

## Tutorial 6: Spring boot (Part 1)

### [Experimental Objective]

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

### [Software being used]

**IDE:** We use IntelliJ IDEA(Ultimate) as integrated development environment

**DataBase:** We use Mysql as database in this tutorial, and you can use other instead.

**EolinkerTool:** This software is for testing the arguments for RestController

**Maven:** You need to add maven into your project framework

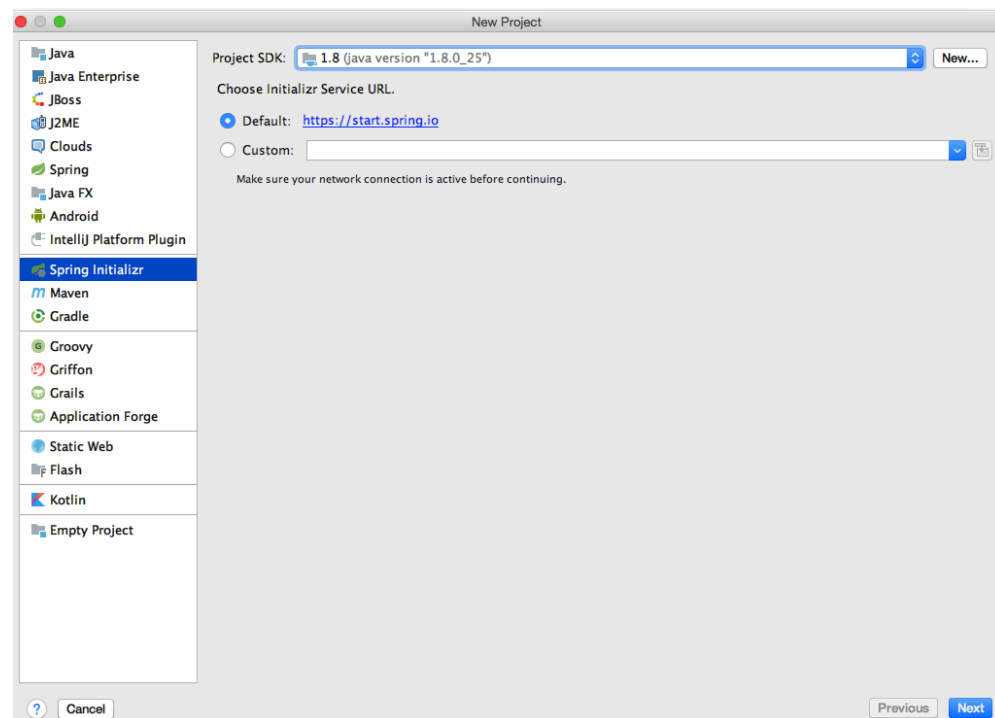
### [Exercise]

### Initialize a spring project

Open IntelliJ IDEA, and click Create New Project

#### 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.

**Project Metadata**

Group:

Artifact:

Type:

Language:

Packaging:

Java Version:

Version:

Name:


Description:

Package:

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

### Initialize from official website

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

 **Spring Initializr**  
Bootstrap your application

Light UI | GitHub

**Project** Maven Project | Gradle Project

**Language** Java | Kotlin | Groovy

**Spring Boot** 2.2.0 RC1 | 2.2.0 (SNAPSHOT) | 2.1.10 (SNAPSHOT) | 2.1.9

**Project Metadata**

Group  
com.example

Artifact  
springproject

Options

Name  
springproject

Description  
SUSTech OOAD exercise for Tutorial 6

The screenshot shows the Spring Initializr web interface. The 'Options' section on the left contains the following fields:

- Name:** springproject
- Description:** SUSTech OOAD exercise for Tutorial 6
- Package Name:** com.example.springproject
- Packaging:** Jar (selected), War
- Java:** 11, 8 (selected)

