Tutorial- Spring-boot JPA

Current Version: 3

Version 1 : Designed by Yueming ZHU (in 2018)

Version 2: Modified by Yueming ZHU (in 2019)

Version 3: Modified by Yueming ZHU (in 2020)

Experiment Objective

- Learn how to build a spring boot project
- Learn how to connect database in a spring boot project
- Learn what is the maven.
- Learn what is dependency and how to import dependency.
- Learn what is Persistency
- Learn how to use Spring data JPA to do simple interaction with database

Software being Used

1. Install JDK

click here to download

2. Database: PostgreSQL

click here to download

You can also use other DBMS instead, but the .yml file about the DB configure part needs to be rewrite accordingly.

3. Install Intellij IDEA (Ultimate)

click here to download

Free student pack

4. API Testing Tool

- Eolinker click here to download
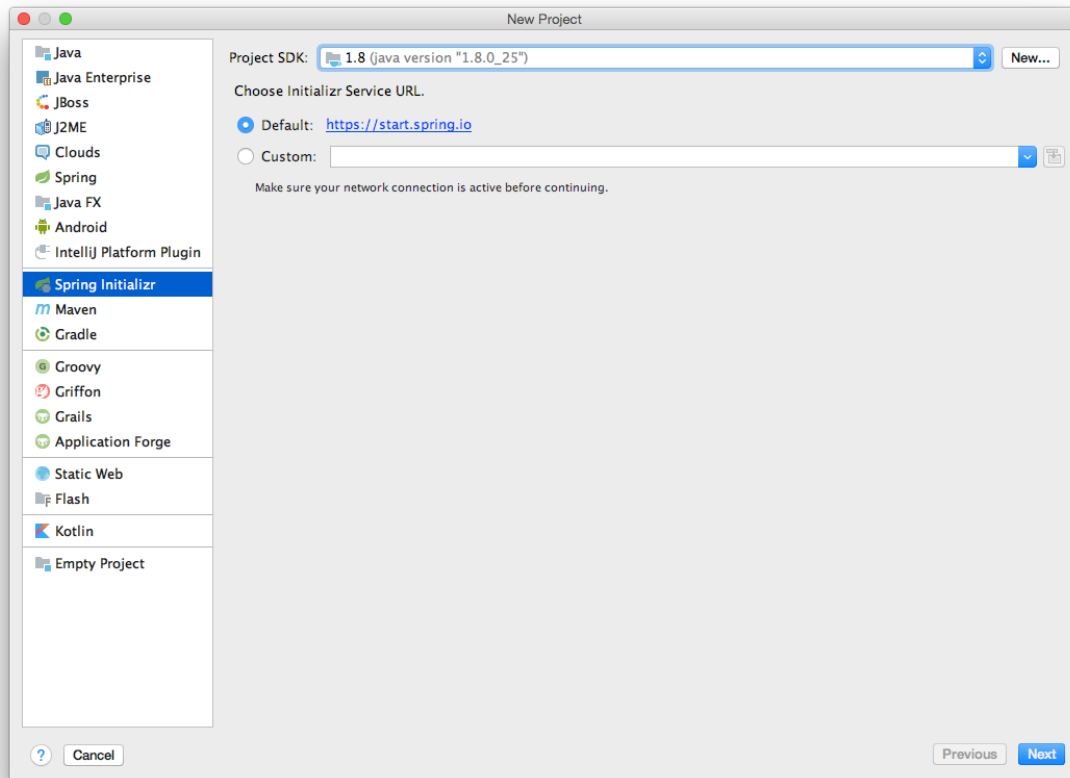- postman click here to download

Part One: Initialize a spring project

Open IntelliJ IDEA, and click Create New Project

1. Initialize

1.1 Initialize by Intellij IDEA

Then press Spring Initializr, and click next. In this section you need to make sure your network connection is active.

Then we can give our project a name `springproject` by changing the name of artifact, and then click next.

Here we need to select "Web" as dependency, and then click next.

1.2 Initialize from official website

If your initialization failed for "start.spring.io", you can try to visit the http://start.spring.io directly.

Then giving a Name as `springproject` and adding `spring web` as dependency.

After that, you can click **File->Open** and find your url for your download file.

