably have considered already, all of our **create** and **alter** logic will be moved in single classes(**Strategies**) implementing the [Strategy pattern](https://en.wikipedia.org/wiki/Strategy_pattern).

### Strategies abstraction

Create an inteface SchemaInitializationStrategy:



It will be implemented by the following abstract class SchemaInitializationStrategyImpl:



Our abstract class will set up some rules and properties for the strategies we are about to implement. Each strategy will:

1. Have a TableCreator class where we will hold our logic for the table creation – retrieval of database fields, tables, mapping types and so on
2. Receive a data source name, in order to work with other schemes in the future
3. Have the ability to **scan the project for entity models so the actualization of the database is easily done**

**NOTE: Start splitting your project into packages and subpackages to organize it more clearly.**

##### Some hints for the project scanning:

1. Start recursive search for entity models in the project directory(you can get it by using the System.getProperty(“user.dir”) path)
   * NOTE: Models are all java classes annotated with the Entity annotation
2. Check if found File is a single file or directory
   * If its a single file(**contains Java Class**) try to instantiate it
     + **HINT**: Replace “\” symbols in it’s path with dots(“.”) and try to instantiate a class with current path. If you don’t succeed cut parts of the path by dots and try again. If you reach an empty path, just continue to next file.
   * On success check if its an entity model and keep it for further use in the application
   * Another option is to be a directory, then recursively continue looking in its subdirectories

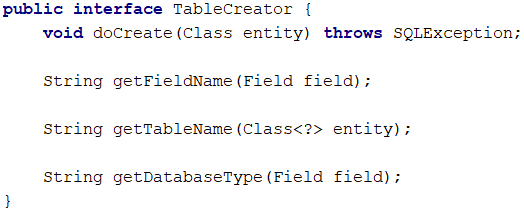
#### DropCreateStrategy

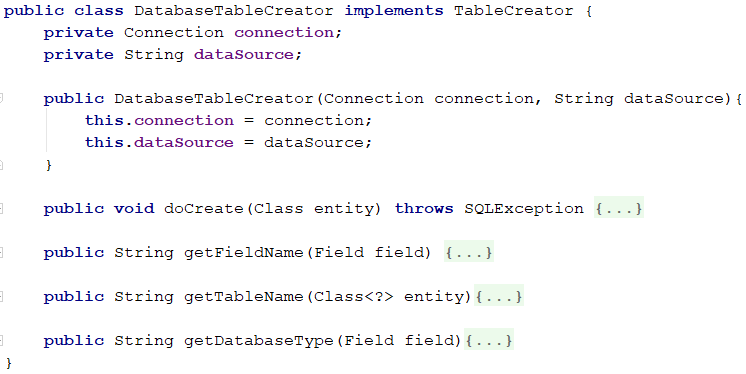
The first implementation of the strategies concept will be the drop-create strategy.

Consider the following case: you test your application and want to make sure the database is always up to date and meets entity models’ architecture 100%. You don’t mind having a fresh empty database everytime. Thus the strategy should drop current data source and build a whole new schema.

To this moment we have written some create logic in the EntityManager. Its time to encapsulate it in other classes and let it do **only CRUD operations**.

Create a TableCreator interface and a DatabaseTableCreator object, holding the create logic:



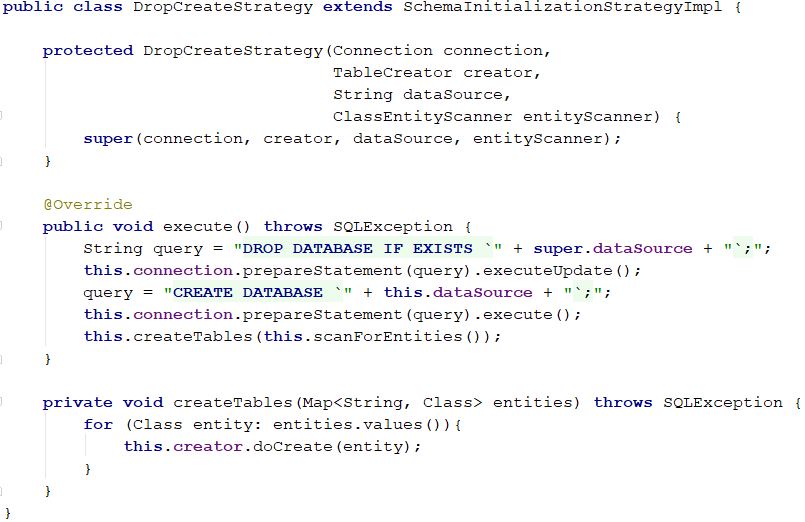


A TableCreator is instantiated on strategy creation in the StrategyConfigurer class. Notice that methods like doCreate, get{Field/Table}Name and getDatabaseType are moved in our new implementation now.

