swagger的学习文档

# 相关网站

Swagger官网 ：<http://swagger.io/>

教程参考网站：<https://blog.csdn.net/sanyaoxu_2/article/details/80555328>

<http://blog.didispace.com/springbootswagger2/>

<https://github.com/swagger-api/swagger-core/wiki/Annotations>

<https://github.com/springfox/springfox>

# 简介

Swagger 是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。总体目标是使客户端和文件系统作为服务器以同样的速度来更新。文件的方法，参数和模型紧密集成到服务器端的代码，允许API来始终保持同步。

**作用：**

    1. 接口的文档在线自动生成。

    2. 功能测试。

**Swagger是一组开源项目，其中主要要项目如下：**

1.   **Swagger-tools**:提供各种与Swagger进行集成和交互的工具。例如模式检验、Swagger 1.2文档转换成Swagger 2.0文档等功能。

2.   **Swagger-core**: 用于Java/Scala的的Swagger实现。与JAX-RS(Jersey、Resteasy、CXF...)、Servlets和Play框架进行集成。

3.   **Swagger-js**: 用于JavaScript的Swagger实现。

4.   **Swagger-node-express**: Swagger模块，用于node.js的Express web应用框架。

5.   **Swagger-ui**：一个无依赖的HTML、JS和CSS集合，可以为Swagger兼容API动态生成优雅文档。

6.   **Swagger-codegen**：一个模板驱动引擎，通过分析用户Swagger资源声明以各种语言生成客户端代码

# Spring Boot中使用Swagger2

## 添加Swagger2依赖

在pom.xml中加入Swagger2的依赖

|  |
| --- |
| <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.7.0</version>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.7.0</version>  </dependency> |

完整的POM.xml文件

<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 http://maven.apache.org/maven-v4\_0\_0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>org.lys.demo</groupId>

<artifactId>demo-swagger-n0001</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>demo-swagger-n0001 Maven Webapp</name>

<url>http://maven.apache.org</url>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.0.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

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

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.7.0</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.7.0</version>

</dependency>

</dependencies>

<build>

<finalName>demo-swagger-n0001</finalName>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

## 创建Application.java

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

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

@SpringBootApplication

**public** **class** Application {

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

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

}

}

## 创建Swagger2配置类

在Application.java同级创建Swagger2的配置类Swagger2。

@Configuration

@EnableSwagger2

**public** **class** Swagger2 {

@Bean

**public** Docket createRestApi() {

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.apiInfo(apiInfo())

.select()

.apis(RequestHandlerSelectors.*basePackage*("top.linyisong.swagger"))

.paths(PathSelectors.*any*())

.build();

}

**private** ApiInfo apiInfo() {

**return** **new** ApiInfoBuilder()

.title("Spring Boot中使用Swagger2构建RESTful APIs")

.description("更多Spring Boot相关文章")

.contact(**new** Contact("作者", "", ""))

.version("1.0")

.build();

}

}

如上代码所示，通过@Configuration注解，让Spring来加载该类配置。再通过@EnableSwagger2注解来启用Swagger2。

再通过createRestApi函数创建Docket的Bean之后，apiInfo()用来创建该Api的基本信息（这些基本信息会展现在文档页面中）。select()函数返回一个ApiSelectorBuilder实例用来控制哪些接口暴露给Swagger来展现，本例采用指定扫描的包路径来定义，Swagger会扫描该包下所有Controller定义的API，并产生文档内容（除了被@ApiIgnore指定的请求）。

## 添加文档内容

在完成了上述配置后，其实已经可以生产文档内容，但是这样的文档主要针对请求本身，而描述主要来源于函数等命名产生，对用户并不友好，我们通常需要自己增加一些说明来丰富文档内容。如下所示，我们通过@ApiOperation注解来给API增加说明、通过@ApiImplicitParams、@ApiImplicitParam注解来给参数增加说明。

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RequestBody;