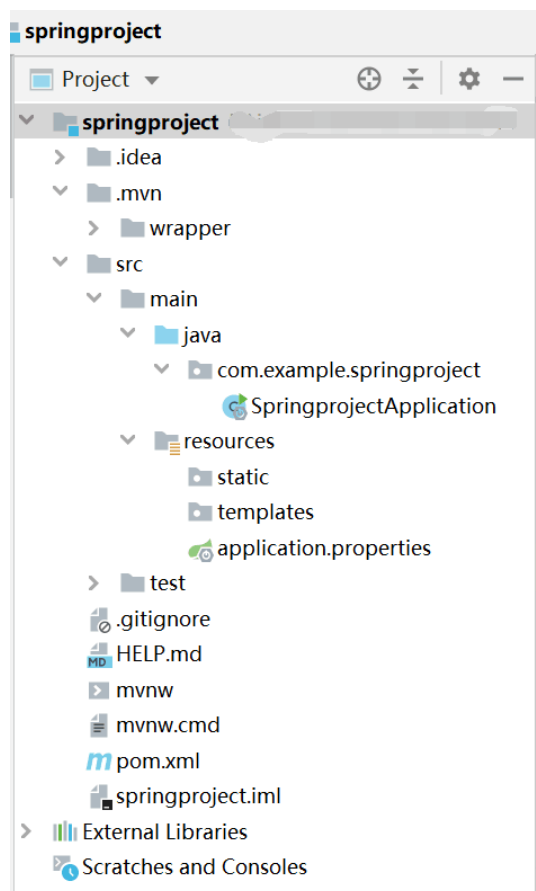
The 'Dependencies' section on the left shows a search bar with the text 'Web, Security, JPA, Actuator, Devtools...'. The 'Selected dependencies' section on the right shows one dependency selected:

- Spring Web**  
Build web, including RESTful, applications using Spring MVC.  
Uses Apache Tomcat as the default embedded container.

At the bottom, there are three buttons: 'Generate - ⌘ + ↵', 'Explore - Ctrl + Space', and 'Share...'. The footer text reads: '© 2019 Pivotal Software. spring.io is powered by Pivotal Web Services'.

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

After that, our spring project has been created, and you need to wait several minutes for downloading resource, and finally, the catalogue as following graph.



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

```
<?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 Tutorial 6</description>

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

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

### 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) {

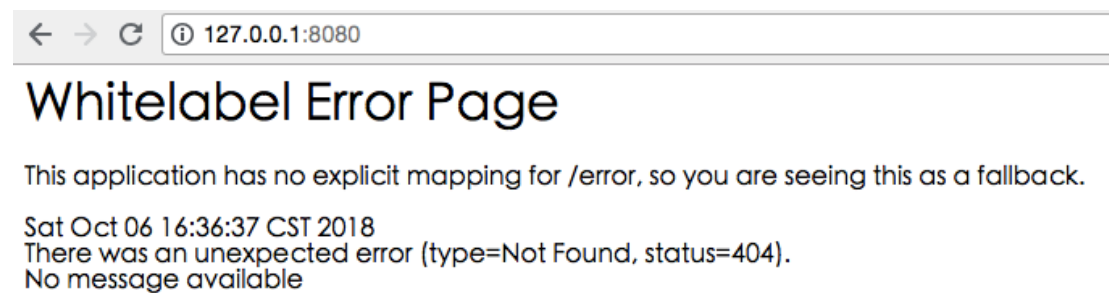
        SpringApplication.run(SpringprojectApplication.class, args);
    }
}
```

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.

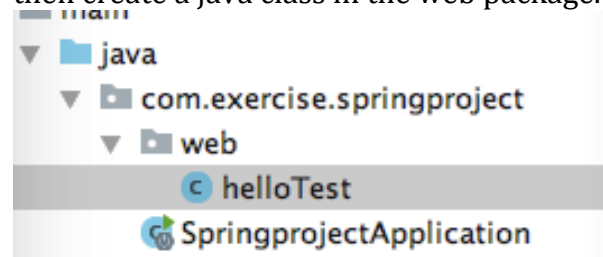
```
main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/*] onto handler of type
main] o.s.j.e.a.AnnotationMBeanExporter      : Registering beans for JMX exposure on sta
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) wi
main] c.e.s.SpringprojectApplication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

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 start our spring project successfully.



### Let's start our first test

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



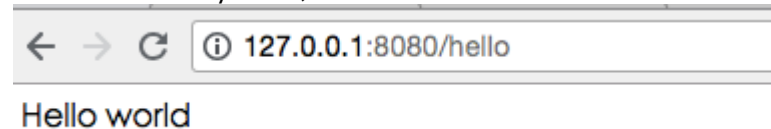
The code in helloTest class as follows

```
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.

