第 2-8 课：Spring Boot 构建一个 RESTful Web 服务

现在越来越多的企业推荐使用 RESTful 风格来构建企业的应用接口，那么什么是 RESTful 呢?

什么是 RESTful

RESTful 是目前最流行的一种互联网软件架构。REST（Representational State Transfer，表述性状态转移）一词是由 Roy Thomas Fielding 在他 2000 年博士论文中提出的，定义了他对互联网软件的架构原则，如果一个架构符合 REST 原则，则称它为 RESTful 架构。

RESTful 架构一个核心概念是“资源”（Resource）。从 RESTful 的角度看，网络里的任何东西都是资源，它可以是一段文本、一张图片、一首歌曲、一种服务等，每个资源都对应一个特定的 URI（统一资源定位符），并用它进行标示，访问这个 URI 就可以获得这个资源。

资源可以有多种表现形式，也就是资源的“表述”（Representation），比如一张图片可以使用 JPEG 格式也可以使用 PNG 格式。URI 只是代表了资源的实体，并不能代表它的表现形式。

互联网中，客户端和服务端之间的互动传递的就只是资源的表述，我们上网的过程，就是调用资源的 URI，获取它不同表现形式的过程。这种互动只能使用无状态协议 HTTP，也就是说，服务端必须保存所有的状态，客户端可以使用 HTTP 的几个基本操作，包括 GET（获取）、POST（创建）、PUT（更新）与 DELETE（删除），使得服务端上的资源发生“状态转化”（State Transfer），也就是所谓的“表述性状态转移”。

Spring Boot 对 RESTful 的支持

Spring Boot 全面支持开发 RESTful 程序，通过不同的注解来支持前端的请求，除了经常使用的注解外，Spring Boot 还提了一些组合注解。这些注解来帮助简化常用的 HTTP 方法的映射，并更好地表达被注解方法的语义。

- @GetMapping，处理 Get 请求
- @PostMapping，处理 Post 请求
- @PutMapping，用于更新资源
- @DeleteMapping，处理删除请求
- @PatchMapping，用于更新部分资源

其实这些组合注解就是我们使用的 @RequestMapping 的简写版本，下面是 Java 类中的使用示例：

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

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

@PutMapping(value="/xxx")
等价于
@RequestMapping(value = "/xxx",method = RequestMethod.PUT)

@DeleteMapping(value="/xxx")
等价于
@RequestMapping(value = "/xxx",method = RequestMethod.DELETE)

@PatchMapping(value="/xxx")
等价于
@RequestMapping(value = "/xxx",method = RequestMethod.PATCH)
```

通过以上可以看出 RESTful 在请求的类型中就指定了对资源的操控。

快速上手

按照 RESTful 的思想我们来设计一组对用户操作的 RESTful API：

请求	地址	说明
get	/messages	获取所有消息
post	/message	创建一个消息
put	/message	修改消息内容
patch	/message/text	修改消息的 text 字段
get	/message/id	根据 ID 获取消息
delete	/message/id	根据 ID 删除消息

put 方法主要是用来更新整个资源的，而 patch 方法主要表示更新部分字段。

开发实体列的操作

首先定义一个 Message 对象：

```
public class Message {
    private Long id;
    private String text;
    private String summary;
    // 省略 getter setter
}
```

我们使用 ConcurrentHashMap 来模拟存储 Message 对象的增删改查，AtomicLong 做为消息的自增组建来使用。ConcurrentHashMap 是 Java 中高性能并发的 Map 接口，AtomicLong 作用是对长整形进行原子操作，可以在高并场景下获取到唯一的 Long 值。

```
@Service("messageRepository")
public class InMemoryMessageRepository implements MessageRepository {

    private static AtomicLong counter = new AtomicLong();
    private final ConcurrentMap<Long, Message> messages = new ConcurrentHashMap<>(
);
}
```

查询所有用户，就是将 Map 中的信息全部返回。

```
@Override
public List<Message> findAll() {
    List<Message> messages = new ArrayList<Message>(this.messages.values());
    return messages;
}
```

保持消息时，需要判断是否存在 ID，如果没有，可以使用 AtomicLong 获取一个。

```
@Override
public Message save(Message message) {
    Long id = message.getId();
    if (id == null) {
        id = counter.incrementAndGet();
        message.setId(id);
    }
    this.messages.put(id, message);
    return message;
}
```

更新时直接覆盖对应的 Key：

```
@Override
public Message update(Message message) {
    this.messages.put(message.getId(), message);
    return message;
}
```

更新 text 字段：

