Contents

* Spring Data JPA
* Angular
* Spring REST
* Spring Microservices
* AWS

Spring Data JPA

Pre-requisites:

* SQL
* JDBC
* ORM/Hibernate/JPA
* Spring Boot

Drawbacks of JDBC

1. You need to write code to create the connection instance
2. You don’t follow any standards to maintain the datasource informations, because you can keep the datasource informations either in Java file/Properties/XML and so on.
3. Most of the checked exceptions should be handled for every operation’s
4. SQL queries are database dependent, the sql queries you write in the JDBC may not work in all the database
5. You write too much code, only to perform type conversion from Java to SQL and vice versa
6. Lot of boilerplate code are written in JDBC.
7. Queries will become more complex if the operations are complex, ex: join queries, sub-queries

ORM (Object Relational Mapping) Framework:

This overcomes the drawback of JDBC, ORM framework will map java objects to the database tables with some predefined functions provided by ORM framework, it avoids lot of things like

1. Type conversion is not required i.e., Java to SQL & vice versa
2. Follows the standard, i.e., datasource informations must be kept in the configuration file (XML)
3. Queries are database independent and in ORM you can avoid writing queries
4. It can create connection for you which you can use it
5. You don’t have to handle any checked exceptions
6. Boiler plate codes are avoided
7. Complex operations can be done with simple code like complex joins can be done with simple annotations

ORM is a specification of JPA (Java Persistence API) i.e., standard, there are many organizations provided the implementation to the ORM

Some of the ORM frameworks are:

1. Hibernate (red-hat)
2. JPA (sun microsystem)
3. Toplink
4. iBatis

Out of the above ORM frameworks, hibernate is the one which is more widely used

Interacting with the database using Hibernate

Steps:

1. you need to create maven project
2. you need to add the hibernate dependencies
3. you need to add the jdbc driver dependencies
4. you need to configure an xml file that will have the datasource information’s, by default the configuration name hibernate looks for is hibernate.cfg.xml
5. you need to create entity classes which will have tables & columns mappings through annotations
   1. @Entity, @Table, @Column, @Id

Note: Primary key is mandatory

Step1: Creating the maven project

Step2: Adding hibernate & mysql-connector dependencies



pom.xml



hibernate.cfg.xml



Create an employee table in mysql



Creating entity class & mentioning that class in the <mapping> tag of hibernate.cfg.xml

Step 1:

Employee.java



Source Code:

**package** com.cognizant;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

**public** **class** Employee {

@Id

@Column(name = "id")

**private** **int** empId;

**private** String name; // @Column is not required because variable & column names are same

**private** **double** salary; // @Column is not required because variable & column names are same

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

}

Step 2: Mention the entity class in the hibernate.cfg.xml



Step 3:

Performing CRUD operations on the entity.

You will use following instances to perform the CRUD operations

* Configuration cfg = new Configuration();

cfg.configure(); // default chooses hibernate.cfg.xml

cfg.configure(“xml file”);

* SessionFactory factory = cfg.buildSessionFactory();
* Session session = factory.openSession();
* session.save(object1);
* session.save(object2)
* session.get(classname.class, primary\_key)

You will get SSL exception so use below url in the hibernate.cfg.xml



HibernateDemo.java

**package** com.cognizant;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.cfg.Configuration;

**public** **class** HibernateDemo {

**public** **static** **void** main(String[] args) {

// 1st step is to create Configuration object that loads the xml configuration file

Configuration cfg = **new** Configuration();

cfg.configure();

// 2nd step is to create SessionFactory

SessionFactory factory = cfg.buildSessionFactory();

// 3rd step is to create Session

Session session = factory.openSession();

Employee employee = **new** Employee();

employee.setEmpId(200);

employee.setName("Bruce");

employee.setSalary(34200);

// 4th step is to save the object but you must begin the transaction

Transaction tx = session.beginTransaction();

session.save(employee);

tx.commit();

session.close();

factory.close();

System.***out***.println("DONE!");

}

}

HibernateDemoToRetrieve.java

**package** com.cognizant;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**public** **class** HibernateDemoToRetrieve {

**public** **static** **void** main(String[] args) {

// 1st step is to create Configuration object that loads the xml configuration file

Configuration cfg = **new** Configuration();

cfg.configure();

// 2nd step is to create SessionFactory

SessionFactory factory = cfg.buildSessionFactory();

// 3rd step is to create Session

Session session = factory.openSession();

Employee emp = session.get(Employee.**class**, 100);