## JPA

**JPA** (Java Persistence API), is used for mapping Java objects to relational 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.

How to use it?

### Step 1. Open database

In this tutorial we use mysql as database, please make sure that your computer has been already installed mysql, (you can also use another database, such as pg) you also need to create a database named **spring\_project** before following exercise.

1. Open terminal window and visit the file "mysql\bin" by your installation path.

2. `./mysql -u root -p` then input the password.
3. `create database spring_project;` Which is the database for this tutorial.
4. `use spring_project;` Which means we will use the scheme.

```

zymMacBook:~ zym$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 161
Server version: 5.7.10 MySQL Community Server (GPL)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database spring_project charset utf8;
Query OK, 1 row affected (0.03 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
|  |
|  |
|  |
| spring_project |
| sys |
+-----+
9 rows in set (0.02 sec)

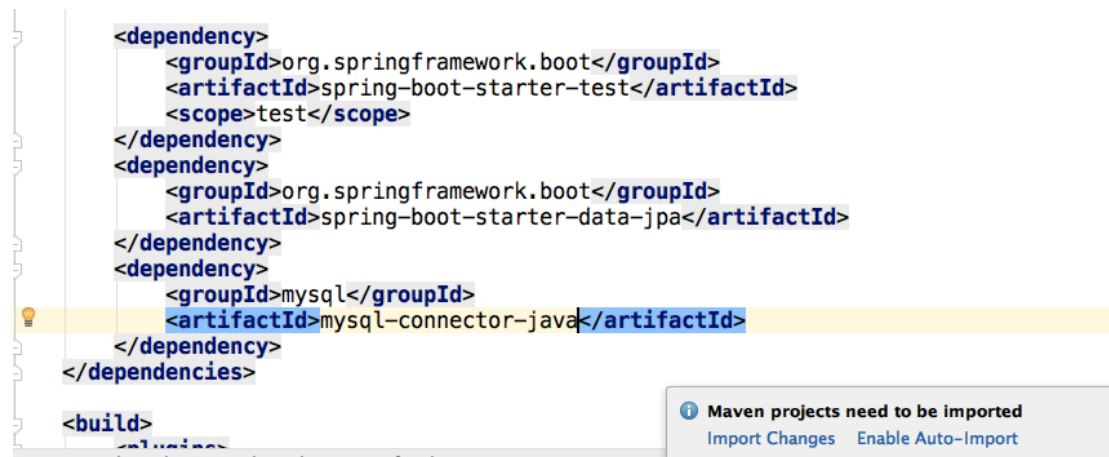
mysql> use spring_project;
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql>

```

## Step 2. Setting Dependency.

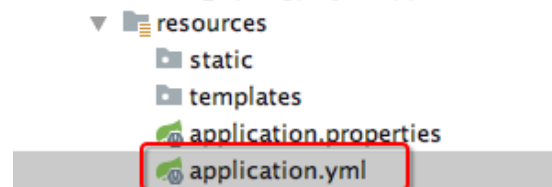
Open pom.xml and add two dependencies as follows. The one is JPA dependency and the other is the jdbc for mysql. It is a [maven](#) project, that can provide an easy way to share JARs according to the groupId and artifactID, so that it is convenient for us to import changes. After we adding those two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



