SpringBoot开发

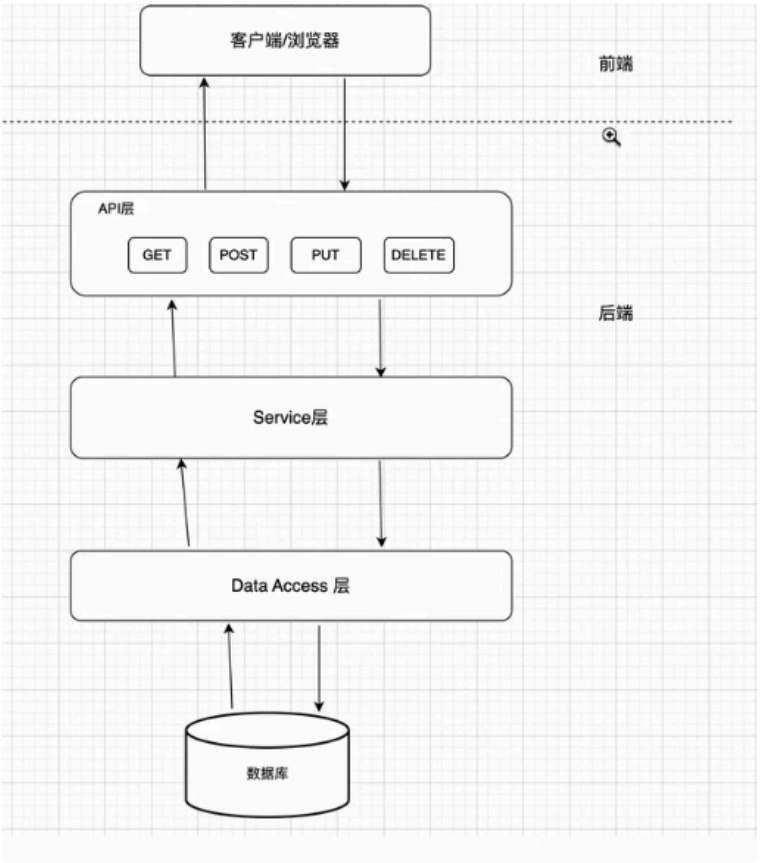
Spring Boot简介

Spring Boot是一个基于Spring框架的开源框架，用于简化Spring应用程序的初始搭建和开发过程。它通过提供约定优于配置的方式，尽可能减少开发者的工作，使得开发Spring应用变得更加快速、便捷和高效。

主要特点：

1. 简化配置：Spring Boot遵循约定优于配置，减少了传统Spring应用中的大量配置。它通过自动配置（auto-configuaration）和起步依赖（starter dependencies）来简化项目的配置过程，让开发者可以快速搭建起一个可运行的Spring应用
2. 集成性强：Spring Boot提供了大量的开箱即用的特性和功能，如内嵌的Servlet容器（如Tomcat、Jetty或Undertow）、健康检查、指标监控等，它还整合了诸多常用的库和框架，如Spring Data、Spring Security等，使得开发者可以快速构建出功能完善的应用
3. 微服务支持：Spring Boot非常适合用于构建微服务架构。它提供了丰富的支持、如通过Spring Cloud进行微服务架构的开发，集成了服务发现、配置中心、负载均衡等功能，帮助开发者构建可伸缩、高可用的微服务系统
4. 内嵌服务器：Spring Boot可以将应用程序打包成一个可执行的JAR文件，并内置了常用的Servlet容器，如Tomcat、Jetty或Undertow。这样一来，开发者可以通过简单的java -jar命令来运行应用程序，而无需部署到外部应用服务器
5. 生态丰富：由于Spring Boot的广泛应用和强大生态系统，开发者可以轻松地使用各种扩展和插件，提高开发效率和应用质量。

项目架构：



- API层：接受浏览器/客户端的相关请求
- Service层：实现具体逻辑操作
- Data Access层：直接访问数据库的代码

Restful API规范：

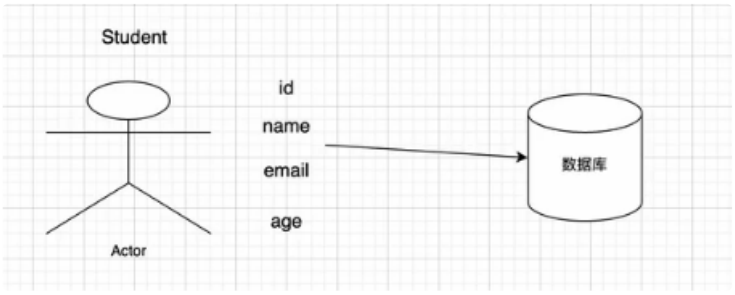
路径：路径又称“终点”（endpoint），表示API的具体网址。在RESTful架构下，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。

HTTP动词：

- GET（SELECT）：从服务器取出资源（一项或多项）
- POST（CREATE）：从服务器新建一个资源
- PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）
- PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）
- DELETE（DELETE）：从服务器删除资源

