УДК 004

Объединение приложения React с помощью Spring boot

Еровлева Регина Викторовна
Приамурский государственный университет имени Шолом-Алейхема
Студент

Еровлев Павел Андреевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье будут рассмотрены возможности объединения приложения React с помощью Spring boot. Исследование будет проводится в среде разработки IntelijIdea с фрейморков Spring.
Ключевые слова: React, Spring, Spring boot

Bundling React app with Spring boot

Erovleva Regina Viktorovna
Sholom-Aleichem Priamursky State University
Student

Erovlev Pavel Andreevich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article will explore the possibilities of bundling a React application using Spring boot. The research will be carried out in the IntelijIdea development environment with Spring frameworks.
Keywords: React, Spring, Spring boot

В наши дни, когда создается приложение, то часто используется отдельный инструментарий как для внутреннего, так и для внешнего интерфейса. При развертывании этих приложений можно либо выбрать развертывание серверной части и интерфейса отдельно, либо объединить их в один артефакт.

Цель работы – объединить приложение React с использованием Spring Boot.

Исследованиями в данной теме занимались следующие авторы. А.А.Байдыбеков, Р.Г.Гильванов, И.А.Молодкин провели сравнительный анализ всех популярных фреймворков, технологий MVC [1]. Д.А.Викулина, С.Н.Макаров рассмотрели в своей работе передовые средства и технологии

для разработки настольных приложений, провели их анализ и сравнительную характеристику [2]. Т.И.Тимофеев, В.В.Козлов произвели сравнительный обзор фреймворков для настольных приложений по ряду критериев [3]. Е.Д.Зайцев, Д.М.Зайцев раскрыли вопросы эффективности автоматизации тестирования мобильных приложений и web. [4].

Чтобы упростить интеграцию частей приложения, создадим многомодульный проект «Maven». Это позволит создать как интерфейс, так и серверную часть с помощью одного и того же инструмента.

В данном случае используется следующая структура(рис.1).

```
my-project/
  my-project-frontend/
    package.json
    pom.xml
  my-project-backend/
    pom.xml
  pom.xml
```

Рисунок 1 – Структура проекта

Следующим шагом будет определение родительского файла pom.xml (рис.2).

```
<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>be.g00glen00b.apps</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>my-project</name>
    <packaging>pom</packaging>

    <modules>
        <module>my-project-frontend</module>
        <module>my-project-backend</module>
    </modules>
</project>
```

Рисунок 2 – Определение родительского файла pom.xml

Чтобы ресурсы внешнего интерфейса можно было позже скопировать в модуль серверной части, где они будут объединены в один артефакт, необходимо сначала создать front-end.

Для того, чтобы запускать сценарии «npm» с контекстом «Maven», нужно использовать «frontend-maven-plugin». Этот плагин не только выполняет эти сценарии, но также устанавливает определенную версию Node.js и «npm.

Чтобы использовать его, сначала нужно настроить pom.xml (рис.3).

```xml
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>be.g00glen00b.apps</groupId>
        <artifactId>my-project</artifactId>
        <version>1.0.0-SNAPSHOT</version>
    </parent>
    <artifactId>my-project-frontend</artifactId>
    <name>my-project-frontend</name>

    <build>
        <plugins>
            <!-- TODO: Сюда будем добавлять плагины -->
        </plugins>
    </build>
</project>
```

Рисунок 3 – Настройка pom.xml

Следующим шагом будет добавление «frontend-maven-plugin». Следующий фрагмент кода должен появиться именно там, где ранее был размещен комментарий TODO (рис.4).

```xml
<plugin>
    <groupId>com.github.eirslett</groupId>
    <artifactId>frontend-maven-plugin</artifactId>
    <version>1.11.3</version>
</plugin>
```

Рисунок 4 – Добавление плагина

Следующим шагом является настройка того, какие именно шаги должны быть выполнены. Эти шаги сконфигурированы как отдельные исполнения внутри плагина, установка Node.js и «npm» (рис.5).

```xml
<plugin>
    <groupId>com.github.eirslett</groupId>
    <artifactId>frontend-maven-plugin</artifactId>
    <version>1.11.3</version>
    <executions>
        <execution>
            <id>install node and npm</id>
            <goals>
                <goal>install-node-and-npm</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Рисунок 5 – Добавление плагина

Далее устанавливаем зависимости внешнего проекта (рис.6).

