Сборка war-проекта

JSP & Servlets
2 уровень, 2 лекция

ОТКРЫТА

Отличия war и jar-файлов

Фактически jar-библиотека – это просто zip архив, что напрямую следует из его имени: **Java Archive**. Чаще всего он содержит просто четыре вещи:

- скомпилированные классы;
- ресурсы: properties-файлы и тому подобное;
- манифест MANIFEST.MF;
- другие jar-библиотеки (редко).

Типичная структура такого архива имеет вид:

```
META-INF/
  MANIFEST.MF
com/
  javarush/
        MyApplication.class
application.properties
```

Теперь давай рассмотрим типичный war-файл. Кстати, war не от слова война, а от **Web Archive**. Структура war-файла обычно посложнее. Чаще всего он состоит из двух частей:

- **Java-часть**
  - скомпилированные классы
  - ресурсы для java-классов: properties-файлы и тому подобное
  - другие jar-библиотеки (часто)
  - манифест MANIFEST.MF
- **Web-часть**
  - web-xml – дескриптор развертывания веб-сервиса
  - jsp-сервелеты
  - статические веб-ресурсы: HTML, CSS, JS-файлы

Пример типичного war-файла:

```
META-INF/
    MANIFEST.MF
WEB-INF/
    web.xml
    jsp/
        helloWorld.jsp
    classes/
        static/
        templates/
        application.properties
```

```
    lib/
        // *.jar files as libs
```

Важно! jar-файл может запустить просто java-машина, для запуска же war-файла его нужно загрузить на веб-сервер. Самостоятельно он не запускается.

Плагин создания war-файла с помощью maven-war-plugin

Давайте представим, что у нас есть простой веб-проект. Пусть проект задается такой структурой файлов, как нам его собрать?

```
|-- pom.xml
 `-- src
    `-- main
        |-- java
        |   `-- com
        |       `-- example
        |           `-- projects
        |               `-- SampleAction.java
        |-- resources
        |   `-- images
        |       `-- sampleimage.jpg
        `-- webapp
            |-- WEB-INF
            |   `-- web.xml
            |-- index.jsp
            `-- jsp
                `-- websource.jsp
```

Во-первых, нам нужно указать Maven, собрать все это в виде **war-файла**, для этого есть тег **<package>**, пример:

```
    <project>
    ...
      <groupId>com.example.projects</groupId>
      <artifactId>simple-war</artifactId>
    <packaging>war</packaging>
      <version>1.0-SNAPSHOT</version>
    <name>Simple War Project</name>
      <url>http://javarush.com</url>
    ...
    </project>
```

Во-вторых, нам нужно подключить плагин **maven-war-plugin**. Пример:

```
  <build>
    <plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
        <configuration>
          <webappDirectory>/sample/servlet/container/deploy/directory</webappDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

Тут мы просто задаем плагин, который в будущем можно конфигурировать. Также с помощью тега webappDirectory переопределяем директорию, в которую будет развернут проект. Сейчас расскажу подробнее, о чем идет речь.

Плагину можно задать два режима сборки (два вида goal):

- **war:war**
- **war:exploded**

В первом случает итоговый war-файл просто кладется в папку **target** и имеет имя **<artifactId>-<version>.war**.

Но можно “попросить” плагин, чтобы в итоговую папку содержимое war-файла было помещено в том состоянии, в котором оно будет распаковано веб-сервером у себя внутри. Для этого используется goal **war:exploded**.

Второй подход используется часто, если ты запускаешь или дебажишь проект прямо из Intellij IDEA.

Кстати, тег webappDirectory в примере выше позволяет переопределить директорию куда будет распакован ваш war-файл при сборке в режиме war:exploded.

О других настройках плагина вы можете узнать из его официальной странички.

Сборка web-приложения на основе SpringBoot

Ну и хотелось бы разобрать какой-нибудь реальный пример сборки. Давай не мелочиться и рассмотрим это на примере приложения на основе SpringBoot.

Шаг первый. Создай пустой Maven web-проект с помощью IDEA.

Шаг второй. Добавь в его pom.xml зависимости от Spring.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
```

Шаг третий. Создай класс `com.javarush.spring.MainController`. Его нужно разместить в папке **src/main/java**:

```
@Controller
public class MainController {

    @GetMapping("/")
    public String viewIndexPage(Model model) {
        model.addAttribute("header", "Maven Generate War");
        return "index";
    }
}
```

Тут описаны 3 вещи. Во-первых, аннотация @Controller указывает фреймворку SpringBoot, что этот класс будет использоваться для обслуживания входящих веб-запросов.

Во-вторых, аннотация @GetMapping, указывает, что наш метод будет вызываться для обслуживания GET-запроса на корневой URI - /

В-третьих, метод возвращает строку "index". Это говорит фреймворку SpringBoot, что в качестве ответа нужно отдать содержимое файла index.html.

Шаг четвертый. Нужно добавить в проект файл index.html с таким содержимым:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    <!-- Bootstrap core CSS -->
    <link th:href="@{/css/bootstrap.min.css}" rel="stylesheet">
</head>
<body>
    <nav class="navbar navbar-light bg-light">
        <div class="container-fluid">
            <a class="navbar-brand" href="#">
                JavaRush Tutorial
            </a>
        </div>
    </nav>
    <div class="container">
        <h1>[[${header}]]</h1>
    </div>
</body>
</html>
```

Это не просто html. Перед тем, как его содержимое отдадут клиенту, оно будет модифицировано на сервере фреймворком Thymeleaf. В этот файл встроены специальные теги, которые позволяют библиотеке Thymeleaf обрабатывать и модифицировать содержимое страницы.

Красным отображены теги, которые будут обработаны библиотекой Thymeleaf, зеленым – стили CSS-библиотеки Bootstrap.

Шаг пятый. Задаем плагин в pom.xml:

```
<plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.3.1</version>
</plugin>
```

Немного переоценил свои силы. Чтобы полностью разобрать простой пример, нужно много времени. Но ты можешь скачать полный код проекта из GitHub и попробовать разобраться в нем самостоятельно. Кстати, процентов 80% своего рабочего времени ты будешь делать именно это :)

Полный код ты можешь скачать по ссылке в GitHub.

‹ Предыдущая лекция

Следующая лекция ›

+15

Комментарии (3) популярные новые старые

JavaCoder

Введите текст комментария

Pixta Уровень 90 13 июля, 07:13

Добавление примера создания war считаю неуместным. В целом не завершенная мысль + использование тем, которые сейчас нет цели рассказать.

Ответить +8

Рогов Игорь Уровень 13, Самара, Russian Federation 17 июня, 21:39

вот мы делали Чат, например. как его , например, собрать чтоб он без идеи работал или вообще с томката например? не хватает наглядности

Ответить +2

Фарид Гулиев Уровень 41, Днепр, Украина 14 июля, 22:49

надо было добавить в сервер клиентский интерфейс, а потом создать два jar-файла, один для сервера, другой для чата. Инструкций в интернете полно, сам так делал

Ответить 0

ЯЗЫК ИНТЕРФЕЙСА

Русский

СКАЧИВАЙТЕ НАШИ ПРИЛОЖЕНИЯ

ДОСТУПНО В Google Play

Загрузите в App Store

"Программистами не рождаются" © 2022 JavaRush