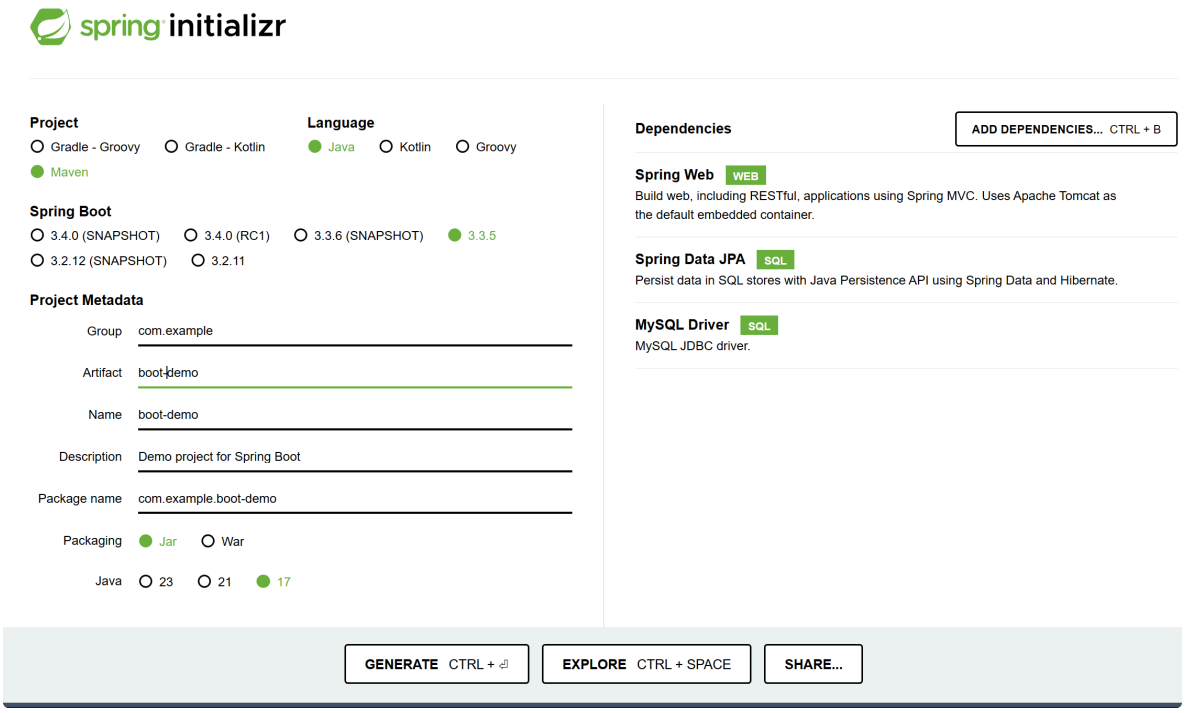
入门项目

实现学生信息的增删改：



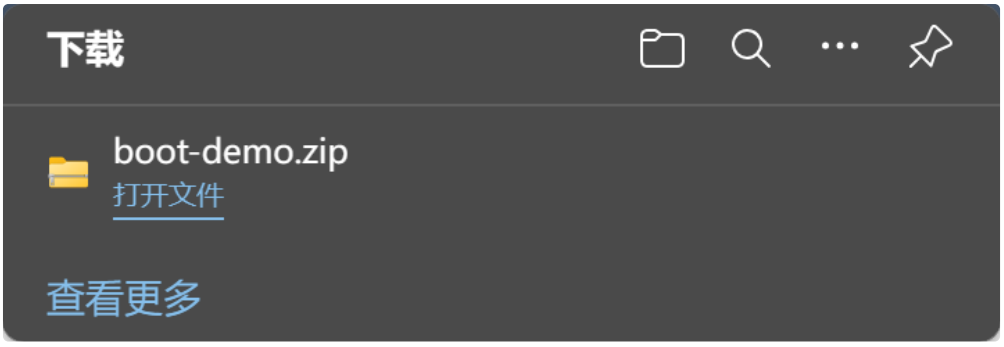
初始化SpringBoot应用

链接至start.spring.io网址，对于一个Web应用来说，可以使用如下的配置：



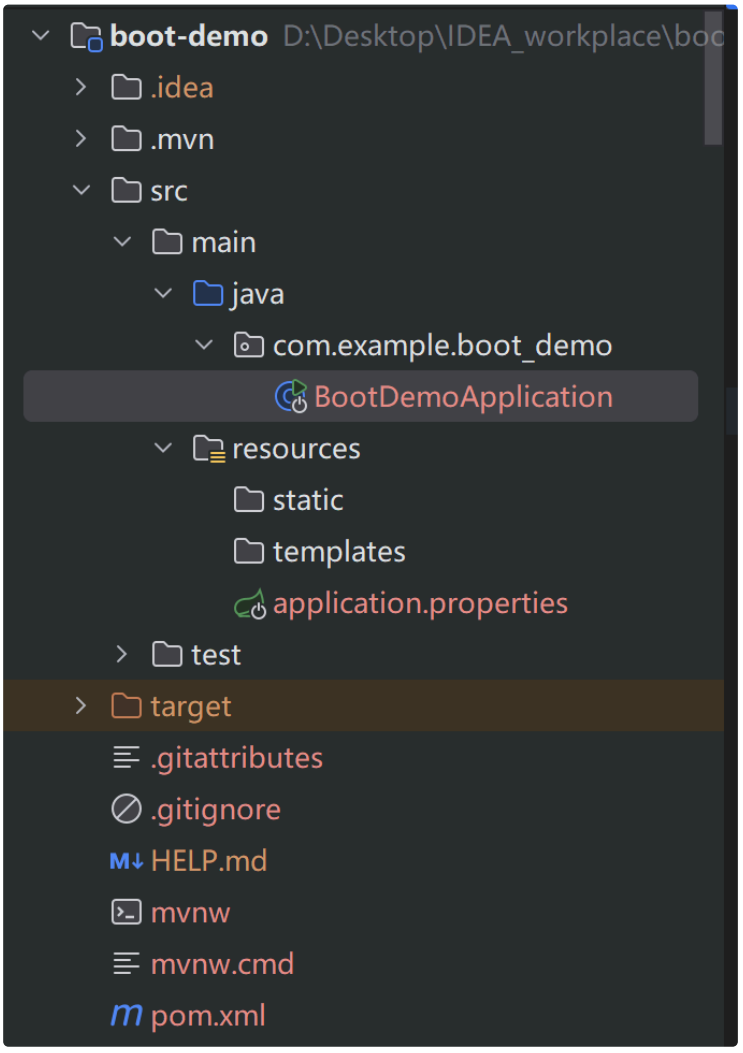
- Spring Web：处理Restful API接口等信息
- Spring Data JPA：数据抽象，简化SQL
- MySQL Driver：数据库的驱动

点击GENERATE后会下载一个zip包：



解压后导入到合适的位置，使用IDEA打开：

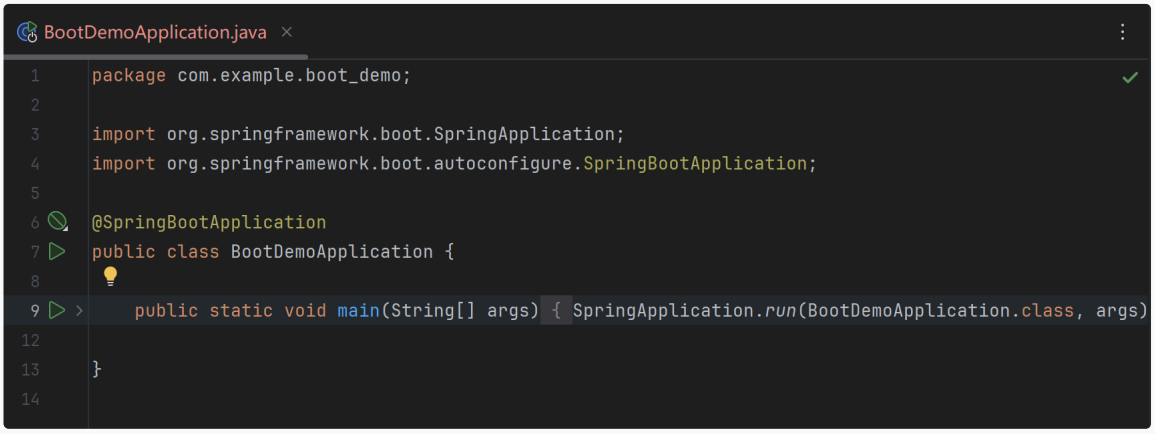
点击Maven配置脚本后自动安装依赖和相关插件，之后我们可以查看项目结构：



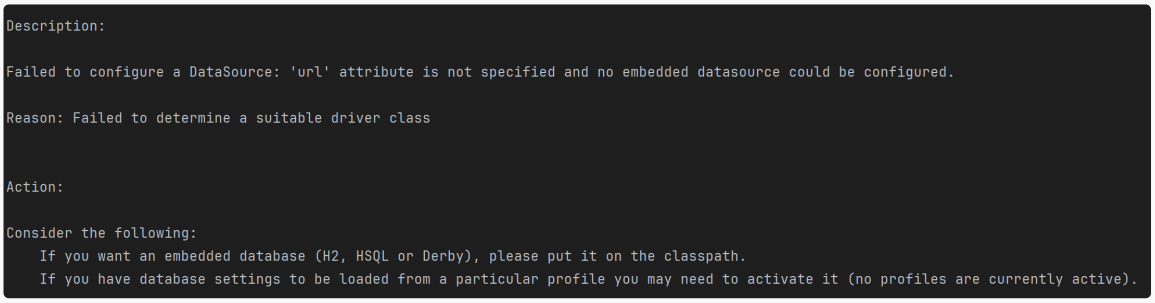
- src/main/java：后端
- src/main/resources：一些资源文件
- pom.xml：项目依赖

如何启动后端?