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

**import** org.springframework.web.bind.annotation.RequestMethod;

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

**import** io.swagger.annotations.Api;

**import** io.swagger.annotations.ApiImplicitParam;

**import** io.swagger.annotations.ApiImplicitParams;

**import** io.swagger.annotations.ApiOperation;

**import** top.linyisong.swagger.model.User;

@RestController

@RequestMapping(value="/users") // 通过这里配置使下面的映射都在/users下，可去除

@Api(value = "/users", tags = "用户管理")

**public** **class** UserController {

**static** Map<Long, User> *users* = Collections.*synchronizedMap*(**new** HashMap<Long, User>());

@ApiOperation(value="获取用户列表", notes="")

@RequestMapping(value={""}, method=RequestMethod.***GET***)

**public** List<User> getUserList() {

List<User> r = **new** ArrayList<User>(*users*.values());

**return** r;

}

@ApiOperation(value="创建用户", notes="根据User对象创建用户")

@RequestMapping(value="", method=RequestMethod.***POST***)

**public** String postUser(@RequestBody User user) {

*users*.put(user.getId(), user);

**return** "success";

}

@ApiOperation(value="获取用户详细信息", notes="根据url的id来获取用户详细信息")

@ApiImplicitParam(name = "id", value = "用户ID", required = **true**, dataType = "Long", paramType = "path")

@RequestMapping(value="/{id}", method=RequestMethod.***GET***)

**public** User getUser(@PathVariable Long id) {

**return** *users*.get(id);

}

@ApiOperation(value="更新用户详细信息", notes="根据url的id来指定更新对象，并根据传过来的user信息来更新用户详细信息")

@ApiImplicitParams({

@ApiImplicitParam(name = "id", value = "用户ID", required = **true**, dataType = "Long", paramType = "path"),

@ApiImplicitParam(name = "user", value = "用户详细实体user", required = **true**, dataType = "User")

})

@RequestMapping(value="/{id}", method=RequestMethod.***PUT***)

**public** String putUser(@PathVariable Long id, @RequestBody User user) {

User u = *users*.get(id);

u.setName(user.getName());

u.setAge(user.getAge());

*users*.put(id, u);

**return** "success";

}

@ApiOperation(value="删除用户", notes="根据url的id来指定删除对象")

@ApiImplicitParam(name = "id", value = "用户ID", required = **true**, dataType = "Long", paramType = "path")

@RequestMapping(value="/{id}", method=RequestMethod.***DELETE***)

**public** String deleteUser(@PathVariable Long id) {

*users*.remove(id);

**return** "success";

}

}

**import** io.swagger.annotations.ApiModel;

**import** io.swagger.annotations.ApiModelProperty;

@ApiModel(value="用户")

**public** **class** User {

@ApiModelProperty(value="id")

**private** Long id;

@ApiModelProperty(required=**true**,value="姓名",example="张三")

**private** String name;

@ApiModelProperty(required=**true**,value="年龄",allowableValues="range[0, 150]",example="18")

**private** Integer age;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

}

## 注解说明

**Swagger使用的注解及其说明：**

[@Api](https://my.oschina.net/u/2396174)：用在类上，说明该类的作用。

@ApiOperation：注解来给API增加方法说明。

@ApiImplicitParams : 用在方法上包含一组参数说明。

@ApiImplicitParam：用来注解来给方法入参增加说明。

@ApiResponses：用于表示一组响应

@ApiResponse：用在@ApiResponses中，一般用于表达一个错误的响应信息

    l   **code**：数字，例如400

    l   **message**：信息，例如"请求参数没填好"

    l   **response**：抛出异常的类

@ApiModel：描述一个Model的信息（一般用在请求参数无法使用@ApiImplicitParam注解进行描述的时候）

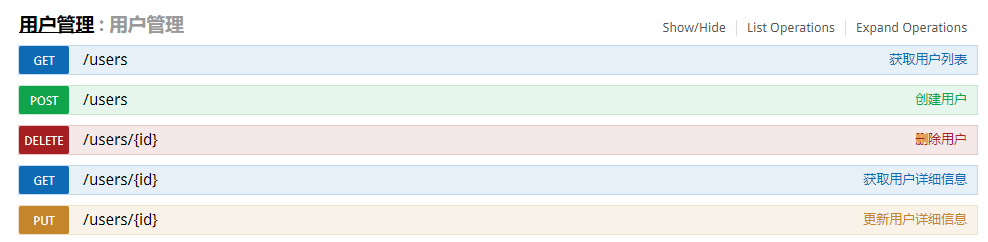
    l   **@ApiModelProperty**：描述一个model的属性