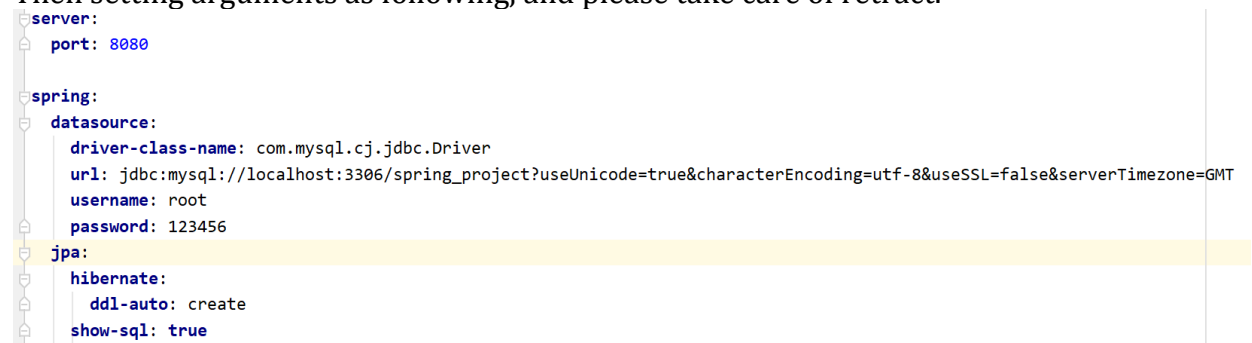
### Step 3. Adding a propertie file (.yml)

.yml file is a kind of configuration file. 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", and the yml file is based on a tree structure.



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

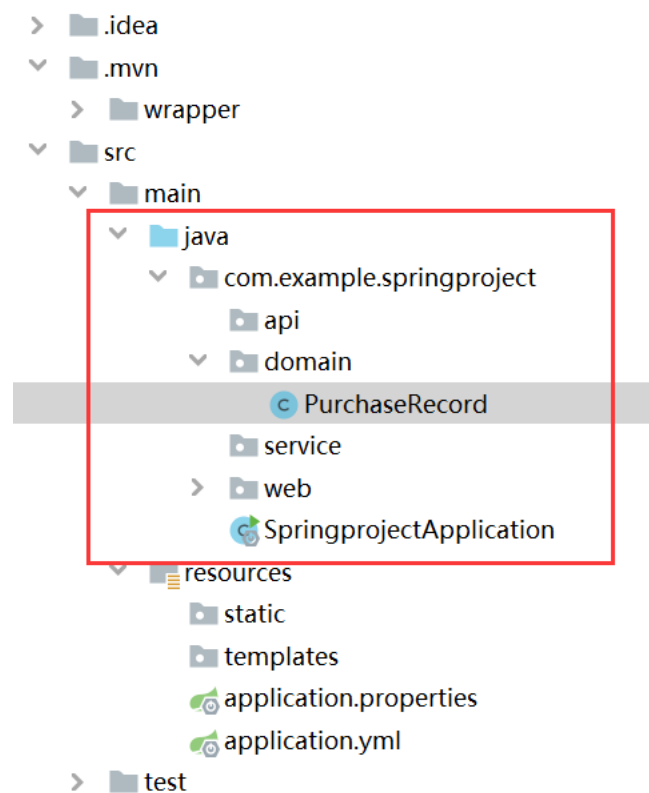




## MVC frame design

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. **web**: The controller classes are defined in this layer, which is to interact with the front-end.

Then we need to create four packages including api, domain, service and web under the package springproject. After that, we need to create an entity class named **PurchaseRecord** in domain layer. The catalog will be as follows:



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, our 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:

```
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.

```
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;  
}
```

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

```
mysql> show tables;
+-----+
| Tables_in_spring_project |
+-----+
| purchase_record           |
+-----+
1 row in set (0.00 sec)

mysql> desc purchase_record;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | bigint(20)    | NO   | PRI | NULL    | auto_increment |
| date       | varchar(255)  | YES  |     | NULL    |                |
| description | varchar(255)  | YES  |     | NULL    |                |
| money      | double        | NO   |     | NULL    |                |
| type       | int(11)       | NO   |     | NULL    |                |
| username   | varchar(255)  | NO   |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.01 sec)

mysql> █
```

Here provides a link that can help to understand more Annotations in JPA Entity Class.

<https://www.javaguides.net/2018/12/jpa-entity-class-basics.html>

## Manipulate the database

### Step 1. 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.