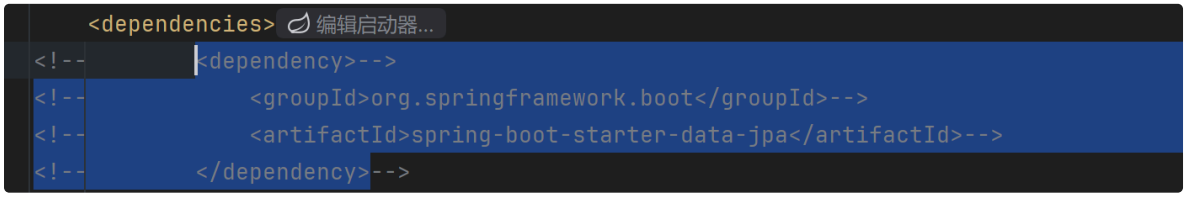
点击BootDemoApplication，点击左边的绿色箭头运行即可



由于目前还没有连接数据库（因为引入jpa的包需要连接数据库），现在运行的效果如下：



我们注释掉JPA相关的依赖，启动后端：



再次点击启动：

```
main] c.example.boot_demo.BootDemoApplication  : Starting BootDemoApplication using Java 22 with PI
main] c.example.boot_demo.BootDemoApplication  : No active profile set, falling back to 1 default p
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
main] o.a.catalina.core.AprLifecycleListener   : An older version [1.3.0] of the Apache Tomcat Nati
main] o.a.catalina.core.AprLifecycleListener   : Loaded Apache Tomcat Native library [1.3.0] using
main] o.a.catalina.core.AprLifecycleListener   : OpenSSL successfully initialized [OpenSSL 3.0.13 3
main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.31]
main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization complet
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context pa
main] c.example.boot_demo.BootDemoApplication  : Started BootDemoApplication in 1.27 seconds (proce
```

访问本地8080端口：

Whitelabel Error Page

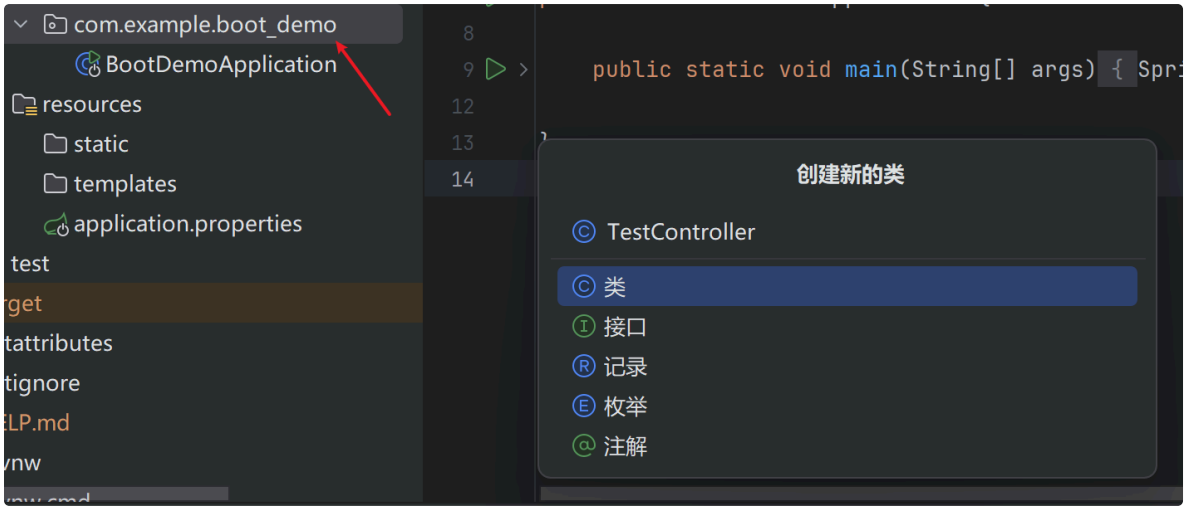
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Nov 08 14:53:53 CST 2024
There was an unexpected error (type=Not Found, status=404).

由于目前一个API端口都没有，因此会报错

在SpringBoot中，所有的API以Controller的方式提供

在com.example.boot_demo下新建一个TestController java类



添加上`RestController`**注解**，表示提供的是一个RestAPI的Controller，这意味着它将处理HTTP请求并作出响应：

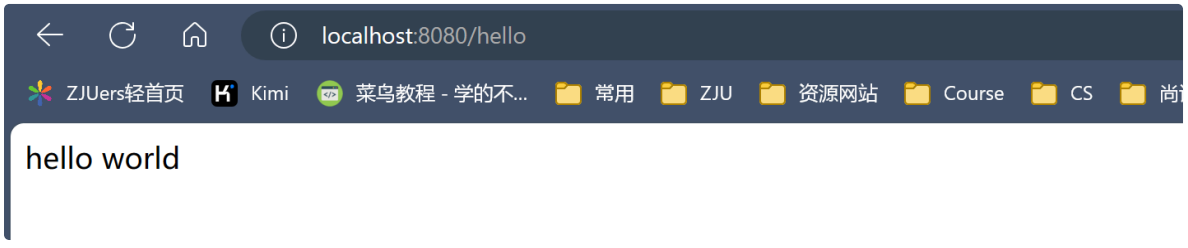
```java
package com.example.boot_demo;

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

@RestController 新 *
public class TestController {
}
```

在这个控制器中，定义了一个处理HTTP GET请求的方法`hello()`，该方法映射到`/hello`路径。当有请求到达这个路径时，`hello()`方法会被调用，并且它会返回一个字符串`"hello world"`作为响应。

> 使用@GetMapping("...")表示请求的路径

```java
@RestController 新 *
public class TestController {

    @GetMapping("/hello") 新 *
    public String hello(){
        return "hello world";
    }
}
```

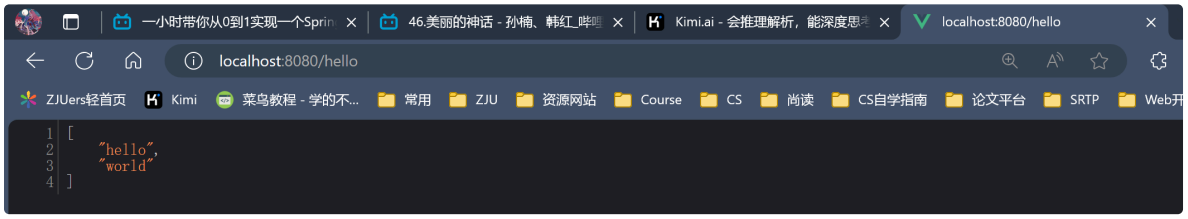
此时再次启动后端，查看本地的8080端口，路径为`/hello`：



假如说我们修改为List：

```java
@RestController 新*
public class TestController {

    @GetMapping("/hello") 新*
//    public String hello(){
//        return "hello world";
//    }
    public List<String> hello(){
        return List.of("hello","world");
    }
}
```

再次启动后端访问：可以发现返回了一个json对象



> 关于RestController中的方法类型以及Spring Boot的解析手段：
>
> 1. **基本数据类型和包装类（如int, long, double, String等）**： 这些类型的返回值将直接作为响应体返回。
> 2. **Java对象**： 返回的Java对象将被自动序列化为JSON（默认使用Jackson库）或XML（如果配置了JAXB）。
> 3. **ResponseEntity**： 返回`ResponseEntity`对象可以让你更细致地控制响应，包括状态码、响应头和响应体。
> 4. **HttpEntity**： 类似于`ResponseEntity`，但用于请求，表示请求的主体和头部信息。
> 5. **List或数组**： 集合或数组类型的返回值将被序列化为JSON数组。
> 6. **Map**： 返回的`Map`对象将被序列化为JSON对象。
> 7. **Optional**： 返回的`Optional`对象将被序列化为包含单个元素的JSON数组，或者在`Optional`为空时序列化为`null`。
> 8. **URI或URL**： 返回的`URI`或`URL`对象可以用来重定向。

9. **InputStream或OutputStream**： 可以直接返回流，用于文件下载或上传。
10. **File或Path**： 返回文件对象，用于文件下载。
11. **StreamingResponseBody**： 用于流式响应，例如视频流或大文件传输。
12. **Callable或DeferredResult**： 用于异步处理，可以延迟响应直到某个操作完成。
13. **Mono或Flux（响应式编程）**： 如果你使用Spring WebFlux，可以返回响应式类型，如 `Mono` 或 `Flux`。
14. **View或String视图名称**： 尽管 `@RestController` 注解通常与视图返回不兼容，但如果你在方法上使用 `@ResponseBody` 注解，仍然可以返回视图名称。
15. **异常处理**： 你可以返回 `Error` 对象或自定义异常，Spring Boot将它们转换为错误响应。