System.***out***.println("Id: "+emp.getEmpId()+", Name: "+emp.getName()+", Salary: "+emp.getSalary());

session.close();

factory.close();

System.***out***.println("DONE!");

}

}

Criteria object in hibernate:

It allows you to fetch all the records in a Collection and can also apply some condition while fetching all the records

To create criteria you will use

Criteria crt = session.createCriteria(EntityClass.class);

List list = crt.list();

Exercise:

Create a menu driven program which shows following options

1: Insert

2: Retrieve by Id

3: Display All

4: Delete by Id

5: Update the salary based on Id

6: Exit

Note: Perform all the CRUD operations in a separate class but create this menu in main class

Note: Reuse the Session Factory instance in separate class

Applying the conditions in the criteria

In criteria you can apply lot of conditions like you apply in sql query.

Criteria crt = session.createCriteria(Employee.class)

crt.add(Restrictions.eq(“variableName”, value));

variableName is the property that maps to columns



Output



Till now we didn’t write any queries, but hibernate/ORM supports passing queries also, the queries write in ORM is called as JPQL/HQL

JPQL: Java Persistence Query Language

HQL: Hibernate Query Language, it is same as JPQL.

In JPQL/HQL you write queries for entities not for the tables.

Ex:

from com.cognizant.Employee : means you get all the entities

select emp from com.cognizant.Employee emp: emp is an alias to access entity property

Some of the conditions also you can write with that alias.

Select emp from com.cognizant.Employee emp

where emp.empId = 100

Above query will get employee entity for the employee id 100

The above query can ignore the package names if the classes don’t have any conflicts

i.e.,

select emp from Employee emp where emp.empId = 100;

Here emp refers to the object completely, however you can also access only few properties

Select emp.empId, emp.name from Employee emp;

Now you will get only empId and name but not salary



Output:



You can also get only few properties instead of getting whole entity



Here select e.empId, e.name from Employee e will only have part of the entity, hence it is going to create an object of type Object that will have 2 properties of Object type(id & name), which is why you will typecast to Object[].

Spring Boot:

It allows you to quickly configure the spring application, it gives you an environment where you can simplify the development of the spring application.

Without spring boot, you do following steps

1. You configure xml file that will have bean configurations
2. You configure xml file to perform component scanning
3. You configure the dependencies of the beans in xml file
4. You add required libraries for spring core, spring mvc, spring orm, spring security and etc
5. If it’s a webapplication you
   1. configure the server
   2. configure the front controller
   3. configure the spring configure file and etc

With spring you can avoid all the above steps

It provides you the autoconfiguration feature where the spring beans are automatically configured without xml file

It provides you the embedded server if you want to develop web applications, it means you don’t have to configure the server

It does the configuration based on the library you add in the project

To perform these auto-configurations spring boot provides lot of starter libraries like:-

1. spring boot web starter
2. spring boot jpa starter
3. spring boot actuator starter
4. spring boot security starter and so on

If you add any starter library of spring boot then all the configurations required will be done by the library the only thing you must do is you should use @SpringBootApplication on top of the class

How to create spring boot application

1. Create maven project and manually add the library from maven repository
2. Use Spring Initializr where you will get UI to add the libraries and it also gives you the project structure required for spring boot applications

Below is the project we are creating from spring initializr



Below is the project structure you get once you import this project in eclipse



@SpringBootApplication: this takes care performing autoconfigurations like:

* Component scanning starts from the package where you have @SpringBootApplication, you can have all the components inside com.cognizant or sub-package of it
* It configures DispatcherServlet, Server if you have added spring boot starter web library
* It configures all the dependency injection required for the application depending on the library i.e., Supplying Datasoruce to the HibernateTemplate/JdbcTemplate to perform database activities if you have added spring boot starter jpa

Without Spring Boot  
EmployeeService can be maintained by spring container in two ways

1. <bean> having EmployeeService in the xml file
2. @Service on top the EmployeeService, but you must <component-scan> in the xml file.