2. Project Configure

2.1 Project Structure

And finally, open it by Intellij IDEA and then our spring project has been created. At this moment you need to wait several minutes for downloading resource, after finishing downloading, the catalogue would be shown as follows.
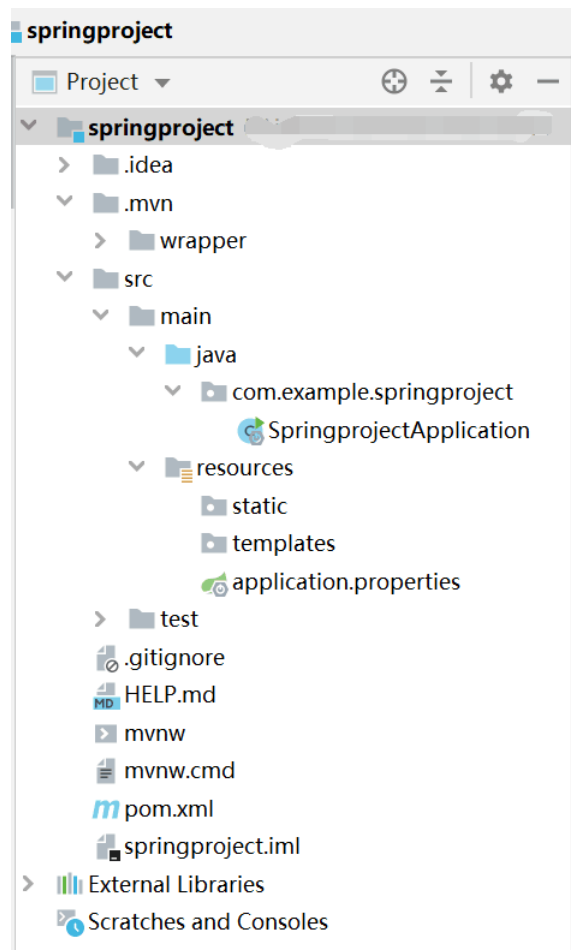
2.2 pom.xml file

In my project, the `pom.xml` file is as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>springproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springproject</name>
    <description>SUSTech OOAD exercise for Spring-boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
```

```xml
                    </dependency>

                    <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                    </dependency>

                </dependencies>

                <build>
                    <plugins>
                        <plugin>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                    </plugins>
                </build>

    </project>
```

If you cannot import repositories successfully, you can try to add following configure into your `<project>` tag of `pom.xml`

```xml
<repositories>
    <repository>
        <id>central</id>
        <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>central</id>
        <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
    </pluginRepository>
</pluginRepositories>
```

2.3 Entrance

The java class `SpringprojectApplication` is the entrance of our program.

`@SpringBootApplication` : It is an annotation of spring, and the function of the annotation is to scan the package, injection classes for our project etc.

`SpringApplication.run()` This method is to start our project.

```
@SpringBootApplication
public class SpringprojectApplication {

    public static void main(String[] args) {

        SpringApplication.run(SpringprojectApplication.class, args);
    }

}
```

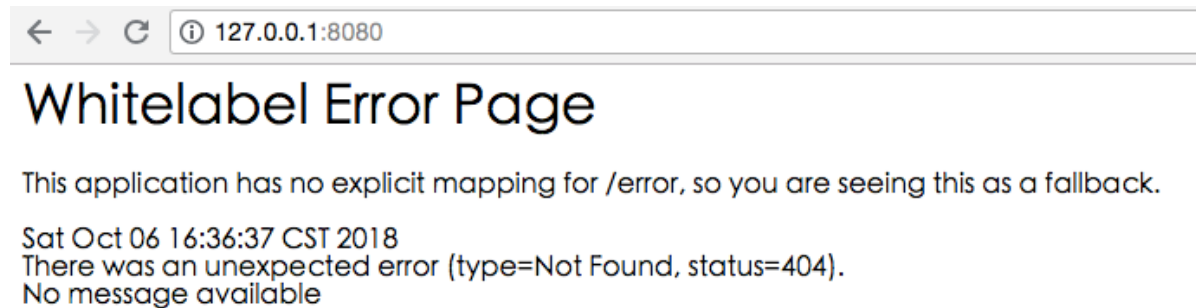Then we click run button to start the server.

In console window, we can find that we use Tomcat as default server and 8080 is the default port.

