# **SwiftBoot-Web**

SwiftBoot 的主模块，开发 Web 应用需要引用。

## **依赖**

* Servlet >= 3.1
* Spring Framework >= 5.1.x.RELEASE
* Spring MVC >= 5.1.x.RELEASE
* Spring Boot >= 2.1.x.RELEASE
* Spring Data JPA >= 2.1.x.RELEASE

## **特性**

* 开箱即用，用更少的代码完成更多的工作。SwiftBoot 是无侵入性的，你可以只使用其中的一部分功能，也可以随时去掉它或者切换为别的框架。
* 统一固定的返回值格式:

{

"code": "<4位的错误代码>",

"msg" : "<错误消息>",

"content": {

"自定义的JSON格式的返回内容"

}

}

* 统一的接口参数对象 HttpCommand。子类 BasePopulateCommand 可实现将接口参数自动填充到实体类中。
* 统一的接口返回对象基类 BasePopulateResult，可实现自动将实体类的属性值填充到返回值中。
* 提供了统一的控制器（Controller）异常处理，自动将未处理的异常转换成 JSON 格式的接口响应对象返回给客户端。
* 定义了实体类基类，包含了必须的和大多数表都需要的字段，所有的实体类都继承它们。包含字段：
  + 主键：ID
  + 创建时间：CREATE\_TIME
  + 更新时间：UPDATE\_TIME
  + 是否逻辑删除：IS\_DELETE
* 自动处理接口参数验证结果，转换为 JSON 格式的统一格式；扩展的表单验证器，可验证：手机号，包含大写数字，包含数字，包含特殊符号。

## **引用 jar 包：**

Maven:

<dependency>

<groupId>com.github.swiftech</groupId>

<artifactId>swiftboot-web</artifactId>

<version>1.0.9-SNAPSHOT</version>

</dependency>

## **如何使用**

### **Model层**

* Dao

SwiftBoot 使用 Spring Data JPA 来实现 Model 层，所有 Dao 接口必须继承 CrudRepository 接口或者其子接口

public interface OrderDao extends CrudRepository<OrderEntity, String> {

}

* 实体类

SwiftBoot 要求实体类必须继承 BaseIdEntity 或者其子类 BaseEntity，BaseIdEntity 定义了主键字段 ID，BaseEntity继承 BaseIdEntity 并定义了创建时间 CREATE\_TIME、更新时间 UPDATE\_TIME、是否逻辑删除 IS\_DELETE 三个字段

@Entity

@Table(name = "DEMO\_ORDER")

public class OrderEntity extends BaseEntity {

@Column(name = "ORDER\_CODE", length = 16, columnDefinition = "COMMENT '订单编号'")

private String orderCode;

@Column(name = "DESCRIPTION", length = 64, columnDefinition = "COMMENT '订单描述'")

private String description;

}

* 主键ID

对于实体类的主键 ID 的赋值，你有两种选择：

* 自行创建 ID 并赋值给实体类，例如：

entity.setId(IdUtils.makeUUID()); // 生成 UUID 主键

* 配置自动创建ID

swiftboot:

web:

model:

autoGenerateId: true

那么，在通过 Dao 的 save() 或 saveAll() 方法保存实体对象的时候，SwiftBoot 默认情况下会给实体类自动生成 UUID 主键。另外，SwiftBoot 提供了一个更好的主键ID生成器 EntityIdGenerator，它可以生成格式为 **业务代码+时间戳+随机字符串** 的长度为32字节的主键ID，例如：order20190422170606462gbxudaaxgt，这个主键既有UUID的优点但是比UUID更容易识别并且带来更好的性能。启用这个 ID 生成器只要配置：

@Bean

IdGenerator idGenerator() {

return new EntityIdGenerator();

}

### **控制器 Controller**

* 所有的控制器接口返回统一定义的响应对象 HttpResponse，包含错误代码、错误消息以及泛型表达的接口返回值。POST 接口的所有输入参数对象继承 HttpCommand 或者它的子类。

例如一个创建订单的接口如下：

@RequestMapping(value = "order/create", method = RequestMethod.POST)

public

@ResponseBody HttpResponse<OrderCreateResult> orderCreate(

@RequestBody @Validated @ApiParam("创建订单参数") OrderCreateCommand command) {

OrderCreateResult ret = orderService.createOrder(command);

return new HttpResponse<>(ret);

}

SpringMVC 的 @ResponseBody 注解会把方法返回的 HttpResponse 对象及其内嵌的对象一起转换成 JSON 格式返回给访问接口的客户端。