```
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
('Yueming','2019-09-12',45,0,'McDonald');
```

```
insert into purchase_record (username, date, money, type, description) values ('Yuqun','2019-
09-06',200,0,'Taobao');
```

```
insert into purchase_record (username, date, money, type, description) values ('Yuqun','2019-
10-07',2500,0,'Southern Airlines');
```

### Step 2. Change configure file

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

```
jpa:
  hibernate:
    ddl-auto: update
  show-sql: true
```

### Step 3. Build MVC structure in springboot

**Api package:** Create an interface named PurchaseRecordRepository, which need to extends JpaRepository.

```
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.

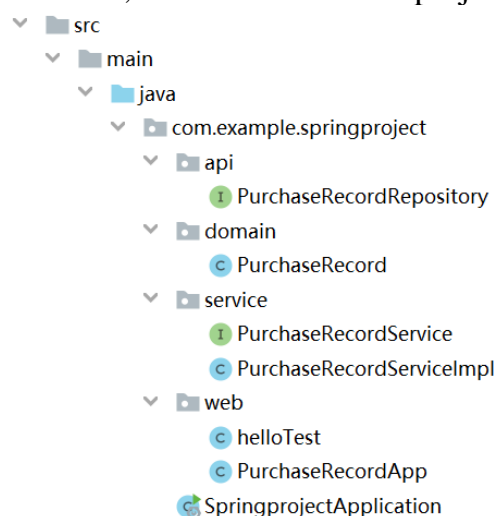
```
public interface PurchaseRecordService {
}
```

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

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

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

After that, the structure of our project would be as follows.



## 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 all rows

Adding following code into corresponding Class.

#### **PurchaseRecordService:**

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

#### **PurchaseRecordServiceImpl:**

```
@Service  
public class PurchaseRecordServiceImpl implements PurchaseRecordService  
{  
    @Autowired  
    private PurchaseRecordRepository purchaseRecordRepository;  
  
    @Override  
    public List<PurchaseRecord> findAll() {  
        return purchaseRecordRepository.findAll();  
    }  
}
```

#### **PurchaseRecordApp:**

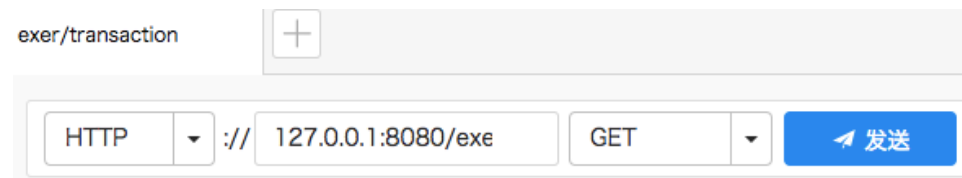
```
@RestController  
@RequestMapping("/exer")  
public class PurchaseRecordApp {  
    @Autowired  
    private PurchaseRecordService purchaseRecordService;  
  
    @GetMapping("/record")  
    public List<PurchaseRecord> findAll(){  
        //For testing the concrete class  
        System.out.println(purchaseRecordService.getClass().getName());  
        return purchaseRecordService.findAll();  
    }  
}
```