```
 main] o.s.w.s.handler.SimpleUrlHandlerMapping   : Mapped URL path [/**] onto handler of type
 main] o.s.j.e.a.AnnotationMBeanExporter          : Registering beans for JMX exposure on star
 main] o.s.b.w.embedded.tomcat.TomcatWebServer    : Tomcat started on port(s): 8080 (http) wit
 main] c.e.s.SpringprojectApplication             : Started SpringprojectApplication in 3.554
-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]        : Initializing Spring FrameworkServlet 'disp
-exec-1] o.s.web.servlet.DispatcherServlet         : FrameworkServlet 'dispatcherServlet': init
```

After that, when we open a browser, and input a local url `127.0.0.1:8080` or `localhost:8080`, then the web page will return message as following, which means we have started our spring project successfully

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Oct 06 16:36:37 CST 2018
There was an unexpected error (type=Not Found, status=404).
No message available

3. Let's start our first project

Firstly, create a package named web under the `com.example.springproject`, and then create a java class named `HelloTest` in the web package.

```
package com.example.springproject.web;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class helloTest {
    @RequestMapping("/hello")
    public String hello(){
        return "Hello world";
    }
}
```

Both `@RestController` and @Controller can map the HTML requests (by URL) to specific control methods. For example, if we visit `127.0.0.1:8080/hello`, the method is executed. After that, we restart the server and visit the url `127.0.0.1:8080/hello`, then the result would be as follows.
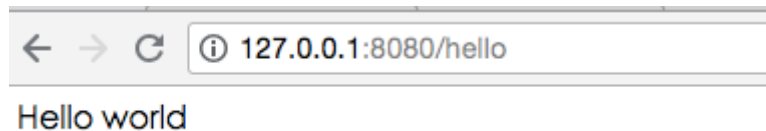
`@RestController` **Annotation**: It is usually interact with a Request API, and the client of which are usually programs, such as JSON or XML.

`@Controller` **Annotation**: It is usually interact with a web page by using model, and the client of which is for human viewers.

`@RequestMapping` **Annotation**: It is the specific path of the class.

Part Two: JPA

JPA (Java Persistence API), is used for mapping Java objects to rational tables in database, which is a kind of the Object Relation Mapping (ORM) and provides standard interface for persistence of entity object without implementation of it. **Hibernate** implements an ORM framework for JPA. JPA is a set of interface specifications, neither an ORM framework, nor a product.

Spring Data JPA further simplifies the implementation of the data access layer based on JPA, which provides a way similar to declarative programming. Developers only need to write a data access interface (Here we call it Repository), and Spring Data JPA can automatically generate implementations based on the naming of the method in interface.

Actually, using JPA can omit the process of creating a table in database manually.

1. Configure Database

1.1 Create your database

In this tutorial we use postgreSQL as DBMS, please make sure that you can connect with postgreSQL server successfully, (you can also use another database, such as mysql).

- Open your terminal window and visit postgreSQL by following command. The first `postgres` just following by `-U` is the original database user, and the second `postgres` is the original database.

  ```
  psql -U postgres -d postgres
  ```

  After that, it will shown as below, which means you have visited the database named `postgres` successfully.

  ```
  postgres=#
  ```

- Create database for our project.

  ```
  create database spring_project encoding utf8;
  ```

- Quit for current database (`postgres`)

  ```
  \q
  ```

- Visit database `spring_project` by following command:

  ```
  psql -U postgres -d spring_project
  ```

  After that, it will shown as below.

  ```
  spring_project=#
  ```

1.2 Set Dependency in pom.xml

Open `pom.xml` and add two dependencies as follows. The one is JPA dependency and the other is the jdbc for postgreSQL. It is a **maven** project, that can provide an easy way to share JARs according to the groupID and artifactID, so that it is convenient for us to import changes. After we adding those two dependencies, you need to click **import Changes**, and then it can download necessary jars automatically.

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
        </dependency>
```

1.3 Create configure file (.yml)

`.yml` file is a kind of configuration file, different from the `.properties` (map struture), it is based on tree structure. In spring boot, we need to use it to configure the access information of database.

