JAVA FULL STACK DEVELOPMENT PROGRAM

Session 12: Maven & Deployment

OUTLINE

- Deployment
  - Java compressed file
- Maven
  - Project Object Model(POM)

SCENARIO

- Each of web application contains lots of files — complied java file, external jars, images, css etc.
- How can we move those files to the production server?

JAVA COMPRESSED FILE

- When Java applications are deployed, all of the files that constitute the Java app are compressed and packaged into a single file.
- While compressed files are typically given a .zip extension, the Java community instead uses the .ear extension for Java EE-based enterprise applications, .war for web applications, and .jar for stand-alone Java applications and linkable libraries.
- But, under the covers, EAR, JAR and WAR files are all simply zip files that contain the various images, XML files, property files and pieces of Java code that make up a Java application.
- If the .ear, .war or .jar extension of any of these files is changed to .zip, it can be opened with any standard decompression tool, including 7-Zip or WinRAR.

JAVA COMPRESSED FILE

- Even the simplest Java applications can be composed of 10 or 20 independent Java files.
- Complex enterprise applications can contain thousands of files.
- You can make software development easier with many files with specific responsibilities, but it complicates application deployment.
- But you can simplify this process when you move applications between environments, link to them at runtime, move them across networks and store them in Maven repositories when you package multipart applications as a single, compressed file.
- Despite the differences between JAR, WAR and EAR files, they can all help simplify your Java application deployment time.

JAVA COMPRESSED FILE

- The biggest difference between JAR, WAR and EAR files is the fact that they are targeted toward different environments.
  - An EAR file requires a fully Java Platform, Enterprise Edition (Java EE)-compliant application server, such as WebSphere or JBoss
  - A WAR file only requires a Java EE Web Profile-compliant application server to run
- A JAR(Java Archive) file only requires a Java installation

Differences between WAR and EAR files

	WAR	EAR
Full name	Web Application Archive	Enterprise Application Archive
Supported API	Servlet and JSP API	Java EE API
Contains	Web-based resources such as images, HTML, property files and compiled Java code	Other Java EE archives such as WAR, RAR, EJB-JAR and JAR files
Target runtime	Java Web Profile-compliant application server such as Liberty or Tomcat	Java EE-compliant application server such as JBoss or WebSphere
Deployment descriptor name	web.xml	application.xml

MAVEN

- Building a software project typically consists of such tasks as downloading dependencies, putting additional jars on a classpath, compiling source code into binary code, running tests, packaging compiled code into deployable artifacts such as JAR, WAR, and ZIP files, and deploying these artifacts to an application server or repository.

- Apache Maven automates these tasks, minimizing the risk of humans making errors while building the software manually and separating the work of compiling and packaging our code from that of code construction.

MAVEN FEATURES

- Simple project setup that follows best practices — Maven tries to avoid as much configuration as possible, by supplying project templates (named archetypes)
- Dependency management — it includes automatic updating, downloading and validating the compatibility, as well as reporting the dependency closures (known also as transitive dependencies)
- Isolation between project dependencies and plugins — with Maven, project dependencies are retrieved from the dependency repositories while any plugin's dependencies are retrieved from the plugin repositories, resulting in fewer conflicts when plugins start to download additional dependencies
- Central repository system — project dependencies can be loaded from the local file system or public repositories, such as Maven Central

PROJECT OBJECT MODEL(POM)

- The configuration of a Maven project is done via a *Project Object Model (POM)*, represented by a *pom.xml* file. The *POM* describes the project, manages dependencies, and configures plugins for building the software.