数据库建立

针对我们的项目，我们需要创建一个存储学生信息的数据库：

进入mysql：

```
RFY@RFY  D:/  master  base 3.11.7
> mysql -u root -p
Enter password: ****************
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.36 MySQL Community Server - GPL

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

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.
```

创建数据库test，创建学生信息表：

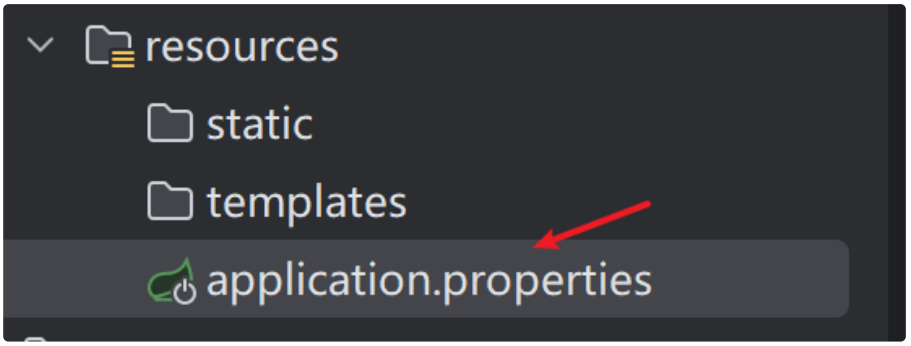
```sql
CREATE DATABASE test;

use test;

CREATE TABLE student (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    age INT
);
```

```
mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.02 sec)

mysql> use test;
Database changed
mysql> CREATE TABLE student (
    ->  id INT AUTO_INCREMENT PRIMARY KEY,
    ->  name VARCHAR(50) NOT NULL,
    ->  email VARCHAR(100) NOT NULL,
    ->  age INT
    -> );
Query OK, 0 rows affected (0.06 sec)

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

插入一个student信息，用于后面的测试：

```
mysql> insert into student (name,email,age) values('rfy','876320233@qq.com',20);
Query OK, 1 row affected (0.02 sec)

mysql> select * from student
    -> ;
+----+------+-----------------+------+
| id | name | email           | age  |
+----+------+-----------------+------+
|  1 | rfy  | 876320233@qq.com |   20 |
+----+------+-----------------+------+
1 row in set (0.00 sec)
```

在项目目录下，在resources/application.properties中配置数据库信息



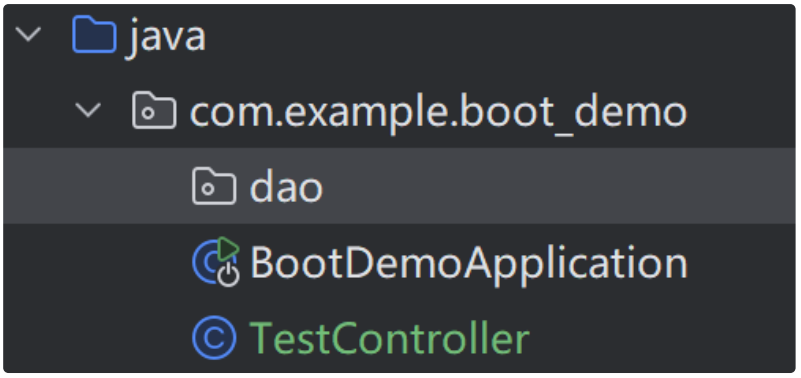
```
spring.application.name=boot-demo
spring.datasource.url = jdbc:mysql://localhost:3306/test?characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=
```

url中指明了ip、端口、数据库名称（test）

Data Access层

主要通过JPA实现，因此需要在pom.xml中将jpa的依赖加进来

建立一个新的package `dao`，表示数据访问层的相关信息：



在dao中我们需要将数据库表映射到一个java类中，例如在本次项目中，学生表需要进行映射到学生对象。因此可以创建一个 `Student` 的java类，然后输入如下代码：

```
package com.example.boot_demo.dao;

public class Student {  0 个用法  新 *
    private int id;  0 个用法
    private String name;  0 个用法
    private String email;  0 个用法
    private int age;  0 个用法
}
```

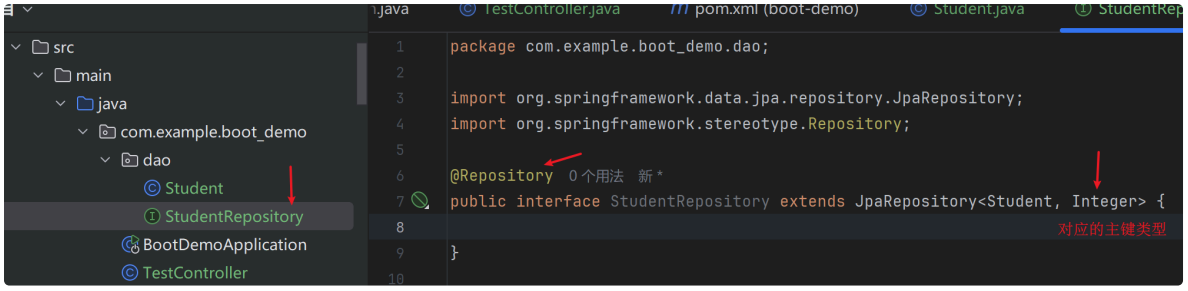
注意，还需要添加一些**注解**（Springboot中注解很有用处）

```java
package com.example.boot_demo.dao;


import jakarta.persistence.*;

@Entity  0 个用法  新 *
@Table(name = "student")
public class Student {
    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "name")  0 个用法
    private String name;

    @Column(name = "email")  0 个用法
    private String email;

    @Column(name = "age")  0 个用法
    private int age;
}
```

- `@Entity` 注解标记这个类是一个JPA实体。
- `@Table(name = "student")` 注解指定这个实体映射到数据库中的`student`表。
- `@Id`注解标记为主键
- `@Column(name = "id")` 注解指定`id`属性映射到数据库表的`id`列。
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` 注解指定主键生成策略为自增长（IDENTITY），这意味着数据库将自动为每个新记录生成一个唯一的ID。

这个实体类可以用来表示数据库中的student表中记录，在Spring Data JPA中，可以使用这个实体来执行CRUD（创建、读取、更新、删除）操作。我们可以创建一个继承自JpaRepository的接口StudentRepository来操作Student实体。

在dao下，创建一个接口类，名字为`StudentRepository`，用于管理操作Student实体。

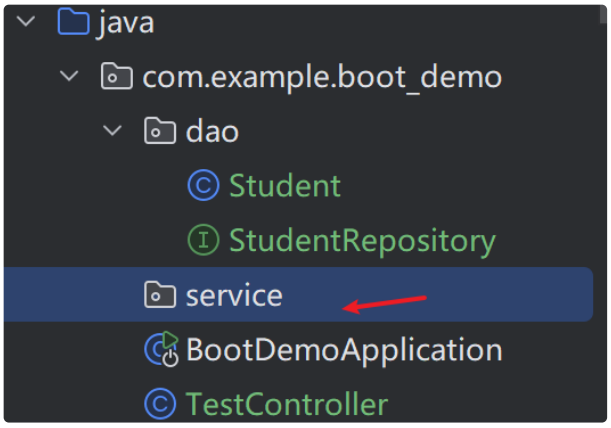


JpaRepository后的两个参数，一个是类，一个是类对应的主键类型，其本身实现了很多常见的操作，后面我们可以看到怎么使用。

Service层

接下来我们要实现Service层的代码，即如何使用这些JPA来实现业务逻辑。

在demo下创建名为service的包：



首先我们创建一个StudentService的接口，里面声明了我们想要实现的各种方法；然后创建一个StudentServiceImpl类，继承自StudentService接口，用于具体定义这些方法。

StudentService中先定义核心功能：通过id查找一个学生