Create an .yml file named `application.yml` under the resource document. The default name of the configure file in spring framework is "application".

Then setting arguments as following, and please take care of retract.

```
server:
  port: 8082

spring:
  datasource:
    driver-class-name: org.postgresql.Driver
    url: jdbc:postgresql://localhost:5432/spring_project
    username: ?
    password: ?
  jpa:
    hibernate:
      ddl-auto: create
```

```
    show-sql: true
    properties:
      hibernate:
        temp:
          use_jdbc_metadata_defaults: false

#    It is for mysql
#    driver-class-name: com.mysql.cj.jdbc.Driver
#    url: jdbc:mysql://localhost:3306/spring_project?
useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone=GMT
#    username: ?
#    password: ?
```

2. Basic Introduce of JPA (PurchaseRecord Model)

1. **api**: The primary business methods should be defined as a form of interfaces in this layer. These methods usually interact with the database, so it also acts as a dao layer.
2. **domain**: The entity classes are defined in this layer, so it also can be called as model or entity.
3. **service**: The service classes are defined in this layer.
4. **app**: The RestController classes are defined in this layer, which is for testing the designing results of back-end by using interface testing tools (eolinker or postman).
5. **web**: The controller classes are defined in this layer, which is to interact with the front-end.

Then we need to create five packages including `api`, `domain`, `service`, `app` and `web` under the package `springproject`.
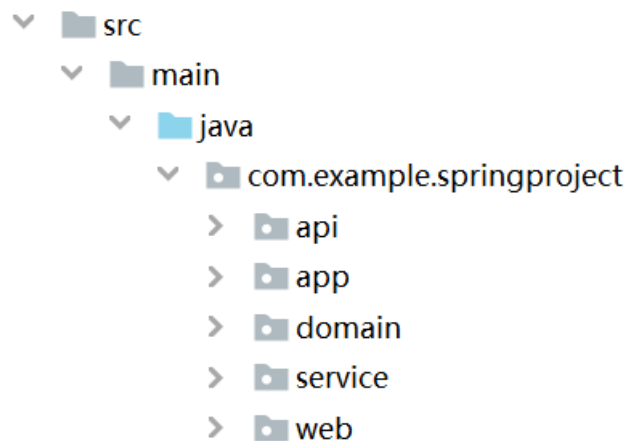
2.1 simple example of data persistence

After that, we need to create an entity class named `PurchaseRecord` in domain layer.

In `PurchaseRecord` class, there are several necessary attributes, which are the important information that need to store in database. Whether we need to create a table in database by ourselves? It's not. Here, we use an mechanism of data persistence to convert java objects to SQL language that can be executed when starting the server. In this section, out purpose is that all attributes in the entity class `PurchaseRecord` is needed to convert to be the corresponding columns in a table in database.

The private attributes in `PurchaseRecord` are:

```java
private String username;
private String date;
private double money;
private int type;
private String description;
```

`@Entity` **Annotation**: Declare a class is an Entity class, the corresponding table of which can be created in database.

`@Id Annotation` : Primary Key

`@GeneratedValue(strategy = GenerationType.IDENTITY)` : Auto increment.

`@NotNull` **Annotation**: The time that validation the attribute should not null is in the entity attribute, which means if the attribute is null, the SQL queries cannot be executed. Different from `@Column(nullable = false)` , the validation time of which is in the interact process with database. See more.

The private fields and their Annotations of `PurchaseRecord` class is as follows, then you need add all **getter and setter** methods of them.

```java
package com.example.springproject.domain;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;


@Entity
public class PurchaseRecord {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @NotNull
    private String username;
    private String date;
    private double money;
    private int type;
    private String description;
    //add getter and setter methods here
}
```

After that, we restart the server and then a table named purchase_record has been created automatically in database.

Here provides a [link](#) that can help to understand more Annotations in JPA Entity Class

2.2 Manipulate the database

- Adding data into table.