注意：@ApiImplicitParam的参数说明：

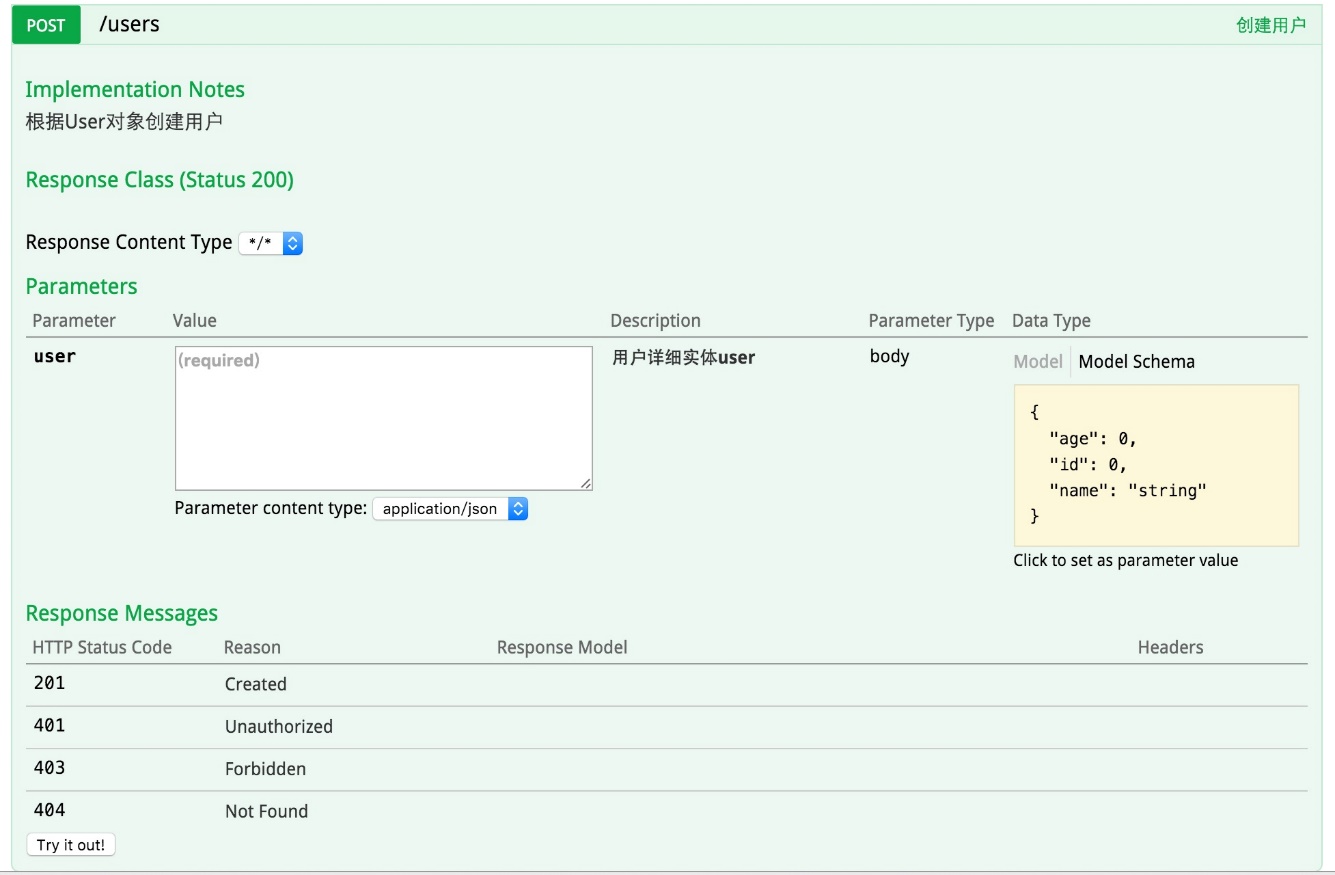
|  |  |
| --- | --- |
| **paramType**：指定参数放在哪个地方 | header：请求参数放置于Request Header，使用@RequestHeader获取  query：请求参数放置于请求地址，使用@RequestParam获取  path：（用于restful接口）-->请求参数的获取：@PathVariable  body：（不常用）  form（不常用） |
| name：参数名 |  |
| dataType：参数类型 |  |
| required：参数是否必须传 | true | false |
| value：说明参数的意思 |  |
| defaultValue：参数的默认值 |  |

