GeekBrains

Разработка интернет-магазина на Spring Framework

Веб-инструментарий Spring Framework

Spring Web Services. Spring Websocket.

Оглавление

Spring Web Services

Spring Web Services (Spring-WS) — это библиотека фреймворка Spring, предназначенная для создания веб-сервисов, управляемых документами. Spring Web Services облегчает разработку SOAP веб-сервисов по контракту, позволяя создавать гибкие веб-службы.

При создании веб-сервисов есть два стиля разработки: **Contract-Last** и **Contract-First**.

При использовании подхода **Contract-Last** разработка начинается с Java-кода, и потом генерируется его контракт веб-сервиса (WSDL). WSDL — это Web Services Definition Language (язык описания веб-служб). WSDL-файл — это XML-документ, описывающий веб-службу: он определяет ее местоположение и методы, предоставляемые ею.

Минус подхода Contract-Last, несмотря на его простоту, — сильная зависимость получаемого интерфейса WSDL-сервиса от внутренней реализации сервиса. То есть достаточно легко нарушить контракт, что может привести к неработоспособности клиентов сервиса.

При подходе **Contract-First** разработка начинается с контракта WSDL, и код Java лишь реализует его. Наличие контракта стабилизирует систему, но требует более основательной проработки на этапе проектирования.

Очень многие госуслуги, системы оплаты (Сбербанк, QIWI) предоставляют сервисы по SOAP-протоколу. 1С может быть как SOAP-клиентом, так и сервером — эта особенность часто используется при интеграции с ней. Кроме того, что у SOAP жесткая структура запросов-ответов, он позволяет подписывать сообщения с помощью различных средств (криптопровайдеров) — и это важный аргумент в его пользу. С точки зрения разработчика клиента, плюсом является возможность сгенерировать классы по имеющейся схеме WSDL и обращаться к удаленному WS как к обычному объекту (вызовом методов).

Пример

Классический Hello World: принимаем на вход строку, на выходе — объект бизнес-логики как функцию от запроса. В данном случае — объект приветствия: два поля — текст и дата.

Добавим зависимости к проекту: Spring Boot стартер Spring Web Services и инструмент для манипуляций над WSDL-документами.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web-services</artifactId>
</dependency>
<dependency>
    <groupId>wsdl4j</groupId>
    <artifactId>wsdl4j</artifactId>
</dependency>
```

Также добавляем в pom.xml **Maven-plugin** для генерации классов на основании WSDL-описания:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxb2-maven-plugin</artifactId>
    <version>1.6</version>
    <executions>
            <execution>
                  <id>xjc</id>
                  <goals>
                        <goal>xjc</goal>
                  </goals>
            </execution>
    </executions>

    <configuration>
      <schemaDirectory>
             ${project.basedir}/src/main/resources/
       </schemaDirectory>
      <outputDirectory>
             ${project.basedir}/src/main/java
       </outputDirectory>
      <clearOutputDir>
            false
      </clearOutputDir>
    </configuration>
</plugin>
```

Описываем домен веб-сервиса — файл **resources/greeting.xsd**:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://example.com/api/greeting"
           targetNamespace="http://example.com/api/greeting"
           elementFormDefault="qualified">
    <xs:element name="getGreetingRequest">                                  (2)
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="getGreetingResponse">                                 (3)
        <xs:complexType>
            <xs:sequence>
                <xs:element name="greeting" type="tns:greeting"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="greeting">                                        (1)
        <xs:sequence>
            <xs:element name="text" type="xs:string"/>
            <xs:element name="date" type="xs:date"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Здесь мы описываем:

1. Пользовательский (сложный) тип данных — в данном случае объект, состоящий из строки и даты.
2. Метод и параметры запроса.
3. Формат ответа.

Позднее этот файл отдается работающим веб-сервисом как его собственное описание через URI **/ws/greeting.wsdl**. Также на основе этого файла Maven-plugin **jaxb2** построит классы домена для веб-сервиса.