```
<execution>
    <id>npm install</id>
    <goals>
        <goal>npm</goal>
    </goals>
    <phase>generate-resources</phase>
</execution>
```

Рисунок 6 – Установка зависимостей плагина

Кроме того, можно определить, на каком этапе должно выполняться это выполнение.

Следующий шаг - также включить выполнение для запуска «npm run build» команды, которая вызывается «react-scripts build». Для этого добавим еще одну зависимость (рис.7).

```
<execution>
    <id>npm run build</id>
    <goals>
        <goal>npm</goal>
    </goals>
    <phase>generate-resources</phase>
    <configuration>
        <arguments>run build</arguments>
    </configuration>
</execution>
```

Рисунок 7 – Добавление зависимости

Как упоминалось ранее, аргумент по умолчанию передается в «npm» команде с использованием «install». Однако, настроив отдельный «<arguments>», можно указать Maven, какую именно команду «npm» следует вызывать. В этом случае используем «run build» аргументы.

Теперь если выполнить команду «mvn package» в модуле внешнего интерфейса, то увидим, что создается папка сборки.

После генерации папки сборки, скопируем эти файлы в загрузочный проект Spring. С помощью Spring загрузки можно легко обслуживать любой статический ресурс, который находится внутри «classpath:static/» или «classpath:public/».

Теперь, откроем «pom.xml» внутреннего проекта и добавим следующий плагин «spring-boot-maven-plugin» (рис.8).

```xml
<plugin>
    <artifactId>maven-resources-plugin</artifactId>
    <version>3.2.0</version>
    <executions>
        <execution>
            <id>copy-resources</id>
            <phase>generate-resources</phase>
            <goals>
                <goal>copy-resources</goal>
            </goals>
            <configuration>
                <outputDirectory>${basedir}/target/classes/static</outputDirectory>
                <resources>
                    <resource>
                        <directory>${basedir}/../my-project-frontend/build</directory>
                        <filtering>false</filtering>
                    </resource>
                </resources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Рисунок 8 – Добавление pom.xml для серверной части

Здесь важно отметить, что фалы не копируются в «src/main/resources/static», как это делается для ресурсов, которые создаются вручную. Вместо этого копируем файлы в папку «target/classes/static».

Далее настроим переадресацию после всех вызовов обратно на начальную страницу. Это необходимо для работы маршрутизации «pushstate» истории при обновлении страницы.

Добавим настраиваемые контроллеры представлений для всех путей, исключая «/api/**» вызовы, поскольку они должны находиться в соответствующем внутреннем контроллере (рис.9).

```java
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController( urlPath: "/").setViewName("forward:/index.html");
        registry.addViewController( urlPath: "/{x:[\\w\\-]+}").setViewName("forward:/index.html");
        registry.addViewController( urlPath: "/{x:^(?!api$).*$}/**/{y:[\\w\\-]+}").setViewName("forward:/index.html");
    }
}
```

Рисунок 9 – Создание WebConfig

Объединение артефактов внешнего и внутреннего интерфейса дает несколько преимуществ. Во-первых, приложение получаете меньше артефактов для развертывания и, следовательно, меньше компонентов, которые нужны для работы.

Кроме того, не будет с проблемами «Cross-Origin Resource Sharing», поскольку и интерфейс, и серверная часть будут обслуживаться из одного источника.

Один из недостатков – это невозможность отдельно масштабировать приложения. Если необходимо будет запустить два экземпляра серверной части, то будет запущено два интерфейса. Однако, поскольку интерфейс — это не что иное, как статические ресурсы, их дублирование не является большой проблемой.

Еще одним недостатком является то, что нельзя изменить ни интерфейс, ни серверную часть, не перестроив их оба. Если обнаружена небольшая опечатка во внешнем интерфейсе, то это значит, что нужно перестроить как интерфейсный, так и внутренний модуль, а также повторно развернуть и перезапустить его.

Библиографический список

1. Байдыбеков А.А., Гильванов Р.Г., Молодкин И.А. Современные фреймворки для разработки web-приложений // Интеллектуальные технологии на транспорте. 2020. №4(24). С. 23-29.
2. Викулина Д.А., Макаров С.Н. Современные технологии создания desktop-приложений // Наука и современность 2012. №18. С. 180-186.
3. Тимофеев Т.И., Козлов В.В. Обзор современных средств создания интерфейса пользователя для desktop приложений // Традиции и инновации в строительстве и архитектуре. строительные технологии. 2017. №1. С.585-588.
4. Зайцев Е.Д., Зайцев Д.М. К вопросу об эффективности автоматизации тестирования web-, desktop- и мобильных приложений // Проблемы инфокоммуникаций 2018. №2(8). С. 56-63.