The creator class is responsible for one thing – to create a table via the Connection to the server instance.

What do we achieve by setting an interface and an implementation of such class? We can easily test actions towards the database by [mocking](https://en.wikipedia.org/wiki/Mock_object) classes.

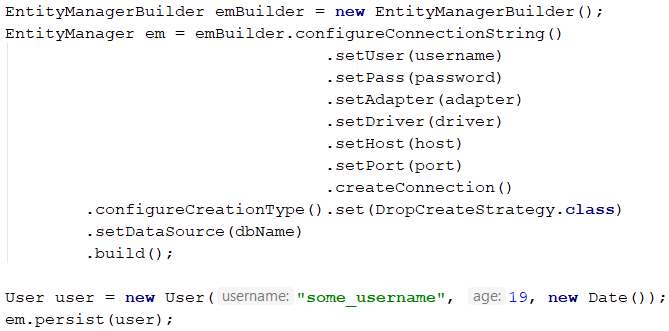
Back to our drop-create strategy creation:



We have extended our abstract implementation of SchemaIntializationStrategy. What we do is simply:

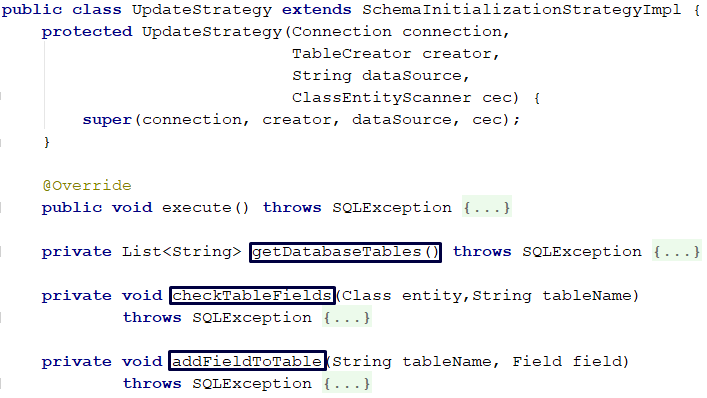
1. Drop the database if it exists in current instance to start fresh
2. Create a new one
3. Use the implemented ClassEntityScanner object to get all entity models
4. For each entity model, create a table holding its fields and data types

Test the strategy by creating an EntityManager with CreateStrategy(in the Main class):



### UpdateStrategy

Very similar to the CreateStrategy implementation, start by extending the SchemaInitializationStrategyImpl class:



Notice all of our alter logic – **alter** and **helper** methods, will be encapsulated here.

The realization of the strategy logic is up to you. Consider the following cases:

1. **Database is present but an entity table might be missing**
   * In this case, we have to create such that meets corresponding entity model – we can use the TableCreator that every strategy has
2. **Entity table’s fields might be missing** - some updates on already existing table should be made
   * Get **missing declared fields from the model** and update the table with corresponding names(from the Column annotation) and data types

Test the strategy by adding new fields to some Entity annotated class that is already persisted once in the database nd setting a UpdateStrategy.class to the EntityManager.

## Conclusion

**What did we achieve?**

1. We can now create an EntityManager that will take actions to update or create a database on instantiation. We can rely on a data source that is up to date with our implemented models on application level. Previously, we were doing that only when persisting entities.
2. We no longer rely on a manually set database – a new one can be created by given name. This is done by initializing a Connection not to specific data source, but to server instance.
3. We implemented a **Builder** and **Strategy pattern** that build EntityManager with strategies very flexible.
4. We encapsulated our create and alter logic into separate classes(Strategies and TableCreator). In result, each class does one thing that is responsible for.
5. We make sure the data source contains all needed tables by scanning the project. This allows us to create missing tables in the data source using the code-first approach.