```java
package com.example.boot_demo.service;

import com.example.boot_demo.dao.Student;

public interface StudentService {  1 个用法  1 个实现  新 *
    Student getStudentById(int id);  0 个用法  1 个实现  新 *
}

```

StudentServiceImple中实现相关功能：

```java
package com.example.boot_demo.service;

import com.example.boot_demo.dao.Student;
import com.example.boot_demo.dao.StudentRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;


@Service  新 *
public class StudentServiceImpl implements StudentService{

    @Autowired
    private StudentRepository studentRepository;

    @Override  0 个用法  新 *
    public Student getStudentById(int id) {
        return studentRepository.findById(id).orElseThrow(RuntimeException::new);
    }
}
```

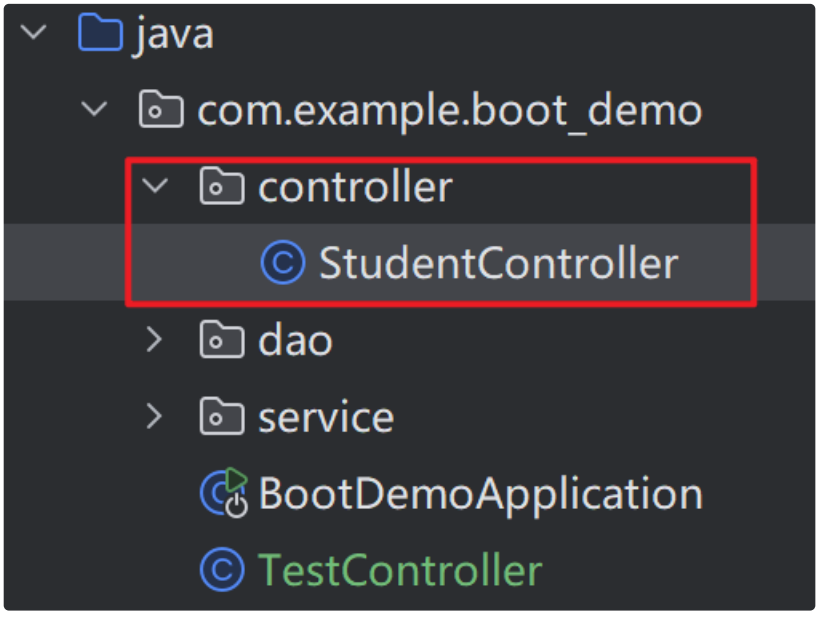
1. **类注解 @Service**：
    - `@Service` 注解表明这个类是一个服务层组件，通常用于业务逻辑的处理。Spring会将这个类标识为一个bean，并将其加入到Spring应用上下文中。
2. **依赖注入**：
    - `@Autowired` 注解用于自动注入 `StudentRepository` 类型的bean。Spring容器会查找一个匹配的 `StudentRepository` 实例，并将其注入到 `studentRepository` 字段中。
3. **方法实现**：
    - `getStudentById(int id)` 方法实现了 `StudentService` 接口中定义的方法，用于根据学生ID获取学生信息。

- `studentRepository.findById(id)` 方法调用了 `StudentRepository` 接口中的方法，根据给定的ID查找学生。这个方法返回一个 `Optional<Student>` 对象。
- `orElseThrow(RuntimeException::new)` 是一个方法引用，它告诉 `Optional` 对象，如果找不到学生（即 `Optional` 为空），则抛出一个 `RuntimeException` 。

Controller

接下来通过Controller实现我们的API。

同样在demo目录下先创建一个controller文件夹，然后创建一个StudentController类



具体代码如下：

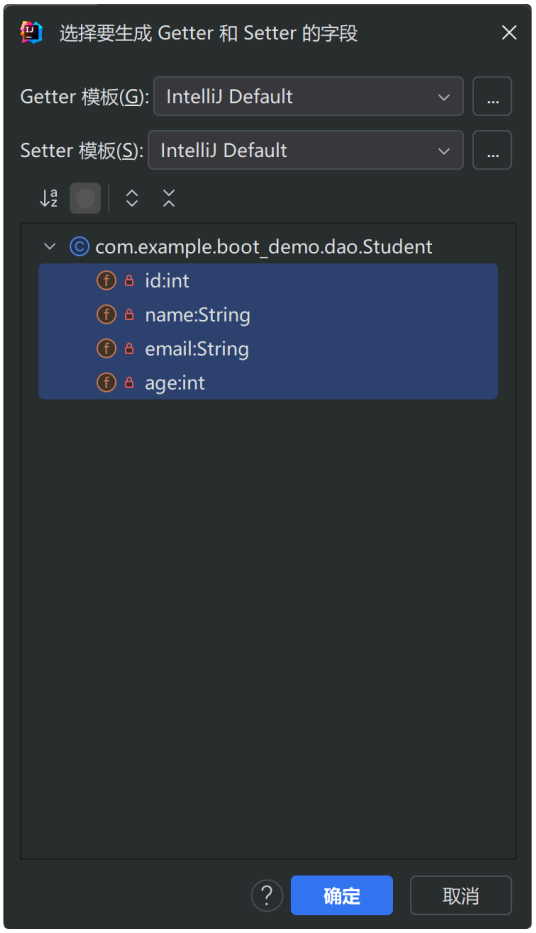
```java
@RestController 新 *
public class StudentController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/student/{id}") 新 *
    public Student getStudentById(@PathVariable int id){
        return studentService.getStudentById(id);
    }
}
```

1. **类注解 @RestController**：
   - `@RestController` 是一个组合注解，它包含了 `@Controller` 和 `@ResponseBody` 。这表明 `StudentController` 是一个处理HTTP请求的控制器，并且其方法的返回值将直接作为HTTP响应正文返回。
2. **依赖注入**：
   - `@Autowired` 注解用于自动注入 `StudentService` 类型的bean。Spring容器会查找一个匹配的 `StudentService` 实例，并将其注入到 `studentService` 字段中。
3. **HTTP GET请求处理**：
   - `@GetMapping("/student/{id}")` 注解将HTTP GET请求映射到 `/student/{id}` 路径。这意味着当有请求到达这个路径时，Spring MVC会调用 `getStudentById` 方法。
   - `@PathVariable int id` ： `@PathVariable` 注解用于将URL中的 `{id}` 部分绑定到方法参数 `id` 上。这样，你就可以在方法中直接使用这个参数。

   > 注：这是符合Restful编程规范的
4. **业务逻辑调用**：
   - `getStudentById` 方法调用了注入的 `StudentService` 中的 `getStudentById` 方法，传入从URL路径中获取的学生ID。
   - `Student` ：方法的返回值是一个 `Student` 对象，这个对象将被自动序列化为JSON格式，并作为HTTP响应正文返回给客户端。

注意，Student对象被自动序列化为JSON格式的时候需要对应的Get，Set方法，因此我们需要在dao层的Student类中添加相应的Get、Set方法：

快捷键：Alt + Insert，点击Getter and Setter，shift全选属性后点击确定：



发现已经创建好了：