How to get the EmployeeService instance

1. @Autowired on top of the EmployeeService field
2. EmployeeService you can also get through getBean() method

Note: getBean() is present inside ApplicationContext, we will create ApplicationContext only if we are using console based program, if we are using web based program no need of creating ApplicationContext

In Console based application

ApplicationContext ctx = new ClassPathXmlApplicationContext(“xmlfile”);

In Web based application

ApplicationContext is created by FrontController, so you can directly auto-wire the beans or get the beans

With Spring Boot

EmployeeService instance is maintained by spring container when it has any of the @Component type annotation like, @Component, @Configuration, @Service, @Repository, @Controller, @RestController

How to get the EmployeeService instance

1. @Autowired on top of the EmployeeService field
2. EmployeeService you can also get through getBean() method

Note: getBean() is a method present in ApplicationContext, however in spring boot you will get the application context object through command line runner which is created using @Bean

EmployeeService.java



Note: This instance will be maintained by the spring container because of component scan and you can get this object through application context from the command line runner when you are using console-based application.

SpringBootDemoApplication.java



Note: ApplicationContext object is supplied by the spring boot application so that you can get all the instances using getBean from the ApplicationContext.

Output:



Note: if you modify the code it wouldn’t be reflected because server will not be having idea about the changes, for that reason you can add another library called Dev tools, that will provide auto-reload options for the server if any changes done in the code

Adding Devtools library



Exercise:

Create one class EmployeeDao and inside that create one test method and autowire the employee dao in the EmployeeService and call test method of EmployeeDao from testEmp method of EmployeeService.

Spring Data JPA

You can simplify performing operations on the database with the predefined interfaces, spring boot will take care of implementing methods for the interfaces.

Spring Data Jpa gives you some interfaces that can perform all the database operations, you can just inherit that interface to get the functionalities, but you don’t have to implement the interface.

Some of the interfaces are:

CrudRepository<T, ID>: it has methods to perform insert, update, delete and retrieve

JpaRepository<T, ID>: it extends CrudRepository + it has methods to perform sorting and pagination.

Note: You don’t have to implement the interface but you just extend the interface the spring boot will take care implementing the interfaces using ORM features.

Adding spring data jpa library and also the mysql library

pom.xml



Once you add spring data jpa library your connection will be automatically established based on the properties you have configured in the application.properties

CrudRepository & JpaRepository has many method to perform crud operation some of them are:

* T save(T t)
* List<T> findAll()
* T findById(ID id)
* void delete(ID id)

Steps:

Firstly me must configure the application.properties to mention datasource information

Secondly we must have entity class mapping to appropriate table

Thirdly we must create an interface which is our own repository that extends either CrudRepository or JpaRepository

Fourthly we can tell spring to inject the object to our repository to perform the operations i.e, in Servie we use @Autowired

Note: Spring Boot has provided some link which has all the possible properties with description you can mention in the application.properties



Modify the application.properties



Create the entity class for the appropriate table

Employee.java



Creating our own repository



Now you can create a reference of EmployeeRepository in the EmployeeService and call any methods of EmployeeRepository

JpaRepository<T, ID> gives you lot of methods to call:

1. List<T> findAll()
2. T save(T)
3. Optional<T> findById(ID)
4. void count()
5. void deleteById(ID)
6. void deleteAll()

EmployeeService.java



Now you can call the fetchAllEmploees from comandLineRuner



How to store the data in spring data jpa

save(object) returns object that is saved

How to retrieve the data by id

findById(ID) returns Optional so that you can get the entity or null

How to delete the data by id

delete(ID) returns void

Apart from this you can also write custom queries in the repository on top of the methods so that if you call that method the query written on top would be executed

Ex:

@Query(“select e from Employee e where e.name = ?1”)

List<Employee> getEmployeesByName(String name);

Now you don’t have to implement the getEmployeesByName instead you just call the method you will get the List<Employee> based on the name you pass

Storing the employee instance

EmployeeService.java



Main class



Assignment is:

1: insert

2: fetch employee by id

3: fetch all employees

4: delete by id

5: update salary based on id

6: exit

Loop until you enter 6

How to create our own methods and perform operations:



Now you can call these methods from EmployeeService



Call from commandLineRunner



List of class-level, method-level, field level annotations

|  |  |  |
| --- | --- | --- |
| Class level | Method level | Field level |
| @Component | @Bean | @Value |
| @Service | @RequestMapping | @NotNull |
| @Controller | @Autowired | @Autowired |
| @RestController | @GetMapping | @Qualifier |
| @Repository | @PostMapping | @OneToOne |
| @Configuration | @PutMapping | @OneToMany |
| @SpringBootApplication | @DeleteMapping | @ManyTOMany |
| @Import | @OneToOne |  |
| @ImportResource | @OneToMany |  |
| @EnableAutoConfiguration | @ManyTOMany |  |
| @ComponentScan |  |  |
| @RequestMapping |  |  |
| @Transactional |  |  |

Performing @OneTo@One, @OneToMany, @ManyToMany, @ManyToOne annotations

These annotations are helpful in joining the entities reduces writing complex join queries these annotations are part of Jpa which can be used in any ORM framework as well in spring data jpa also.