## API文档访问与调试

启动Spring Boot程序，访问：<http://localhost:8080/swagger-ui.html>。就能看到前文所展示的RESTful API的页面。



我们可以再点开具体的API请求，以POST类型的/users请求为例，可找到上述代码中我们配置的Notes信息以及参数user的描述信息，如下图所示。

[](http://blog.didispace.com/content/images/2016/04/swagger2_2.png)

在上图请求的页面中，我们看到user的Value是个输入框？是的，Swagger除了查看接口功能外，还提供了调试测试功能，我们可以点击上图中右侧的Model Schema（黄色区域：它指明了User的数据结构），此时Value中就有了user对象的模板，我们只需要稍适修改，点击下方“Try it out！”按钮，即可完成了一次请求调用！

此时，你也可以通过几个GET请求来验证之前的POST请求是否正确。

相比为这些接口编写文档的工作，我们增加的配置内容是非常少而且精简的，对于原有代码的侵入也在忍受范围之内。因此，在构建RESTful API的同时，加入swagger来对API文档进行管理，是个不错的选择。

## 代码示例

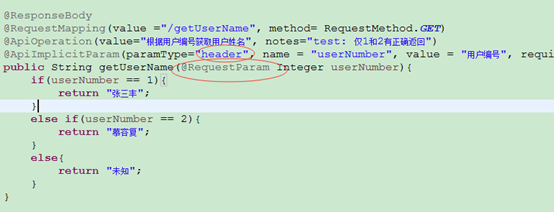
本文的相关例子可以查看下面仓库中的chapter3-1-5目录：

* Github：<https://github.com/dyc87112/SpringBoot-Learning>
* Gitee：<https://gitee.com/didispace/SpringBoot-Learning>

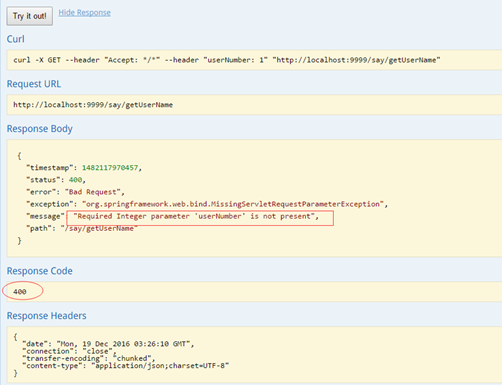
## 注意点

1.  paramType会直接影响程序的运行期，如果paramType与方法参数获取使用的注解不一致，会直接影响到参数的接收。

例如：



**使用Sawgger UI进行测试，接收不到！**



**2.  还有一个需要注意的地方：**

Conntroller中定义的方法必须在@RequestMapper中显示的指定RequestMethod类型，否则SawggerUi会默认为全类型皆可访问， API列表中会生成多条项目。



如上图：updatePassword()未指定requestMethod，结果生成了7条API信息。所以如果没有特殊需求，建议根据实际情况加上requestMethod。