Генерируем исходный код — выполняем Maven-цель **jaxb2:xjc**. Получаем сгенерированный код в каталоге **./com/example/api/greeting**:

```
.
└──com
    └──example
        └──api
            └──greeting
                    GetGreetingRequest.java
                    GetGreetingResponse.java
                    Greeting.java
                    ObjectFactory.java
                    package-info.java
```

Описываем репозиторий:

```
@Component
public class GreetingRepository {
    public Greeting getGreeting(String name) throws
                                                DatatypeConfigurationException {
        Assert.notNull(name, "name can't be null");
        Greeting greeting = new Greeting();
        greeting.setText(String.format("Hello, %s", name));
        greeting.setDate(DatatypeFactory.newInstance().
                            newXMLGregorianCalendar(new GregorianCalendar()));
        return greeting;
    }
}
```

Задача репозитория — предоставить объект домена веб-сервиса на основе данных запроса, в данном случае — строки **name**.

Endpoint (конечная точка) веб-сервиса:

```
@Endpoint
public class GreetingEndpoint {
    private static final String NAMESPACE_URI =
"http://example.com/api/greeting";

    private GreetingRepository greetingRepository;

    @Autowired
    public GreetingEndpoint(GreetingRepository greetingRepository) {
        this.greetingRepository = greetingRepository;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getGreetingRequest")
    @ResponsePayload
    public GetGreetingResponse getGreeting(@RequestPayload GetGreetingRequest
request) throws DatatypeConfigurationException {
        GetGreetingResponse response = new GetGreetingResponse();
        response.setGreeting(greetingRepository.getGreeting(request.getName()));
        return response;
    }
}
```

Семантический аналог **@Controler** / **@RequestMapping** / **@ResponseBody** / **@RequestBody** из MVC.

- **@Endpoint** регистрирует класс с Spring WS в качестве потенциального кандидата для обработки входящих сообщений SOAP. *// контроллер*
- **@PayloadRoot** используется Spring WS для выбора метода обработчика, основанного на пространстве имен сообщения и **localPart**. *// маппер*
- **@RequestPayload** указывает, что входящее сообщение будет сопоставлено с параметром запроса метода. *// параметры запроса*

- **@ResponsePayload** указывает Spring WS отображать возвращаемое значение в полезную нагрузку ответа. *// полезная нагрузка*

Наконец, конфигурируем наше приложение:

```
@EnableWs                                                                    // (1)
@Configuration
public class WebServiceConfig extends WsConfigurerAdapter {
    @Bean
    public ServletRegistrationBean messageDispatcherServlet(ApplicationContext
applicationContext) {
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        servlet.setApplicationContext(applicationContext);
        servlet.setTransformWsdlLocations(true);
        return new ServletRegistrationBean(servlet, "/ws/*");                // (2)
    }

    @Bean(name = "greeting")                                                 // (3)
    public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema xsdSchema)
{
        DefaultWsdl11Definition wsdl11Definition = new
DefaultWsdl11Definition();
        wsdl11Definition.setPortTypeName("GreetingPort");
        wsdl11Definition.setLocationUri("/ws");
        wsdl11Definition.setTargetNamespace("http://example.com/api/greeting");
        wsdl11Definition.setSchema(xsdSchema);
        return wsdl11Definition;
    }

    @Bean                                                                    // (4)
    public XsdSchema xsdSchema() {
        return new SimpleXsdSchema(new ClassPathResource("greeting.xsd"));
    }
}
```

Здесь мы:

1. «Включаем» Spring Web Services (**@EnableWs**).
2. Регистрируем сервлет и назначаем URI, на котором он будет слушать входящие запросы.
3. Определяем URL-адрес, по которому будут доступны веб-служба и сгенерированный файл WSDL. В этом случае WSDL будет доступен по адресу http://<host>:<port>/ws/greeting.wsdl.
4. Загружаем XML-схему.

Все готово, запускаем Spring Boot-приложение.

Проверяем доступность описания веб-сервиса:

```
GET http://localhost/ws/greeting.wsdl HTTP/1.1
```

В ответ получаем описание:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><wsdl:definitions
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:sch="http://example.com/api/greeting"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://example.com/api/greeting"
targetNamespace="http://example.com/api/greeting">
  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="http://example.com/api/greeting">

    <xs:element name="getGreetingRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
 ....
```

Делаем тестовый запрос к сервису:

```
POST http://localhost/ws HTTP/1.1

Content-Type: text/xml; charset=utf-8

<x:Envelope xmlns:x="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:gre="http://example.com/api/greeting">
    <x:Header/>
    <x:Body>
        <gre:getGreetingRequest>
            <gre:name>John</gre:name>
        </gre:getGreetingRequest>
    </x:Body>
</x:Envelope>
```

Ответ:

```
HTTP/1.1 200

content-type: text/xml;charset=utf-8

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
    <ns2:getGreetingResponse xmlns:ns2="http://example.com/api/greeting">
        <ns2:greeting>
            <ns2:text>Hello, John</ns2:text>
            <ns2:date>2017-10-20+03:00</ns2:date>
        </ns2:greeting>
    </ns2:getGreetingResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Spring Websocket

Протокол WebSocket определяет способ двусторонней связи между клиентом и сервером. Не только клиент инициирует действие (передачу информации), но и сервер наравне с ним может инициировать действие (передачу информации другой стороне).

WebSocket наиболее полно раскрывает свой потенциал в веб-приложениях, где клиенту и серверу необходимо обмениваться событиями очень часто и с минимальной задержкой. Главные кандидаты — биржевые приложения, игры, чаты, приложения для совместной работы.

Важно понимать, что HTTP используется только для первоначального «рукопожатия», которое опирается на механизм, встроенный в протокол, для запроса его обновления — на него сервер может отвечать HTTP-статусом 101 («переключение протоколов»), если согласен. Предполагая, что квитирование выполнено успешно, TCP-соединение, лежащее в основе запроса HTTP, остается открытым, и клиент с сервером могут использовать его для отправки сообщений друг другу.

Spring Framework включает в себя модуль **spring-websocket** с полной поддержкой WebSocket.

WebSocket подразумевает архитектуру обмена сообщениями, но не требует для этого использовать конкретный протокол. Это тонкий слой поверх TCP, который преобразует поток байтов в поток сообщений (текстовый или двоичный).

В отличие от HTTP, который является протоколом уровня приложения, в протоколе WebSocket нет информации для его маршрутизации во входящем сообщении для фреймворка или контейнера. Чистый WebSocket предлагает слишком низкий уровень для комфортной разработки.

Поэтому WebSocket RFC определяет использование подпротоколов. Во время рукопожатия клиент и сервер могут использовать заголовок **Sec-WebSocket-Protocol** для согласования подпротокола. Spring Framework поддерживает STOMP — протокол простого обмена сообщениями, который подходит для использования совместно с WebSocket.

Пример

В данном примере контент (сообщение), отправленное одним из пользователей, появляется на всех клиентах, открывших приложение.

Для работы с WebSocket подключим зависимость:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

Нам потребуется поддержка **WebSocket** на стороне сервера:

```
        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>sockjs-client</artifactId>
            <version>RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>stomp-websocket</artifactId>
            <version>RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>jquery</artifactId>
            <version>RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>webjars-locator</artifactId>
        </dependency>
```

Такая поддержка нужна и на стороне клиента. Кроме того, там же потребуется JQuery для скрипта.

Конфигурируем приложение:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {... }

    @Configuration
    @EnableWebSocketMessageBroker
    public class WebSocketConfig extends
AbstractWebSocketMessageBrokerConfigurer {
        // имя сокета (аналог TCP-сокета)
        @Override
        public void registerStompEndpoints(StompEndpointRegistry registry) {
            registry.addEndpoint("/socket").withSockJS();
        }

        @Override
        public void configureMessageBroker(MessageBrokerRegistry config) {
            // префикс отправителя
            config.enableSimpleBroker("/topic");
            // префикс получателя
            config.setApplicationDestinationPrefixes("/app");
        }
    }
}
```

Здесь мы включаем поддержку WebSocket и регистрируем сокет (аналог TCP-сокета). К нему будет подключаться клиентский код из браузера. Конфигурируем брокер сообщений, указывая префиксы адресов отправителя и получателя.

Определим модель — сущность, которая будет передаваться между клиентом и сервером в реальном времени, без перезагрузки страницы и выполнения ajax-запросов.