@OneToOne: one entity mapped to one entity

Ex: One Employee has One Address

@OneToMany: one entity mapped to multiple entities

EX: One State has many Cities

@ManyToOne: Many entities mapping to One entity

EX: Many Cities belonging to same state

@ManyTOMany: Many entities mapping to many entities, however it is a combination of one to many in both the tables

EX: many students taking many courses, in that one student enrolled to many courses and same course could be enrolled by many students.

One to One extra



Below are the employees



Below are the address



Create a new project



application.properties

server.port=8081

spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver

spring.datasource.url = jdbc:mysql://localhost:3306/cts\_2021?useSSL=false

spring.datasource.username = root

spring.datasource.password = root

Create the command line runner in Main class



Address entity

**package** com.cognizant.model.entities;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.OneToOne;

**import** javax.persistence.Table;

@Entity

@Table(name = "address")

**public** **class** Address {

@Id

@Column(name = "address\_id")

**private** **int** addressId;

**private** String state;

**private** String city;

@JoinColumn(name = "emp\_id")

@OneToOne

**private** Employee empIdLink;

**public** **int** getAddressId() {

**return** addressId;

}

**public** **void** setAddressId(**int** addressId) {

**this**.addressId = addressId;

}

**public** String getState() {

**return** state;

}

**public** **void** setState(String state) {

**this**.state = state;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

@Override

**public** String toString() {

**return** "Address [addressId=" + addressId + ", state=" + state + ", city=" + city + "]";

}

**public** Employee getEmpIdLink() {

**return** empIdLink;

}

**public** **void** setEmpIdLink(Employee empIdLink) {

**this**.empIdLink = empIdLink;

}

}

Employee.java

**package** com.cognizant.model.entities;

**import** javax.persistence.CascadeType;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.FetchType;

**import** javax.persistence.Id;

**import** javax.persistence.OneToOne;

@Entity

**public** **class** Employee {

@Id

@Column(name = "id")

**private** **int** empId;

**private** String name;

**private** **double** salary;

@OneToOne(cascade = CascadeType.***ALL***, fetch = FetchType.***EAGER***, mappedBy = "empIdLink")

**private** Address address;

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

@Override

**public** String toString() {

**return** "Employee [empId=" + empId + ", name=" + name + ", salary=" + salary + ", address=" + address + "]";

}

**public** Address getAddress() {

**return** address;

}

**public** **void** setAddress(Address address) {

**this**.address = address;

}

}

Create the repository to perform crud operations on employee/address

Note: You can create AddressRepository to perform CRUD on address entity however you don’t have to create this as you are not independently working on address & you can create EmploeeRepository to perform CRUD operation employee entity.

EmployeeRepository.java

**package** com.cognizant.model.dao;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.cognizant.model.entities.Employee;

**public** **interface** EmployeeRepository **extends** JpaRepository<Employee, Integer>{

}

EmployeeService.java

**package** com.cognizant.model.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.cognizant.model.dao.EmployeeRepository;

**import** com.cognizant.model.entities.Employee;

@Service

**public** **class** EmployeeService {

@Autowired

**private** EmployeeRepository employeeDao;

**public** Employee fetchEmployee(**int** id) {

**return** employeeDao.findById(id).get();

}

**public** List<Employee> fetchEmployees() {

**return** employeeDao.findAll();

}

}

Call these methods in commandLineRunner

**package** com.cognizant;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.annotation.Bean;

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.service.EmployeeService;

@SpringBootApplication

**public** **class** SpringDataJpaMappingsApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringDataJpaMappingsApplication.**class**, args);

}

@Bean

**public** CommandLineRunner runner(ApplicationContext context) {

**return** (args) -> {

EmployeeService service = context.getBean(EmployeeService.**class**);

System.***out***.println("------------------ Getting a particular employee --------------------");

Employee e = service.fetchEmployee(700);

System.***out***.println(e);

};

}

}

Exercise:

Try out other types of mapping with different tables

1. One to Many with States & Cities table
2. Many to Many with Course & Students table

Hint: you must have List reference to hold many entities

Implementing One to Many

States table



Cities table



Cities.java

**package** com.cognizant.model.entities;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

**public** **class** Cities {

@Id

@Column(name = "city\_id")