  Before we do it by JPA, we need to add some test cases into purchase_record table as follows or insert other rows determined by you.

  ```sql
  insert into purchase_record (username, date, money, type, description)
  values ('Yueming','2019-09-01',50,0,'Meituan');
  insert into purchase_record (username, date, money, type, description)
  values ('Helly','2020-10-10',20,0,'Taobao');
    insert into purchase_record (username, date, money, type, description)
  values ('Bob','2021-11-11',210,0,'Taobao');
  ```

- Change `application.yml` file (create -> update)

  Please open `application.yml` again, and we need to change ddl-auto: `create` by ddl-auto: `update`, otherwise when we start the server again, the purchase_record table would be created again.

  ```yaml
    jpa:
      hibernate:
        ddl-auto: update
  ```

2.3 Build layers of project

- **api package**: Create an interface named `PurchaseRecordRepository`, which need to extends `JpaRepository`.

  ```java
  public interface PurchaseRecordRepository extends
  JpaRepository<PurchaseRecord, Long> {
  }
  ```

- **service package**: Create a class named `PurchaseRecordServiceImpl` and an Interface named `PurchaseRecordService`. In `PurchaseRecordService`, declare an attribute of `PurchaseRecordRepository` type. Don't forget to add those two annotations in it.

```java
public interface PurchaseRecordService {
}
```

```java
@Service
public class PurchaseRecordServiceImpl implements PurchaseRecordService {
    @Autowired
    private PurchaseRecordRepository purchaseRecordRepository;
}
```

- **app package**: Create a class named `PurchaseRecordApp`, in which we need to declare an object of `PurchaseRecordService`. Don't forget to add annotations in it.

```java
@RestController
@RequestMapping("/exer")
public class PurchaseRecordApp {
    @Autowired
    private PurchaseRecordService purchaseRecordService;
}
```

2.4 Using methods defined in Repository

Several methods are defined in repository, and we can invoke them directly by the object of `JpaRepository` (here is purchaseRecordRepository). Don't be surprised that we can use an object without instantiated, because the JPA repository can automatically assemble the object by adding an annotation `@Autowired`.

To interact with database, several operations are necessary such as select all rows, insert into one row, update one row and delete rows. Then we will introduce them one by one.

findAll() select * from...

Adding following code into corresponding Class.

`PurchaseRecordService`

```java
public interface PurchaseRecordService {
    public List<PurchaseRecord> findAll();
}
```

`PurchaseRecordServiceImpl`

```java
@Service
public class PurchaseRecordServiceImpl implements PurchaseRecordService {
    @Autowired
    private PurchaseRecordRepository purchaseRecordRepository;

    @Override
    public List<PurchaseRecord> findAll() {
        return purchaseRecordRepository.findAll();
    }
}
```

`PurchaseRecordApp`:

```java
@RestController
@RequestMapping("/exer")
public class PurchaseRecordApp {
    @Autowired
    private PurchaseRecordService purchaseRecordService;

    @GetMapping("/record")
    public List<PurchaseRecord> findAll(){
        //Fot testing the concrete class
        System.out.println(purchaseRecordService.getClass().getName());
        return purchaseRecordService.findAll();
    }
}
```

Test of findAll

We will introduce the front-end part next week, in this tutorial we use an interface tester: eolinker to test the interaction between controller and server.

https://www.eolinker.com/#/

After we restart the server, open eolinker and input url as `127.0.0.1:8080/exer/record`, because the Annotation `@RequestMapping("/exer")`, defines the upper URL as "/exer", and then for each methods defined in `PurchaseRecordApp`, has its own sub URL by using Annotation `@GetMapping("/record")`, the whole url for findall request is `ip address:Port number/exer/record`

Then select "GET" as a request method because we use the `@GetMapping` annotation for `findall()` method. It will return all information from purchase_record table.

save() insert into ...

Adding following code into corresponding Class.

`PurchaseRecordService`:

```
public PurchaseRecord save(PurchaseRecord purchaseRecord);
```

`PurchaseRecordServiceImpl`:

```
@Override
public PurchaseRecord save(PurchaseRecord purchaseRecord) {
    return purchaseRecordRepository.save(purchaseRecord);
}
```

`PurchaseRecordApp`:

```
@PostMapping("/record")
public PurchaseRecord addOne(PurchaseRecord purchaseRecord){
    return purchaseRecordService.save(purchaseRecord);
}
```

Then restart the server, and we can set attribute in **eolinker**. The name of attribute in request form should be same with the attributes defined in Entity class.