- The *POM* also defines the relationships among modules of multi-module projects

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XM
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <!-- Your own application should inherit from spring-boot-starter-parent -->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.0.2.RELEASE</version>
    </parent>
    <artifactId>SpringBootTest</artifactId>
    <groupId>com.springboot</groupId>
    <name>Spring Boot Web Secure Sample</name>
    <description>Spring Boot Web Secure Sample</description>
    <version>0.0.1-SNAPSHOT</version>
    <url>http://projects.spring.io/spring-boot/</url>
    <organization>
        <name>BeaconFireSolution</name>
        <url>http://www.spring.io</url>
    </organization>
    <properties>
        <main.basedir>${basedir}/../..</main.basedir>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>…
        <dependency>…
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

SCENARIO

- After we deploy our application, we may have bugs reported by production support team. And we may be asked to add additional functions to it.
- What if our code fix is not working properly?
- What if some of our customers don't want the new features we provide in new deployment?

PROJECT IDENTIFIER

- Maven uses a set of identifiers, also called coordinates, to uniquely identify a project and specify how the project artifact should be packaged

  - *groupId* – a unique base name of the company or group that created the project

  - *artifactId* – a unique name of the project

  - *version* – a version of the project

  - *packaging* – a packaging method (e.g. *WAR*/*JAR*/*EAR*)

- eg. *com.beaconfire.sample.1.0-SNAPSHOT*

DEPENDENCIES

- The external libraries that a project uses are called dependencies.
- The dependency management feature in Maven ensures automatic download of those libraries from a central repository, so you don't have to store them locally.
- Uses less storage by significantly reducing the number of downloads off remote repositories
- Makes checking out a project quicker
- Provides an effective platform for exchanging binary artifacts within your organization and beyond without the need for building artifact from source every time

DEPENDENCIES

- In order to declare a dependency on an external library, you need to provide the *groupId, artifactId*, and the *version* of the library

```
<!-- Servlet Spec -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.4</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
</dependency>
```

- As Maven processes the dependencies, it will download Servlet and JSP library into your local Maven repository — .m2 folder

REPOSITORIES

- A repository in Maven is used to hold build artifacts and dependencies of varying types. The default local repository is located in the *.m2/repository* folder under the home directory of the user.

- If an artifact or a plug-in is available in the local repository, Maven uses it. Otherwise, it is downloaded from a central repository and stored in the local repository. The default central repository is Maven Central.

- Some libraries, such as JBoss server, are not available at the central repository but are available at an alternate repository. For those libraries, you need to provide the URL to the alternate repository inside *pom.xml* file

REPOSITORIES

```
<repositories>
    <repository>
        <id>springsource-milestones</id>
        <name>SpringSource Milestones Proxy</name>
        <url>https://oss.sonatype.org/content/repositories/springsource-milestones</url>
    </repository>
</repositories>
```

- Please note that you can use multiple repositories in your projects.
- In most case, in enterprise environment, we will not use Maven Central Repo for security purpose.
    - That is, companies will have their own repositories setup and you have to download all dependencies from there.

PROPERTIES

- Custom properties can help to make your *pom.xml* file easier to read and maintain.
    - In the classic use case, you would use custom properties to define versions for your project's dependencies.
- Maven properties are value-placeholders and are accessible anywhere within a *pom.xml* by using the notation *${name}*, where *name* is the property.

PROPERTIES

```
<properties>
    <spring.version>4.3.5.RELEASE</spring.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>
```

PROFILES

- Another important feature of Maven is its support for *profiles.* A *profile* is basically a set of configuration values.

- By using *profiles*, you can customize the build for different environments such as Production/Test/Development

```
<profiles>
    <profile>
        <id>production</id>
        <build>
            <plugins>
                <plugin>
                //...
                </plugin>
            </plugins>
        </build>
    </profile>
    <profile>
        <id>development</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <build>
            <plugins>
                <plugin>
                //...
                </plugin>
            </plugins>
        </build>
     </profile>
</profiles>
```

PLUGINS AND GOALS

- A Maven *plugin* is a collection of one or more *goals*.
- Goals are executed in phases, which helps to determine the order in which the *goals* are executed.
- There are several common used goals
  - clean — Clean up after the build (Clean up the target folder)
  - install — Install the built artifact into the local repository.
  - test — Run unit test cases

ANY QUESTIONS?