```
@Override
public Message updateText(Message message) {
    Message msg=this.messages.get(message.getId());
    msg.setText(message.getText());
    this.messages.put(msg.getId(), msg);
    return msg;
}
```

最后封装根据 ID 查找和删除消息。

```
@Override
public Message findMessage(Long id) {
    return this.messages.get(id);
}

@Override
public void deleteMessage(Long id) {
    this.messages.remove(id);
}
```

封装 RESTful 的处理

将上面封装好的 MessageRepository 注入到 Controller 中，调用对应的增删改查方法。

```java
@RestController
@RequestMapping("/")
public class MessageController {

    @Autowired
    private  MessageRepository messageRepository;

    // 获取所有消息体
    @GetMapping(value = "messages")
    public List<Message> list() {
        List<Message> messages = this.messageRepository.findAll();
        return messages;
    }

    // 创建一个消息体
    @PostMapping(value = "message")
    public Message create(Message message) {
        message = this.messageRepository.save(message);
        return message;
    }

    // 使用 put 请求进行修改
    @PutMapping(value = "message")
    public Message modify(Message message) {
        Message messageResult=this.messageRepository.update(message);
        return messageResult;
    }

    // 更新消息的 text 字段
    @PatchMapping(value="/message/text")
    public Message patch(Message message) {
        Message messageResult=this.messageRepository.updateText(message);
        return messageResult;
    }

    @GetMapping(value = "message/{id}")
    public Message get(@PathVariable Long id) {
        Message message = this.messageRepository.findMessage(id);
        return message;
    }

    @DeleteMapping(value = "message/{id}")
    public void delete(@PathVariable("id") Long id) {
        this.messageRepository.deleteMessage(id);
    }
}
```

进行测试

我们使用 MockMvc 进行测试。MockMvc 实现了对 Http 请求的模拟，能够直接使用网络的形式，转换到 Controller 的调用，这样可以使得测试速度快、不依赖网络环境，而且提供了一套验证的工具，这样可以使得请求的验证统一而且很方便。

下面是 MockMvc 的主体架构：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MessageControllerTest {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }
}
```

- @SpringBootTest 注解是 SpringBoot 自 1.4.0 版本开始引入的一个用于测试的注解
- @RunWith(SpringRunner.class) 代表运行一个 Spring 容器
- @Before 代表在测试启动时候需要提前加载的内容，这里是提前加载 MVC 环境

1. 测试创建消息（post 请求）

我们先来测试创建一个消息体：

```
@Test
public void saveMessage() throws Exception {
    final MultiValueMap<String, String> params = new LinkedMultiValueMap<>();
    params.add("text", "text");
    params.add("summary", "summary");
    String mvcResult=  mockMvc.perform(MockMvcRequestBuilders.post("/message")
            .params(params)).andReturn().getResponse().getContentAsString();
    System.out.println("Result === "+mvcResult);
}
```

- MultiValueMap 用来存储需要发送的请求参数。
- MockMvcRequestBuilders.post 代表使用 post 请求。

运行这个测试后返回结果如下：

```
Result === {"id":10,"text":"text","summary":"summary","created":"2018-07-28T06:27:
23.176+0000"}
```

表明创建消息成功。

2. 批量添加消息体（post 请求）

为了方便后面测试，需要启动时在内存中存入一些消息来测试。

封装一个 saveMessages() 方法批量存储 9 条消息：

```
private void  saveMessages()  {
    for (int i=1;i<10;i++){
        final MultiValueMap<String, String> params = new LinkedMultiValueMap<>();
        params.add("text", "text"+i);
        params.add("summary", "summary"+i);
        try {
            MvcResult mvcResult=  mockMvc.perform(MockMvcRequestBuilders.post("/me
ssage")
                    .params(params)).andReturn();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

并且将 saveMessages() 方法添加到 setup() 中，这样启动测试的时候内存中就已经保存了一些数据。

```
@Before
public void setup() {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    saveMessages();
}
```

3. 测试获取所有消息（get 请求)

```
@Test
public void getAllMessages() throws Exception {
    String mvcResult= mockMvc.perform(MockMvcRequestBuilders.get("/messages"))
            .andReturn().getResponse().getContentAsString();
    System.out.println("Result === "+mvcResult);
}
```

运行后返回结果：