After sending the form, a row has been added into `parcharse_record` table, and the eolinker shows below:

save() update one row

Compare to insert operation, update operation needs to use `@PutMapping` annotation. In addition, update operation needs to pass an id as the unique identifier of the entity object. We use `setId()` method to identify that it is an update operation but not insert.

`PurchaseRecordApp`

```java
@PutMapping("/record")
public PurchaseRecord update(@RequestParam long id,
                            @RequestParam String username,
                            @RequestParam String date,
                            @RequestParam double money,
                            @RequestParam int type,
                            @RequestParam String description){
    PurchaseRecord purchaseRecord=new PurchaseRecord();
    purchaseRecord.setId(id);
    purchaseRecord.setUsername(username);
    purchaseRecord.setDescription(description);
    purchaseRecord.setMoney(money);
    purchaseRecord.setType(type);
    purchaseRecord.setDate(date);
    return purchaseRecordService.save(purchaseRecord);
}
```

In eolinker. Type in data as below:

In addition, update operation can be simplified as below, in which the only request parameter we need provide is the id.

```java
@PutMapping("/record2")
    public PurchaseRecord update(PurchaseRecord purchaseRecord){
        return purchaseRecordService.save(purchaseRecord);
    }
```

deleteById(): Delete one row

Adding following code into corresponding Classes

`PurchaseRecordService`

```
public void deleteById(long id);
```

`PurchaseRecordServiceImpl`

```
@Override
public void deleteById(long id) {
    purchaseRecordRepository.deleteById(id);
}
```

`PurchaseRecordApp`

```
@DeleteMapping("record/{id}")
public void deleteOne(@PathVariable long id){
    purchaseRecordService.deleteById(id);
}
```

| DELETE ▾ | 127.0.0.1:8082/exer/record/4 | 测试用例 ▼ | 测试 [通过本地测试] ▼ |

And do not surprise nothing would return in eolinker, because the return value of the return value of the method `public void deleteOne(@PathVariable long id)` is void. When we change to the get method, it cannot be return the row, the id of which is 4.

3. Many to One in JAP (UserInfo and City Model)

When actually working on a project, there are few cases of an isolated table, but the case of a foreign key connection between tables is very common. In ORM architecture, it can also indicate the presence of foreign key constraints.

We can use `@ManyToOne` annotation to mark the attribute, that can map to be a foreign key of joined table, and `@JoinColumn(name = "xxx")` annotation helps hibernate figure out the name of column in joined table in database. On the other hand, in another associated entity, we can use `@OneToMany(mappedBy = "attribute name")` annotation to mark which attribute in `@ManyToOne` side that is associated of this entity.

3.1 Build Entity class of (UserInfo and City)

In domain package, create two classes named `UserInfo` and `City`. Then add attributes in it.

`UserInfo`

```
@Entity
public class UserInfo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(unique = true)
    private String username;
    private String password;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "city_id")
    private City city;
}
```
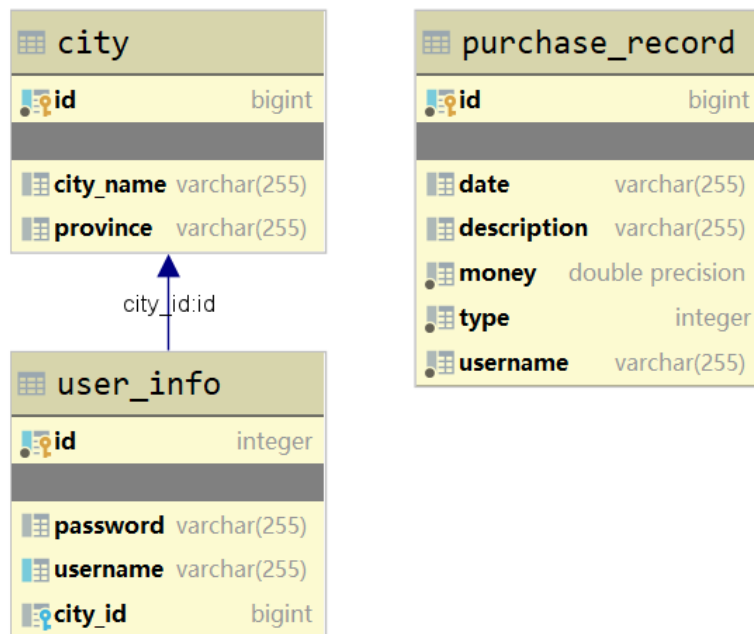
`City`

```
@Entity
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String city_name;
    private String province;
    /**
     * mappedBy="city", city marked the attribute in UserInfo, which can help
     * to make association of JPA
     */
    @OneToMany(mappedBy = "city")
    private List<UserInfo> users = new ArrayList<>();

    public void setUsers(List<UserInfo> users) {
        for (UserInfo user : users
        ) {
            user.setCity(this);
        }
        this.users = users;
    }
}
```