## Test

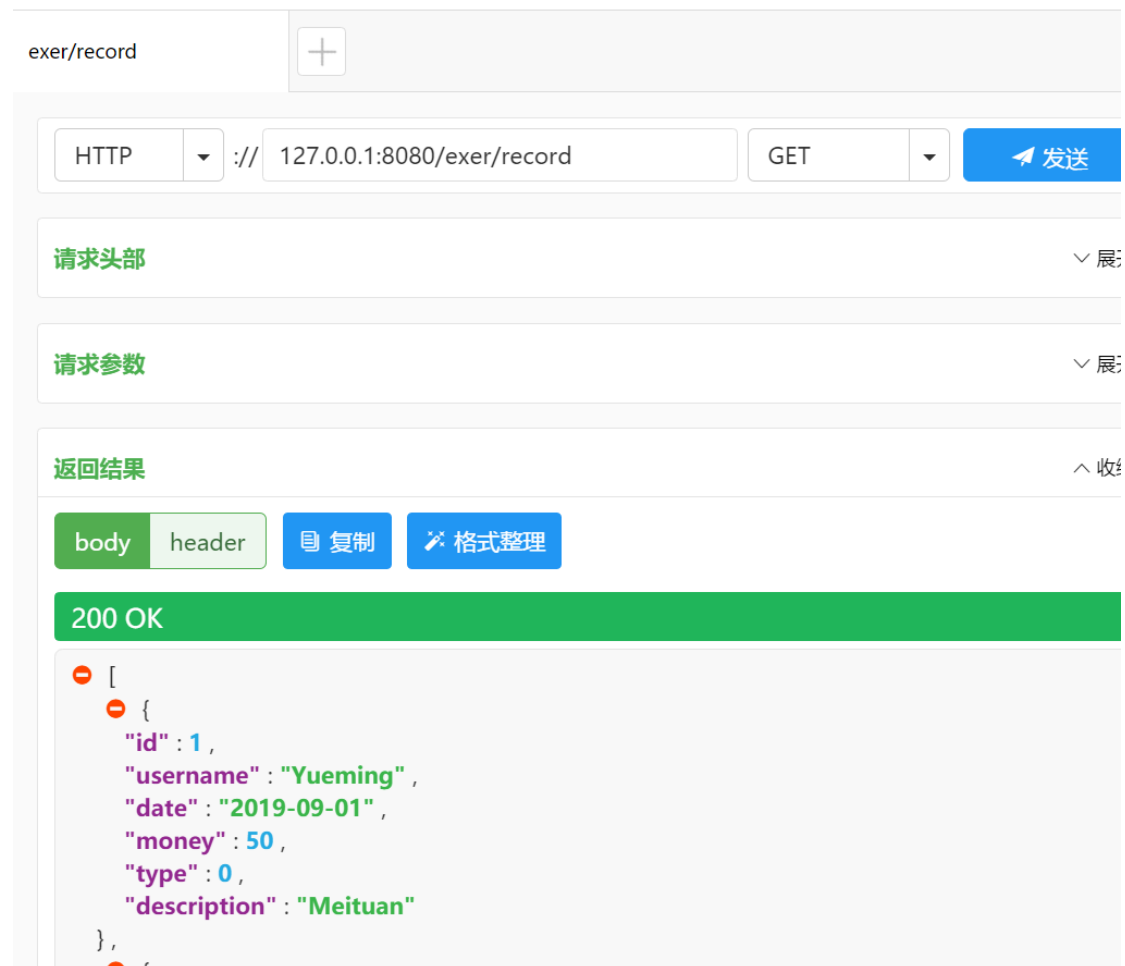
We will introduce the front-end part next week, in this tutorial we use an interface tester: [eolinker](http://eolinker.com/#/) to test the interaction between controller and server.  
<https://www.eolinker.com/#/>

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.



Finally, it will return all information from **purchase\_record** table.



#### **save(): Adding one row**

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.

exer/record

HTTP :// 127.0.0.1:8080/exer/record POST 发送

请求头部 展开

请求参数 ? 收缩

表单[form-data] RESTful 源数据[raw] 表单转源数据[Json] ?

请求参数	类型	参数值	操作
<input checked="" type="checkbox"/> username	[Text]	Yueming	—
<input checked="" type="checkbox"/> date	[Text]	2019-10-16	—
<input checked="" type="checkbox"/> money	[Text]	200	—
<input checked="" type="checkbox"/> type	[Text]	0	—
<input checked="" type="checkbox"/> description	[Text]	Taobao	—

After sending the form, a row has been added into purchase\_record table, and the eolinker would display:

body header 复制 格式整理

200 OK

```
{
  "id": 6,
  "username": "Yueming",
  "date": "2019-10-16",
  "money": 200,
  "type": 0,
  "description": "Taobao"
}
```

### 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.

### PurchaseRecordService:



Also using save() method.