```
Result === [{"id":1,"text":"text1","summary":"summary1","created":"2018-07-28T06:34:20.583+0000"},{"id":2,"text":"text2","summary":"summary2","created":"2018-07-28T06:34:20.675+0000"},{"id":3,"text":"text3","summary":"summary3","created":"2018-07-28T06:34:20.677+0000"},{"id":4,"text":"text4","summary":"summary4","created":"2018-07-28T06:34:20.678+0000"},{"id":5,"text":"text5","summary":"summary5","created":"2018-07-28T06:34:20.680+0000"},{"id":6,"text":"text6","summary":"summary6","created":"2018-07-28T06:34:20.682+0000"},{"id":7,"text":"text7","summary":"summary7","created":"2018-07-28T06:34:20.684+0000"},{"id":8,"text":"text8","summary":"summary8","created":"2018-07-28T06:34:20.685+0000"},{"id":9,"text":"text9","summary":"summary9","created":"2018-07-28T06:34:20.687+0000"}]
```

可以看出初始化的数据已经保存到内存 Map 中，另一方面表明获取数据测试成功。

4. 测试获取单个消息（get 请求)

```
@Test
public void getMessage() throws Exception {
    String mvcResult= mockMvc.perform(MockMvcRequestBuilders.get("/message/6"))
            .andReturn().getResponse().getContentAsString();
    System.out.println("Result === "+mvcResult);
}
```

上面代码表明获取 ID 为 6 的消息。

运行后返回结果：

```
Result === {"id":6,"text":"text6","summary":"summary6","created":"2018-07-28T06:37:26.014+0000"}
```

5. 测试修改（put 请求）

```
@Test
public void modifyMessage() throws Exception {
    final MultiValueMap<String, String> params = new LinkedMultiValueMap<>();
    params.add("id", "6");
    params.add("text", "text");
    params.add("summary", "summary");
    String mvcResult= mockMvc.perform(MockMvcRequestBuilders.put("/message").params(params))
            .andReturn().getResponse().getContentAsString();
    System.out.println("Result === "+mvcResult);
}
```

上面代码更新 ID 为 6 的消息体。

运行后返回结果：

```
Result === {"id":6,"text":"text","summary":"summary","created":"2018-07-28T06:38:3
2.277+0000"}
```

我们发现 ID 为 6 的消息 text 字段值由 text6 变为 text，summary 字段值由 summary6 变为 summary，表示消息更新成功。

6. 测试局部修改（patch 请求)

```
@Test
public void patchMessage() throws Exception {
    final MultiValueMap<String, String> params = new LinkedMultiValueMap<>();
    params.add("id", "6");
    params.add("text", "text");
    String mvcResult= mockMvc.perform(MockMvcRequestBuilders.patch("/message/text"
).params(params))
            .andReturn().getResponse().getContentAsString();
    System.out.println("Result === "+mvcResult);
}
```

同样是更新 ID 为 6 的消息体，但只是更新消息属性的一个字段。

运行后返回结果：

```
Result === {"id":6,"text":"text","summary":"summary6","created":"2018-07-28T06:41:
51.816+0000"}
```

这次发现只有 text 字段值由 text6 变为 text，summary 字段值没有变化，表明局部更新成功。

7. 测试删除（delete 请求）

```
@Test
public void deleteMessage() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.delete("/message/6"))
            .andReturn();
    String mvcResult= mockMvc.perform(MockMvcRequestBuilders.get("/messages"))
            .andReturn().getResponse().getContentAsString();
    System.out.println("Result === "+mvcResult);
}
```

测试删除 ID 为 6 的消息体，最后重新查询所有的消息。

运行后返回结果：

```
Result === [{"id":1,"text":"text1","summary":"summary1","created":"2018-07-28T06:43:47.185+0000"},{"id":2,"text":"text2","summary":"summary2","created":"2018-07-28T06:43:47.459+0000"},{"id":3,"text":"text3","summary":"summary3","created":"2018-07-28T06:43:47.461+0000"},{"id":4,"text":"text4","summary":"summary4","created":"2018-07-28T06:43:47.463+0000"},{"id":5,"text":"text5","summary":"summary5","created":"2018-07-28T06:43:47.464+0000"},{"id":7,"text":"text7","summary":"summary7","created":"2018-07-28T06:43:47.468+0000"},{"id":8,"text":"text8","summary":"summary8","created":"2018-07-28T06:43:47.468+0000"},{"id":9,"text":"text9","summary":"summary9","created":"2018-07-28T06:43:47.470+0000"}]
```

运行后发现 ID 为 6 的消息已经被删除。

总结

RESTful 是一种非常优雅的设计，相同 URL 请求方式不同后端处理逻辑不同，利用 RESTful 风格很容易设计出更优雅和直观的 API 交互接口。同时 Spring Boot 对 RESTful 的支持也做了大量的优化，方便在 Spring Boot 体系内使用 RESTful 架构。

> 点击这里下载源码。