After that, do not forget to add Getter and Setter methods in both two Entity classes.

Finally, restart the server, the two tables are created in database.

3.2 Build layers of City and UserInfo model

api package

create two classes named `CityRepository` and `UserRepository`

```
public interface CityRepository extends JpaRepository<City, Integer> {
}
```

```
public interface UserRepository extends JpaRepository<UserInfo, Integer> {
}
```

service package

create two interface named `CityService` and `` `UserService `` , and then create two implement classes of those two interfaces respectively.

- City

```
@Service
public interface CityService {
}
```

```
@Service
public class CityServiceImpl implements CityService {

    @Autowired
    CityRepository cityRepository;

}
```

- User

```java
public interface UserService {
}
```

```java
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserRepository userRepository;
}
```
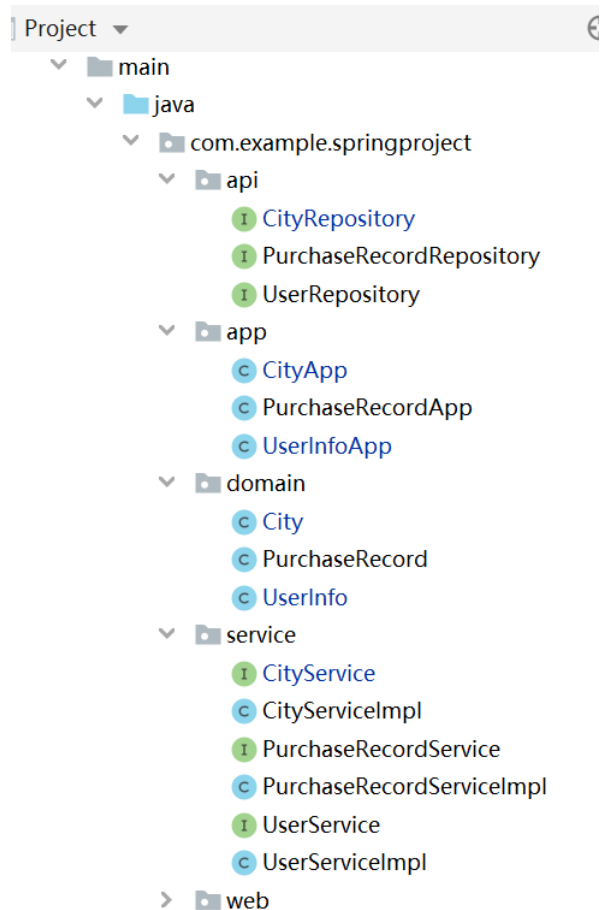
app package

Create two classes named `CityApp` and `UserInfoApp`

```java
@RestController
@RequestMapping("/city")
public class CityApp {
    @Autowired
    CityService cityService;
}
```

```java
@RestController
@RequestMapping("/user")
public class UserInfoApp {
    @Autowired
    private UserService userService;
}
```

After that the structure of the whole project would be shown as graph below:

```
Project ▾                                                    €
  ∨ ■ main
    ∨ ■ java
      ∨ ■ com.example.springproject
        ∨ ■ api
            I CityRepository
            I PurchaseRecordRepository
            I UserRepository
        ∨ ■ app
            c CityApp
            c PurchaseRecordApp
            c UserInfoApp
        ∨ ■ domain
            c City
            c PurchaseRecord
            c UserInfo
        ∨ ■ service
            I CityService
            c CityServiceImpl
            I PurchaseRecordService
            c PurchaseRecordServiceImpl
            I UserService
            c UserServiceImpl
        › ■ web
```

3.3 Design app methods

save city

Adding following code into corresponding Class.