```
public int getId() {  0 个用法  新 *
    return id;
}

public void setId(int id) {  0 个用法  新 *
    this.id = id;
}

public String getName() {  0 个用法  新 *
    return name;
}

public void setName(String name) {  0 个用法  新 *
    this.name = name;
}

public String getEmail() {  0 个用法  新 *
    return email;
```

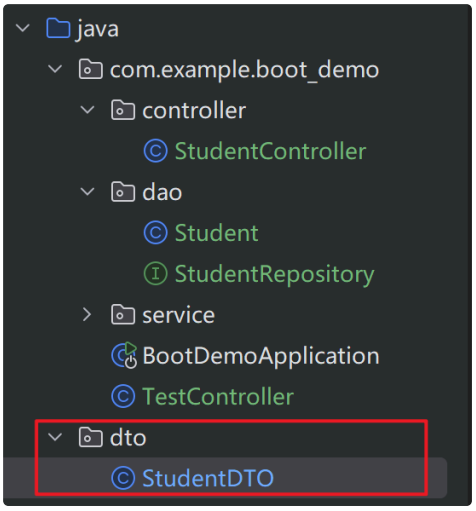
从下往上执行完数据库、数据表的创建，Data Access层JPA对象（Student，StudentRepository），Service层利用JPA实现具体逻辑以及Controller处理Http请求，我们已经初步完成了一个简易的后端，启动服务器，输入域名`localhost:8080/student/1`

Good!

优化一：前端对象转化

在StudentController中，我们直接将Data Access层的Student对象进行了返回，这在实际的业务中有时不太行得通，例如一些密码、加密信息、隐私信息等，我们需要进行一些过滤+转化，展示给前端。

demo目录下创建dto文件夹，在dto目录下，创建StudentDTO类：



实现StudentDTO类，只返回学生的id、name、email信息，而不返回其age信息：

```java
public class StudentDTO {  0 个用法  新 *
    private int id;  2 个用法
    private String name;  2 个用法
    private String email;  2 个用法

    public int getId() {  0 个用法  新 *
        return id;
    }

    public void setId(int id) {  0 个用法
        this.id = id;
```

其次需要将Student对象转化为StudentDTO对象，简单点的做法就是直接修改，但是工程规范期间，通常是在demo目录下创建一个convert目录，在里面实现类的转化：

创建StudentConverter类，实现Student和StudentDTO的转化：

> Converter类中通常使用static静态方法实现，也可以使用动态注解注入

```java
public class StudentConverter {  0 个用法  新 *
    public static StudentDTO convertStudent(Student student) {  0 个用法  新 *
        StudentDTO studentDTO = new StudentDTO();
        studentDTO.setId(student.getId());
        studentDTO.setName(student.getName());
        studentDTO.setEmail(student.getEmail());
        return studentDTO;
    }
}
```

这样，我们在Service层实现的时候需要进行相应的修改：

对StudentService进行修改：

```java
package com.example.boot_demo.service;

import com.example.boot_demo.dao.Student;
import dto.StudentDTO;

public interface StudentService {  3 个用法  1 个实现  新 *  1 个相关问题
//    Student getStudentById(int id);
    StudentDTO getStudentBy(int id);  0 个用法  新 *
}
```

对StudentServiceImpl进行修改：

```java
@Service  新 *
public class StudentServiceImpl implements StudentService{

    @Autowired
    private StudentRepository studentRepository;

    @Override  0 个用法  新 *
//    public Student getStudentById(int id) {
//        return studentRepository.findById(id).orElseThrow(RuntimeException::new);
//    }
    public StudentDTO getStudentById(int id){
        Student student = studentRepository.findById(id).orElseThrow(RuntimeException::new);
        return StudentConverter.convertStudent(student);
    }
}
```

最后在Controller中也进行相应的修改：

```java
@RestController  新 *
public class StudentController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/student/{id}")  新 *
    public StudentDTO getStudentById(@PathVariable int id){
        return studentService.getStudentById(id);
    }
}
```

此时我们再次启动后端，查看8080端口`student/1`：

可以发现已经少了age字段了。

> 小结：本质上就是对Service层内容进行了一些优化，通过新增一些类和转化方法来实现。

优化二：统一Response

前端请求后端数据的时候，需要有一定的格式，例如查看是否成功、错误信息、状态码字段等，因此实际工程中需要对后端返回对象进行封装。

在demo目录下创建一个Response类，由于其data字段类型不固定，因此是一个泛型类：

```
public class Response<T> {  6个用法  新*
    private T data;  2个用法
    private boolean success;  2个用法
    private String errorMsg;  2个用法

    public static <K> Response<K> success(K data) {  0个
        Response<K> response = new Response<>();
        response.setData(data);
        response.setSuccess(true);
        return response;
    }

    public static Response<Void> newFail(String errorMsg
        Response<Void> response = new Response<>();
        response.setSuccess(false);
        response.setErrorMsg(errorMsg);
        return response;
    }
    public T getData() {  0个用法  新*
        return data;
    }
```

注意这个泛型的写法，第一个<K>表示这是一个泛型方法，K是一个泛型参数，返回一个Response<K>的对象。

然后我们在Controller层对返回给前端的信息进行Response的封装：

修改StudentController：

```java
@RestController 新*
public class StudentController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/student/{id}") 新*
    public Response<StudentDTO> getStudentById(@PathVariable int id){
        return Response.newSuccess(studentService.getStudentById(id));
    }
}
```

注意此处的写法，不是Response<...>.newSuccess

重新启动后端查看：

成功

其余功能实现

我们在此前完成了查询相关的后端逻辑，我们后续添加“增加”、“删除”、“修改”功能。

增加功能，我们从上到下进行分析：

- Controller层

  StudentController

  ```java
  @PostMapping("/student") 新*
  public Response<Integer> addStudent(@RequestBody StudentDTO studentDTO){
      return Response.newSuccess(studentService.addNewStudent(studentDTO));
  }
  ```

首先新增学生是Post请求，前端返回一个Json对象，通过`@RequsetBody`注解反序列化到后端与前端交互的`StudentDTO`类。然后调用Service层的addNewStudent方法，返回学生的新Id

- Service层

StudentService接口：

```
public interface StudentService {  3 个用法  1 个实现  新 *
//    Student getStudentById(int id);
    StudentDTO getStudentById(int id);  1 个用法  1 个实现  新 *

    重命名用法
    Integer addNewStudent(StudentDTO studentDTO);  1 个实现  新 *
}
```

StudentServiceImpl：

```
@Override  1 个用法  新 *
public Integer addNewStudent(StudentDTO studentDTO) {
    List<Student> students = studentRepository.findByEmail(studentDTO.getEmail());
    if(!students.isEmpty()){
        throw new IllegalStateException("email " + studentDTO.getEmail() + " has been taken");
    }
    Student student = studentRepository.save(StudentConverter.convertStudentDTO(studentDTO));
    return student.getId();
}
```

首先调用DataAccess层的studentRespository的findByEmail方法，用于查询是否有相同Email的学生（我们假设学生的email不能相同）。如果列表非空，则抛出异常。

否则，调用studentRespository的save方法，同时里面使用Converter将StudentDTO转化成Student类，新增一个新的student，获得自增主键id的值后返回给student。最后返回新学生的id。

> Converter类

实现StudentDTO和Student之间的转化

```java
public class StudentConverter {  3 个用法  新 *
    public static StudentDTO convertStudent(Student student) {  1 个用法  新 *
        StudentDTO studentDTO = new StudentDTO();
        studentDTO.setId(student.getId());
        studentDTO.setName(student.getName());
        studentDTO.setEmail(student.getEmail());
        return studentDTO;
    }

