SpringBoot 业务类项目注解

1. @SpringBootApplication

这里先单独拎出@SpringBootApplication 注解说一下，虽然我们一般不会主动去使用它。

```
@SpringBootApplication
public class SpringSecurityJwtGuideApplication {
    public static void main(java.lang.String[] args) {
        SpringApplication.run(SpringSecurityJwtGuideApplication.class, args);
    }
}
```

我们可以把 @SpringBootApplication 看作是 @Configuration、@EnableAutoConfiguration、@ComponentScan 注解的集合。

```
package org.springframework.boot.autoconfigure;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes =
        AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
.....
}

package org.springframework.boot;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

根据 SpringBoot 官网，这三个注解的作用分别是：

@EnableAutoConfiguration：启用 SpringBoot 的自动配置机制
@ComponentScan： 扫描被@Component (@Service,@Controller)注解的 bean，注解默认会扫描该类所在的包下所有的类。
@Configuration：允许在 Spring 上下文中注册额外的 bean 或导入其他配置类

2. Spring Bean 相关

2.1. @Autowired

自动导入对象到类中，被注入进的类同样要被 Spring 容器管理比如：Service 类注入到 Controller 类中。

```
@Service
public class UserService {
......
}

@RestController
@RequestMapping("/users")
public class UserController {
@Autowired
private UserService userService;
......
}
```

2.2. Component,@Repository,@Service, @Controller

我们一般使用 @Autowired 注解让 Spring 容器帮我们自动装配 bean。要想把类标识成可用于 @Autowired 注解自动装配的 bean 的类,

可以采用以下注解实现：

@Component ：通用的注解，可标注任意类为 Spring 组件。如果一个 Bean 不知道属于哪个层，可以使用@Component 注解标注。

@Repository : 对应持久层即 Dao 层，主要用于数据库相关操作。

@Service : 对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。

@Controller : 对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

2.3. @RestController

@RestController 注解是@Controller 和@ResponseBody 的合集,表示这是个控制器 bean, 并且是将函数的返回值直 接填入 HTTP 响应体中,

是 REST 风格的控制器。

单独使用 @Controller 不加 @ResponseBody 的话一般使用在要返回一个视图的情况，这种情况属于比较传统的 Spring MVC 的应用，

对应于前后端不分离的情况。@Controller +@ResponseBody 返回 JSON 或 XML 形式数据

关于@RestController 和 @Controller 的对比，请看这篇文章：@RestController vs @Controller。

2.4. @Scope

声明 Spring Bean 的作用域，使用方法:

```
@Bean
@Scope("singleton")
public Person personSingleton() {
    return new Person();
}
```

四种常见的 Spring Bean 的作用域：

singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
prototype : 每次请求都会创建一个新的 bean 实例。
request : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效。
session : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效。

2.5. Configuration

一般用来声明配置类，可以使用 @Component 注解替代，不过使用 Configuration 注解声明配置类更加语义化。

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

}
```

3. 处理常见的 HTTP 请求类型

5 种常见的请求类型:

GET ：请求从服务器获取特定资源。举个例子：GET /users（获取所有学生）
POST ：在服务器上创建一个新的资源。举个例子：POST /users（创建学生）
PUT ：更新服务器上的资源（客户端提供更新后的整个资源）。举个例子：PUT /users/12（更新编号为 12 的学生）
DELETE ：从服务器删除特定的资源。举个例子：DELETE /users/12（删除编号为 12 的学生）

PATCH ：更新服务器上的资源（客户端提供更改的属性，可以看做作是部分更新），使用的比较少，这里就不举例子了。

3.1. GET 请求

@GetMapping("users") 等价于 @RequestMapping(value="/users",method=RequestMethod.GET)

```
@GetMapping("/users")
public ResponseEntity<List<User>> getAllUsers() {
return userRepository.findAll();
}
```

3.2. POST 请求

@PostMapping("users") 等价于 @RequestMapping(value="/users",method=RequestMethod.POST)

关于@RequestBody 注解的使用，在下面的“前后端传值”这块会讲到。

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody UserCreateRequest
                    userCreateRequest) {
      return userRespository.save(user);
}
```

3.3. PUT 请求

@PutMapping("/users/{userId}") 等价于 @RequestMapping(value="/users/{userId}",method=RequestMethod.PUT)

```
@PutMapping("/users/{userId}")
public ResponseEntity<User> updateUser(@PathVariable(value = "userId") Long userId,
@Valid @RequestBody UserUpdateRequest userUpdateRequest) {
.....
}
```

3.4. DELETE 请求

@DeleteMapping("/users/{userId}")等价于 @RequestMapping(value="/users/{userId}",method=RequestMethod.DELETE)

```
@DeleteMapping("/users/{userId}")
public ResponseEntity deleteUser(@PathVariable(value = "userId") Long userId){
.....
}
```

3.5. PATCH 请求

一般实际项目中，我们都是 PUT 不够用了之后才用 PATCH 请求去更新数据。

```
@PatchMapping("/profile")
public ResponseEntity updateStudent(@RequestBody StudentUpdateRequest
        studentUpdateRequest) {
    studentRepository.updateDetail(studentUpdateRequest);
    return ResponseEntity.ok().build();
}
```

4. 前后端传值

掌握前后端传值的正确姿势，是你开始 CRUD 的第一步！

4.1. @PathVariable 和 @RequestParam

@PathVariable 用于获取路径参数，@RequestParam 用于获取查询参数。

举个简单的例子：

```
@GetMapping("/klasses/{klassId}/teachers")
public List<Teacher> getKlassRelatedTeachers(
        @PathVariable("klassId") Long klassId,
        @RequestParam(value = "type", required = false) String type ) {
...
}
```

如果我们请求的 url 是：/klasses/{123456}/teachers?type=web

那么我们服务获取到的数据就是：klassId=123456,type=web。

4.2. @RequestBody

用于读取 Request 请求（可能是 POST,PUT,DELETE,GET 请求）的 body 部分并且 Content-Type 为 application/json 格式的数据，

接收到数据之后会自动将数据绑定到 Java 对象上去。系统会使用 HttpMessageConverter 或者自定义的 HttpMessageConverter 将请求的

body 中的 json 字符串转换为 java 对象。

我用一个简单的例子来给演示一下基本使用！

我们有一个注册的接口：

```
@PostMapping("/sign-up")
public ResponseEntity signUp(@RequestBody @Valid UserRegisterRequest
    userRegisterRequest) {
```

```
userService.save(userRegisterRequest);
return ResponseEntity.ok().build();
}
```

UserRegisterRequest 对象：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserRegisterRequest {
    @NotBlank
    private String userName;
    @NotBlank
    private String password;
    @FullName
    @NotBlank
    private String fullName;
}
```

我们发送 post 请求到这个接口，并且 body 携带 JSON 数据：

```
{"userName":"coder","fullName":"shuangkou","password":"123456"}
```

这样我们的后端就可以直接把 json 格式的数据映射到我们的 UserRegisterRequest 类上。

需要注意的是：一个请求方法只可以有一个@RequestBody，但是可以有多个@RequestParam 和@PathVariable。

如果你的方法必须要用两个 @RequestBody 来接受数据的话，大概率是你的数据库设计或者系统设计出问题了！