### PurchaseRecordApp:

```
@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 follows:

exer/record

HTTP :// 127.0.0.1:8080/exer/record PUT 发送

请求参数 ? 收缩

表单[form-data] RESTful 源数据[raw] 表单转源数据[Json] ?

请求参数	类型	参数值	操作
<input checked="" type="checkbox"/> username	[Text]	Yueming	—
<input checked="" type="checkbox"/> date	[Text]	2019-10-15	—
<input checked="" type="checkbox"/> money	[Text]	500	—
<input checked="" type="checkbox"/> type	[Text]	0	—
<input checked="" type="checkbox"/> description	[Text]	Taobao	—
<input checked="" type="checkbox"/> id	[Text]	6	—

And the result would be

返回结果 收缩

body header 复制 格式整理

200 OK

```
{
  "id": 6,
  "username": "Yueming",
  "date": "2019-10-15",
  "money": 500,
  "type": 0,
  "description": "Taobao"
}
```

### DeleteById(): Delete one row

Adding following code into corresponding Class.

#### PurchaseRecordService:

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

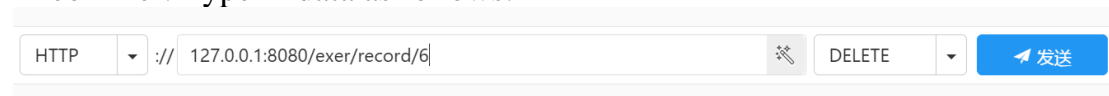
#### PurchaseRecordServiceImpl:

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

#### PurchaseRecordApp:

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

In eolinker. Type in data as follows:



The screenshot shows the Eolinker interface for configuring an API request. The method is set to 'DELETE' and the URL is 'http://127.0.0.1:8080/exer/record/6'. A blue button labeled '发送' (Send) is visible on the right.

And do not surprise nothing would return in eolinker, because the return value of the `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 6.

### @Query. How to add User Defined Queries.

Suppose it is a requirement that we want to find data according to its name and type, so we need to design our own query.

#### PurchaseRecordRepository:

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

#### PurchaseRecordService:

```
public List<PurchaseRecord> findByNameAndAndType(String name, int type);
```

#### PurchaseRecordServiceImpl

```
@Override
public List<PurchaseRecord> findByNameAndAndType(String name, int type)
{
    return purchaseRecordRepository.findByNameAndAndType(name,type);
}
```

## PurchaseRecordApp:

```
@PostMapping("find")
public List<PurchaseRecord> findByNameANDType(@RequestParam String
username,
                                             @RequestParam int type){
    return purchaseRecordService.findByNameAndAndType(username,type);
}
```

## Eolinker

exerciser/find

HTTP POST 127.0.0.1:8080/exer/find 发送

请求参数 ? 收缩

表单[form-data] RESTful 源数据[raw] 表单转源数据[Json] ?

请求参数	类型	参数值	操作
<input checked="" type="checkbox"/> username	[Text]	Yuqun	—
<input checked="" type="checkbox"/> type	[Text]	0	—

The result is:

```
[
  {
    "id": 5,
    "username": "Yuqun",
    "date": "2019-6-30",
    "money": 300,
    "type": 0,
    "description": null
  },
  {
    "id": 6,
    "username": "Yuqun",
    "date": "2019-6-1",
    "money": 200,
    "type": 0,
    "description": null
  }
]
```

### Other Resource:

In this tutorial, we only provide some basic examples for you to exercise how to interact with database by JPA. If you are the database designer or the rear-end designer in your project group, you need to understand more about Persistency and Spring data JPA:

1. Official documents about Spring data JPA  
<https://docs.spring.io/spring-data/jpa/docs/current/reference/html>
2. Other comprehensive introduce of JPA  
<https://www.javaguides.net/p/jpa-tutorial-java-persistence-api.html>

**Designed by Yueming ZHU (in 2018 and modified in 2019)**