    public static Student convertStudentDTO(StudentDTO studentDTO) {  1 个用法  新 *
        Student student = new Student();
        student.setName(studentDTO.getName());
        student.setEmail(studentDTO.getEmail());
        return student;
    }
}
```

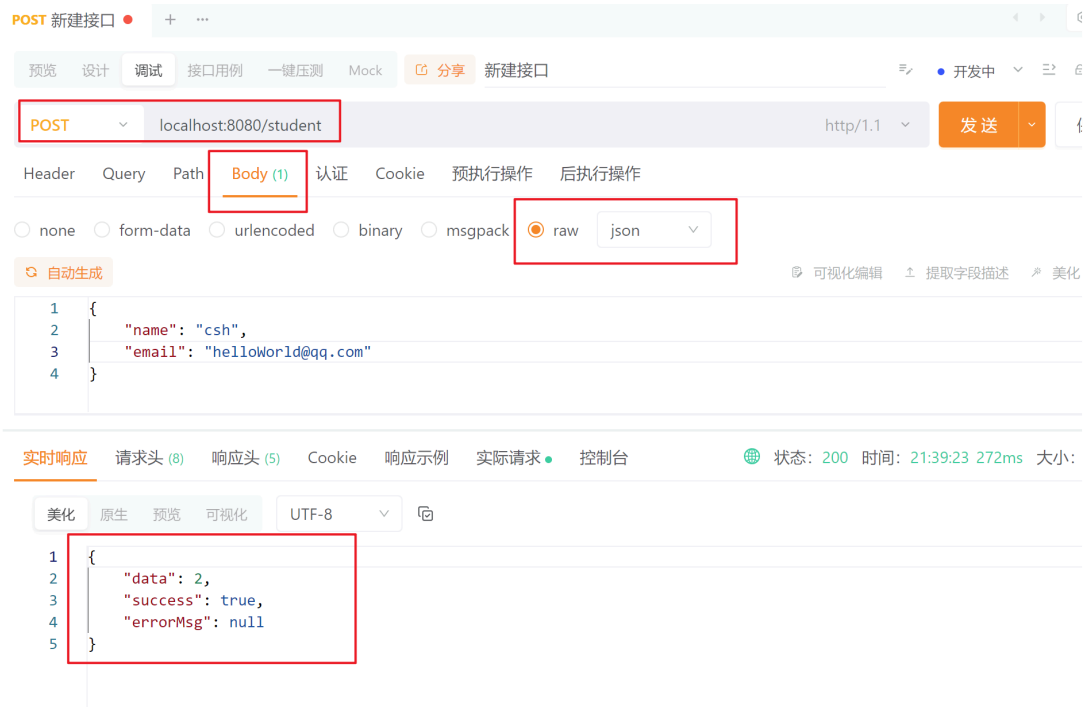
- Data Access层

  StudentRepository接口

```java
@Repository  2 个用法  新 *
public interface StudentRepository extends JpaRepository<Student, Integer> {

    List<Student> findByEmail(String name);  1 个用法  新 *
}
```

  JpaRepository中已经实现好了的是findById，如果我们想要实现其他字段的find，可以按照其既定的格式，例如上例中按照Email查询。

至此完成好了student的新增功能，我们使用Apipost进行测试：



可以看到成功返回了新的学生id，我们去查看是否新增到了我们的数据库中：

```
mysql> select * from student;
+----+------+--------------------+------+
| id | name | email              | age  |
+----+------+--------------------+------+
|  1 | rfy  | 876320233@qq.com   |   20 |
|  2 | csh  | helloWorld@qq.com  |    0 |
+----+------+--------------------+------+
2 rows in set (0.00 sec)
```

成功

删除功能类似：

- Controller层：

  StudentController

```
@DeleteMapping("/student/{id}")  新 *
public void delStudent(@PathVariable int id){
    studentService.delStudent(id);
}
```

  通常删除不需要Response，但有时候可以附带删除成功信息

- Service层

  StudentService接口

```
public interface StudentService {  3 个用法  1 个实现  新 *
//    Student getStudentById(int id);
    StudentDTO getStudentById(int id);  1 个用法  1 个实现  新 *

    Integer addNewStudent(StudentDTO studentDTO);  1 个用法  1 个实现  新 *

    void delStudent(int id);  1 个用法  新 *
}
```

  StudentServiceImpl：

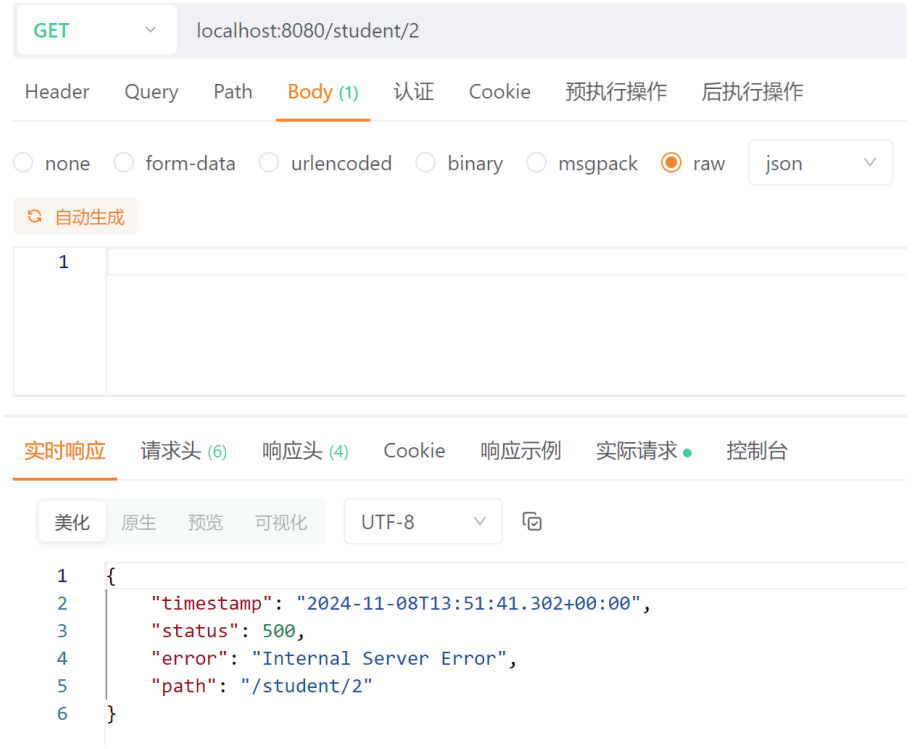
```
@Override  1 个用法  新 *
public void delStudent(int id) {
    Student student = studentRepository.findById(id).orElseThrow(() -> new IllegalStateException("id " + id + "doesn't exist"));
    studentRepository.delete(student);
}
```

  调用DataAccess层的studentRepository的findById方法，判断这个删除的id对应的学生是否存在，不存在则抛出异常。否则，**调用studentRepository的delete函数删除这个对象**

我们来测试一下：

我们已经发送了删除2号学生的请求

使用Get方法再次查看：发现已经找不到了



数据库中查看：发现也没了，删除成功

```
mysql> select * from student;
+----+------+-------------------+------+
| id | name | email             | age  |
+----+------+-------------------+------+
|  1 | rfy  | 876320233@qq.com |   20 |
+----+------+-------------------+------+
1 row in set (0.00 sec)
```

修改功能：

- Controller层：

  StudentController

  方法一：此时前端需要发送form-data类型的请求，即表单数据，在Apipost测试时需要选择form-data

```
@PutMapping("/student/{id}")  新*
public Response<StudentDTO> updateStudent(@PathVariable int id, @RequestParam(required = false) String name,
                                          @RequestParam(required = false) String email){
    return Response.newSuccess(studentService.updateStudentById(id, name, email));
}
```

  - `@RequestParam(required = false) String name`：这是一个请求参数，它将HTTP请求中的`name`参数绑定到方法的`name`参数上。`required = false`表示这个参数是可选的。
  - `@RequestParam(required = false) String email`：这同样是一个请求参数，它将HTTP请求中的`email`参数绑定到方法的`email`参数上。`required = false`表示这个参数也是可选的。

  > 可选的原因：这些参数可以是空的，也可以是非空

  方法二：允许使用Json，同时也允许name、email是空的

```
@PutMapping("/student/{id}")  新*
public Response<StudentDTO> updateStudent(@PathVariable int id, @RequestBody StudentDTO studentDTO) {
    return Response.newSuccess(studentService.updateStudentById(id, studentDTO.getName(), studentDTO.getEmail()));
}
```

- Service层：

  StudentService接口：

```java
public interface StudentService {  3 个用法  1 个实现  新 *
//    Student getStudentById(int id);
    StudentDTO getStudentById(int id);  1 个用法  1 个实现  新 *

    Integer addNewStudent(StudentDTO studentDTO);  1 个用法  1 个实现  新 *

    void delStudent(int id);  1 个用法  1 个实现  新 *

    StudentDTO updateStudentById(int id, String name, String email);  1 个用法  1 个实现  新 *
}
```

StudentServiceImpl：