@ApiModel

public class OrderCreateCommand extends BasePopulateCommand<OrderEntity> {

@ApiModelProperty(value = "订单编号", example = "2019032411081201")

@JsonProperty("order\_code")

@Length(max = 16)

private String orderCode;

@ApiModelProperty(value = "订单描述", example = "越快越好")

@JsonProperty("description")

@Length(max = 64)

private String description;

}

* 控制器中抛出的异常直接抛出会使得客户端的错误展示非常不友好，而通过代码去捕获即繁琐又容易遗留，SwiftBoot 实现了控制器增强 ExceptionProcessor，他将异常信息以统一的 JSON 格式输出给客户端，配置方法如下：

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = {"org.swiftboot.web"})

public class MyDemoConfig {

}

或者

@Configuration

@EnableWebMvc

public class MyDemoConfig {

@Bean

ExceptionProcessor exceptionProcessor() {

return new ExceptionProcessor();

}

}

* 输入参数验证

除了常规的异常处理增强之外，SwiftBoot 还实现了 ValidationExceptionProcessor 控制器增强来处理验证异常信息的转换。它会捕获验证框架抛出的异常，并把异常转换为 SwiftBoot 定义的 JSON 输出格式。（自动从 Command 对象的注解中获取到参数对应的描述信息） 配置：

@Bean

ValidationExceptionProcessor validationExceptionProcessor() {

return new ValidationExceptionProcessor();

}

如果接口参数中有 BindingResult 这个参数，那么验证异常就不会抛出，此时可以在控制器类上添加注解 @ConvertValidateResult 来标识需要拦截并抛出 ValidationException 异常。这个注解也可以加在控制器方法上，只有该方法执行的时候才会进行增强处理。

@Controller

@RequestMapping("/order")

@ConvertValidateResult

public class OrderController {

}

### **Service 层**

Web 开发中最无趣的工作之一就是从接口参数对象中复制每个参数值到 Dao 层的实体类中进行保存，反之亦然。 SwiftBoot 实现了自动化的参数填充，它能够有选择性的将参数值填充到对应的实体类中，也能将实体类中的值填充到返回值对象中（如果实体类关联了其他实体类对象，它也会对应的填充到返回值对象的内嵌对象中去）。

* 输入参数自动填充实现方法：
  + 输入参数对象继承 BasePopulateCommand
  + 对于新建数据的操作，调用 createEntity() 方法即可实例化相对应的实体类，并把输入参数对象中所有名称对应的值填充到实体类中。
  + 对于修改数据的操作，查询出需要修改的实体类之后，调用 populateEntity() 方法将输入参数对象中所有名称对应的值填充到实体类中。
* 输出参数自动填充实现方法：
  + 返回值对象继承 BasePopulateResult
  + 在需要的地方调用 BasePopulateResult 的静态方法 createResult() 即可实例化返回值对象，并把将查询到的实体类中所有对应名称的值（包括一对一、一对多关联的实体类）填充到输出对象中。或者在代码中直接实例化返回对象实例，然后调用它的 populateByEntity() 方法进行填充。

样例：

@Service

public class OrderServiceImpl implements OrderService {

@Override

public OrderCreateResult createOrder(OrderCreateCommand cmd) {

OrderEntity p = cmd.createEntity();

p.setId(IdUtils.makeUUID()); // 如果设置了自动生成 ID 就不需要这一行

OrderEntity saved = orderDao.save(p);

return new OrderCreateResult(saved.getId());

}

@Override

public OrderResult queryOrder(String orderId) {

Optional<OrderEntity> optEntity = orderDao.findById(orderId);

if (optEntity.isPresent()) {

return OrderResult.createResult(OrderResult.class, optEntity.get());

}

return null;

}

}

输入参数对象类定义：

@ApiModel

public class OrderCreateCommand extends BasePopulateCommand<OrderEntity> {

@ApiModelProperty(value = "订单编号", example = "2019032411081201")

@JsonProperty("order\_code")

@Length(max = 16)

private String orderCode;

@ApiModelProperty(value = "订单描述", example = "越快越好")

@JsonProperty("description")

@Length(max = 64)

private String description;

}

返回对象类定义：

public class OrderResult extends BasePopulateResult {

@ApiModelProperty(value = "订单编号", example = "2019032411081201")

@JsonProperty("order\_code")

private String orderCode;

@ApiModelProperty(value = "订单描述", example = "越快越好")

@JsonProperty("description")

private String description;

}