```
public class Item {
    private String content;
    // конструкторы, геттеры и сеттеры опущены
 }
```

Контроллер:

```
@Controller
public class MvcController {
    private final List<Item> items = new ArrayList<>();

    @ModelAttribute("items")                    // 1
    public List<Item> getItems() {
        return items;
    }

    @RequestMapping({"/", "/index.html"})       // 2
    public String get() {
        return "index";
    }

    // 3
    @MessageMapping("/item") // вход — канал, куда JS-клиент отправляет
сообщения
    @SendTo("/topic/items") // выход — канал, на который подписывается JS-клиент
    public Item addItem(Item item) throws Exception {
        items.add(item);
        return item;
    }
}
```

По пунктам:

1. Добавляем модель в представление — это потребуется при рендере после перезагрузки страницы.
2. Определяем представление — классический MVC.
3. Определяем конечную точку контроллера обмена STOMP-сообщениями **(не путать с MVC).**

Представление — классический шаблон MVC (используем Thymeleaf):

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
   ...
    <!--STOMP-->
    <script src="/webjars/sockjs-client/sockjs.min.js"></script>
    <script src="/webjars/stomp-websocket/stomp.min.js"></script>

    <!--app.js dependance: jquery-->
    <script src="/webjars/jquery/jquery.min.js"></script>
    <script src="/app.js"></script>
</head>
<body>
<div id="comments" th:if="${!items.empty}">
    <h3>Comments</h3>
    <ul id="list">
        <li th:each="item: ${items}"><span
th:text="${item.content}">comment</span></li>
    </ul>
</div>

<div id="whatsup">
    <form>
        <label for="content">What's up?</label>
        <input type="text" id="content" placeholder="type news here...">
        <button id="send" type="submit">Submit</button>
    </form>
</div>
...
```

файл /templates/**index.html**

В хедере подключаем необходимые библиотеки и клиентский JS-код. Далее выводим список уже имеющихся на сервере сообщений (при перезагрузке страницы). Далее — форма для отправки нового сообщения. Классика.

Клиентский код, общающийся с сервером через WebSocket по протоколу STOMP:

```
var stomp = null;

// подключаемся к серверу по окончании загрузки страницы
window.onload = function() {
    connect();
}

function connect() {
    var socket = new SockJS('/socket');
    stomp = Stomp.over(socket);
    stomp.connect({}, function (frame) {
        console.log('Connected: ' + frame);
        stomp.subscribe('/topic/items', function (item) {
            renderItem(JSON.parse(item.body).content);
        });
    });
}

// хук на интерфейс
$(function () {
    $("form").on('submit', function (e) {
        e.preventDefault();
    });
    $( "#send" ).click(function() { sendContent(); });
});

// отправка сообщения на сервер
function sendContent() {
    stomp.send("/app/item", {}, JSON.stringify({'content':
$("#content").val()}));
}

// рендер сообщения, полученного от сервера
function renderItem(text) {
    $("#list").append("<li style='color: red'>" + text + "</li>");
}
```

файл /static/**app.js**

Здесь мы при загрузке страницы устанавливаем соединение с сервером. Переопределяем действия при отсылке формы, ставим свой обработчик. Определяем функции отправки сообщения и отображения полученного сообщения.

Запускаем. Проверяем. Работает.

Добавим в контроллер конечную точку, принимающую данные по REST и оповещающую WebSocket-клиентов:

```
@RequestMapping(value = "/", method = RequestMethod.PUT)
public ResponseEntity put(@RequestBody String body) throws Exception {
    if (body != null && !body.trim().isEmpty()) {
        Item item = new Item(body);
        items.add(item);
        // оповещаем WebSocket-клиентов
        sendItem(item);
        return new ResponseEntity(HttpStatus.CREATED);
    }
    else {
        return new ResponseEntity(HttpStatus.NOT_ACCEPTABLE);
    }
}

@Autowired
private SimpMessagingTemplate template;

private void sendItem(Item item) {
    this.template.convertAndSend("/topic/items", item);
}
```

При изменении состояния модели ответ (201 CREATED) получает инициирующая сторона. Кроме того, квант изменений отправляется всем клиентским сеансам, подключенным по протоколу WebSocket.

Мы рассмотрели реализацию использования протокола WebSocket в Spring-приложениях.

Практическое задание

1. Используя Spring-WS, реализовать Web Service для выгрузки списка товаров.

Дополнительные материалы

1. Коротко о SOAP.
2. Пример использования (клиент) SOAP WebService в 1С.
3. Тестер SOAP веб-сервисов.
4. Boomerang — SOAP & REST Client (приложение Chrome).
5. Пример использования Spring Websocket.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальная документация.

2. A visual vocabulary for describing information architecture and interaction design.
3. Craig Walls. Spring in Action. — Manning, 2013.
4. STOMP Over WebSocket.
5. RFC6455 — The WebSocket Protocol.