**private** **int** cityId;

@Column(name = "city\_name")

**private** String cityName;

**public** **int** getCityId() {

**return** cityId;

}

**public** **void** setCityId(**int** cityId) {

**this**.cityId = cityId;

}

**public** String getCityName() {

**return** cityName;

}

**public** **void** setCityName(String cityName) {

**this**.cityName = cityName;

}

@Override

**public** String toString() {

**return** "Cities [cityId=" + cityId + ", cityName=" + cityName + "]";

}

}

States.java

**package** com.cognizant.model.entities;

**import** java.util.List;

**import** javax.persistence.CascadeType;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.FetchType;

**import** javax.persistence.Id;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.OneToMany;

@Entity

**public** **class** States {

@Id

@Column(name = "state\_id")

**private** **int** stateId;

@Column(name = "state\_name")

**private** String stateName;

@OneToMany(cascade = CascadeType.***ALL***, fetch = FetchType.***EAGER***)

@JoinColumn(name = "state\_id\_ref")

**private** List<Cities> cities;

**public** **int** getStateId() {

**return** stateId;

}

**public** **void** setStateId(**int** stateId) {

**this**.stateId = stateId;

}

**public** String getStateName() {

**return** stateName;

}

**public** **void** setStateName(String stateName) {

**this**.stateName = stateName;

}

**public** List<Cities> getCities() {

**return** cities;

}

**public** **void** setCities(List<Cities> cities) {

**this**.cities = cities;

}

@Override

**public** String toString() {

**return** "States [stateId=" + stateId + ", stateName=" + stateName + ", cities=" + cities + "]";

}

}

StatesRepository.java

**package** com.cognizant.model.dao;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.cognizant.model.entities.States;

**public** **interface** StatesRepository **extends** JpaRepository<States, Integer> {

}

StatesService.java

**package** com.cognizant.model.service;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.cognizant.model.dao.StatesRepository;

**import** com.cognizant.model.entities.States;

@Service

**public** **class** StatesService {

@Autowired

**private** StatesRepository statesDao;

**public** States fetchState(**int** stateId) {

**return** statesDao.findById(stateId).get();

}

}

CommandLineRunner code

**package** com.cognizant;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.annotation.Bean;

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.entities.States;

**import** com.cognizant.model.service.CustomerService;

**import** com.cognizant.model.service.EmployeeService;

**import** com.cognizant.model.service.StatesService;

@SpringBootApplication

**public** **class** SpringDataJpaMappingsApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringDataJpaMappingsApplication.**class**, args);

}

@Bean

**public** CommandLineRunner runner(ApplicationContext context) {

**return** (args) -> {

// EmployeeService service = context.getBean(EmployeeService.class);

// System.out.println("------------------ Getting a particular employee --------------------");

// Employee e = service.fetchEmployee(200);

// System.out.println(e);

// System.out.println("------------------Getting all the customers--------------------------");

// CustomerService customerService = context.getBean(CustomerService.class);

// customerService.fetchCustomers().forEach(item -> System.out.println(item));

StatesService statesService = context.getBean(StatesService.**class**);

States states = statesService.fetchState(2);

System.***out***.println(states);

};

}

}

Lombok:

It provides you a feature where you can reduce your code to be more concise by not writing certain boiler plate code, i.e,. you can avoid writing setters, getters, constructors, toStirng, equals, hashCode, validations like null check and so on, Lombok provides some annotations using which you can generate these codes without writing it.

Lombok provides some annotations like:

@Data: setters & getters, you can write on top of the class

@Setter & @Getter: It is also for setters & getters but this can be written on top the fields

@ToString:

@NoArgsConstructor, AllArgsConstructor

Official website of Lombok

<https://projectlombok.org/>

pom.xml



Users.java



Now you will get setters & getters for all the fields



We could able to call setters & getters without writing because of Lombok.

If Lombok doesn’t generate setters & getters you can use java -jar <file-name>.jar as below:



Now a wizard will open you need to locate the eclipse installation folder



After you click on install/update you will get complete wizard



Note: At the end update the maven project and check the Lombok features

Using AllArgsConstructor to generate the parameterized constructor



Lombok provides you the feature of null checking, so that you can avoid writing code to handle null values

@NonNull can be used on top of the variable as well in the parameter also.



If you pass null to the name then you will get NullPointerException.

Exercise:

Try to use Lombok features in the spring data jpa entities