```java
@Override  1 个用法  新 *
@Transactional
public StudentDTO updateStudentById(int id, String name, String email) {
    Student studentInDB = studentRepository.findById(id).orElseThrow(() -> new IllegalStateException("id " + id + "doesn't exist"));
    if(StringUtils.hasLength(name) && !studentInDB.getName().equals(name)){
        studentInDB.setName(name);
    }
    if(StringUtils.hasLength(email) && !studentInDB.getEmail().equals(email)){
        studentInDB.setEmail(email);
    }
    Student student = studentRepository.save(studentInDB);
    return StudentConverter.convertStudent(student);
}
```

- Transactional注解：用于更新中途中失败的话，回滚事务
- 如果name或者email是空的或者和原先的相同，就不需要进行设置了

  > 注意这个地方判断是否是null/空，不能直接使用email.isempty()，因为如果是空的话就会报错

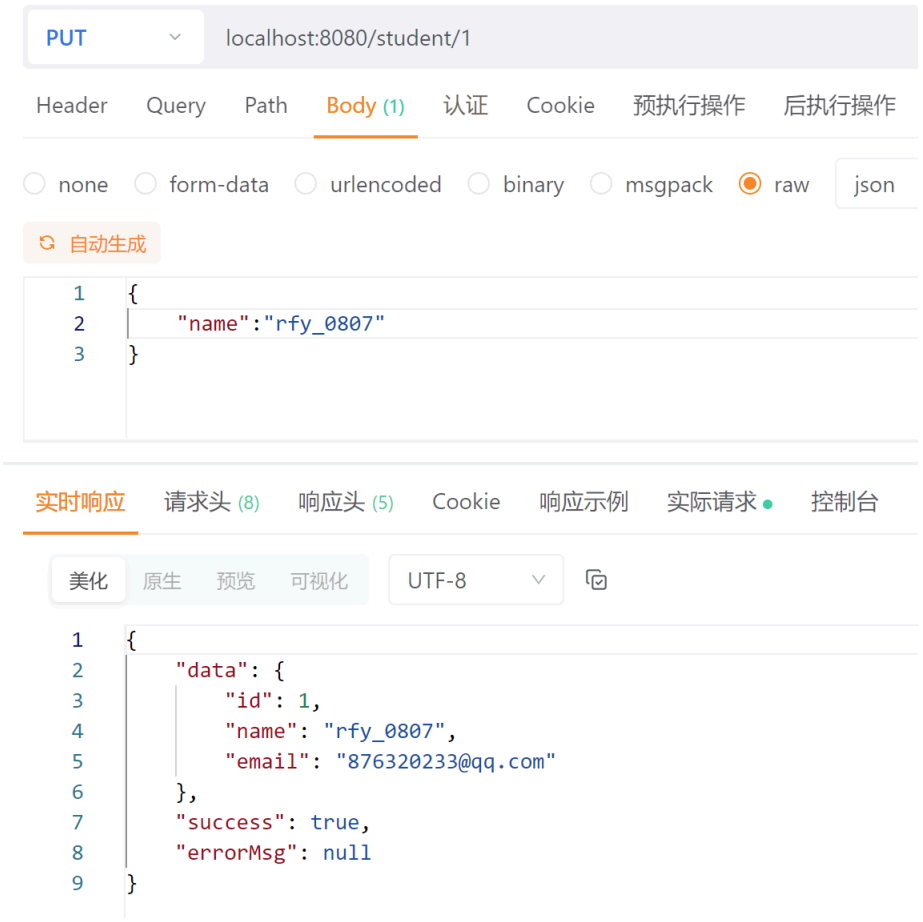
- 最后设置好后返回对应的StudentConverter对象

> 关于StudentRepository的save方法：
>
> `save` 方法是 `JpaRepository` 中的一个方法，用于保存或更新实体。具体来说，`save` 方法的行为如下：
>
> 1. **保存新实体**：如果传递给 `save` 方法的实体在其 ID 属性上为 `null` 或者尚未被持久化（即，它是一个新实体），`save` 方法会将其保存到数据库中，并返回一个带有生成的 ID 的实体。这通常涉及到一个 `INSERT` 操作。
> 2. **更新现有实体**：如果传递给 `save` 方法的实体已经有一个非 `null` 的 ID 属性，并且该实体已经被持久化（即，它是一个已存在的实体），`save` 方法会更新数据库中的实体，并返回更新后的实体。这涉及到一个 `UPDATE` 操作。

进行测试：

- 首先进行Get：

- 然后PUT请求，修改名字信息成功

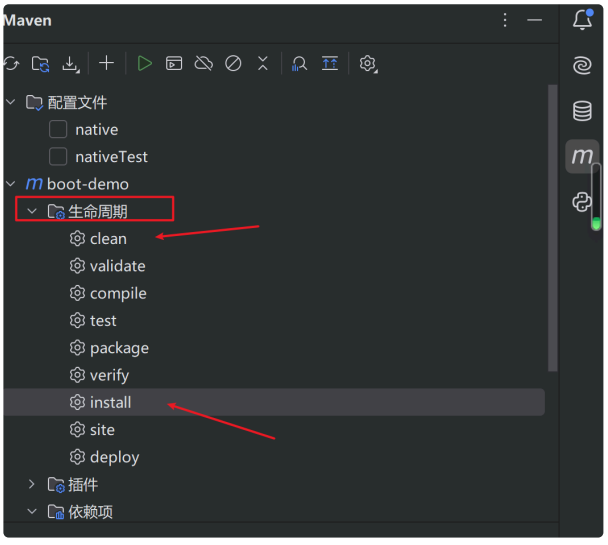


- 继续PUT请求，这次添加email信息，更新成功

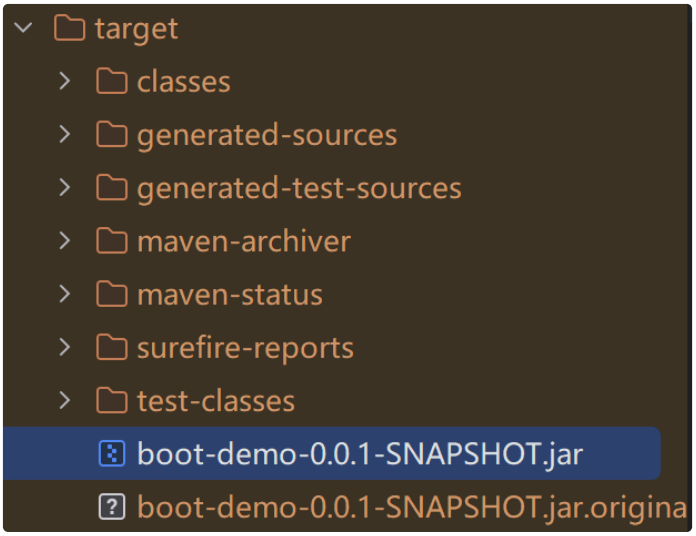
项目打包

方法一：使用IDEA的插件Maven

点击生命周期，点击clean先清除target包，然后install安装jar包



然后在target目录下可以看到生成了jar包，然后cd到target目录，然后执行`java -jar ...`，其中`...`表示打包的jar包的名称，这样相当于我们启动了后端。




```
(base) PS D:\Desktop\IDEA_workplace\boot-demo\boot-demo\target> java -jar boot-demo-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.3.5)
```

> 注意需要修改本地java版本到17，否则可能会有一些问题

支持修改端口，在原先`java -jar ...`后添加指令`--server.port = xxxx`即可。

方法二：当java版本与maven版本对应的时候，我这里是java17，使用命令`mvn clean install`也能达到相应的效果。

至此，我们已经完成了入门项目的后端建设，具体前端的细节就可以参考Vue项目等，如何实现前后端的交互，发送Http请求等。

完结撒花~