`CityService`

```java
public void saveCity(City city);
```

`CityServiceImpl`

```java
@Override
    public void saveCity(City city) {
        cityRepository.save(city);
    }
```

`CityAPP`

```java
@PostMapping("/addCity")
    public City addCity(City city) {
        cityService.saveCity(city);
        return city;
    }
```

Then try to add some cities here.

save user with foreign key

Adding following code into corresponding Class.

`UserService`

The method `saveWithFk` has two parameters, UserInfo indicates the entity should be inserted into database, and the parameter id, is the id of `City`

```java
public void saveWithFk(UserInfo userInfo, long id);
```

`UserServiceImpl`

`EntityManager` used to interact with persistence context. More specific introduction can be referenced [here](#)

```
@Autowired
private EntityManager entityManager;

@Override
    public void saveWithFk(UserInfo userInfo, long id) {
        City city = entityManager.getReference(City.class, id);
        userInfo.setCity(city);
        userRepository.save(userInfo);
    }
```

In this step, I recommend to created a new package named `domain` , in which we can store other entities that needn't persistent in database.

create classes named `UserWithoutCity` in `bean` package

`UserWithoutCity`

```
public class UserWithoutCity {
    private int id;
    private String username;
    private String password;
    private long city_id;

    public UserWithoutCity(int id, String username, String password, long
city_id) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.city_id = city_id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
```

```
        this.password = password;
    }

    public long getCity_id() {
        return city_id;
    }

    public void setCity_id(long city_id) {
        this.city_id = city_id;
    }
}
```

and then, add a convert method in `UserInfo` to convert `UserInfo` to `UserWithoutCity`

`UserInfo`

```
public UserWithoutCity convertToUserWithoutCity() {
        return new UserWithoutCity(this.id, this.username, this.password,
(this.city == null) ? 0 : this.city.getId());
    }
```

`UserInfoApp`

```
@PostMapping("/addOne")
    public UserWithoutCity addOneUser(UserInfo userInfo, @RequestParam long
city_id) {
        userService.saveWithFk(userInfo, city_id);
        return userInfo.convertToUserWithoutCity();
    }
```

find users by cityId

There are two ways to implements this requirements by JPA

```
select u.* from user_info u join city c on u.city_id=c.id
```

(1) From city

Adding following code into corresponding Class.

`CityRepository`

```
public City findCityById(long id);
```

`CityService`

```
public List<UserInfo> findUsersInCity(long id);
```

`CityServiceImpl`

```
@Override
    public List<UserInfo> findUsersInCity(long id) {
        City city = cityRepository.findCityById(id);
        return city.getUsers();
    }
```

`CityApp`

```java
    @PostMapping("/findAllUser")
    public List<UserWithoutCity> findAllUsersByCityId(@RequestParam long
city_id) {
        return cityService.findUsersInCity(city_id).stream()
        .map(UserInfo::convertToUserWithoutCity).collect(Collectors.toList());
    }
```

Testing:

(2) From user

Adding following code into corresponding Class.

`UserRepository`

```java
    public List<UserInfo> findUserInfosByCity_Id(long id);
```

`UserService`

```java
    public List<UserInfo> findByCity_Id(long id);
```

`UserServiceImpl`

```java
    @Override
    public List<UserInfo> findByCity_Id(long id) {
        return userRepository.findUserInfosByCity_Id(id);
    }
```

`UserInfoApp`

```java
    @PostMapping("/cityId")
    public List<UserWithoutCity> findByCity_Id(@RequestParam long id) {
        return userService.findByCity_Id(id).stream()
        .map(UserInfo::convertToUserWithoutCity).collect(Collectors.toList());
    }
```

Test:

4. Handwriting SQL queries in JPA

The framework cannot implement all SQL statements, especially for some complex SQL queries. In this part, we will given you some example to design SQL statements by handwriting.

For more detailed introduction about `@Query` annotation, you can learn from here

4.1 Example 1

`PurchaseRecordRepository`

```
@Query("select p from PurchaseRecord p where p.username=?1 and p.type=?2")
    List<PurchaseRecord> findByNameAndAndType(String username, int type);
```