Additional Spring Boot Topics

- Starter Dependencies
- Auto Configuration
- Deploying a WAR

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Dependent Libraries

- Most large-scale Java applications use third-party libraries, which are typically packaged as JAR files
- Many libraries themselves depend on other libraries, leading to **transitive dependencies**
- Build tools such as Gradle and Maven can satisfy transitive dependencies

---

The Apache Ant build tool can also handle dependencies if you extend it with the Ivy tool.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Traditional Approaches for Dependencies

- Libraries typically have multiple versions, with newer versions having additional functionality and/or bug fixes
- Gradle and Maven let you specify specific library version numbers
- One challenge with transitive dependencies is ensuring that the versions "play well together" - traditionally, this job falls to the developer

```
compile("org.springframework:spring-web:4.1.2.RELEASE")
compile("org.springframework.data:spring-data-jpa:1.7.0.RELEASE")
compile("org.hibernate:hibernate-entitymanager:jar:4.2.2.Final")
runtime("com.h2database:h2:1.3.64")
```

The above shows a fragment of a Gradle script that explicitly specific versions of a few dependencies.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Boot Starter Dependencies

- Spring Boot provides **starter dependencies** for both Gradle and Maven that are curated versions of common libraries that work well together
- Instead of specifying individual library versions, you specify a Spring Boot version and Spring Boot provides the dependent versions

```
compile('org.springframework.boot:spring-boot-starter-jdbc')
compile('org.springframework.boot:spring-boot-starter-web')
runtime('com.h2database.h2')
```

- You can see a list of available starters using the CLI's *spring init --list* command

---

The above shows a fragment of a Gradle build script. Note how there's no version numbers! The 'buildscript' portion of the script might look like:

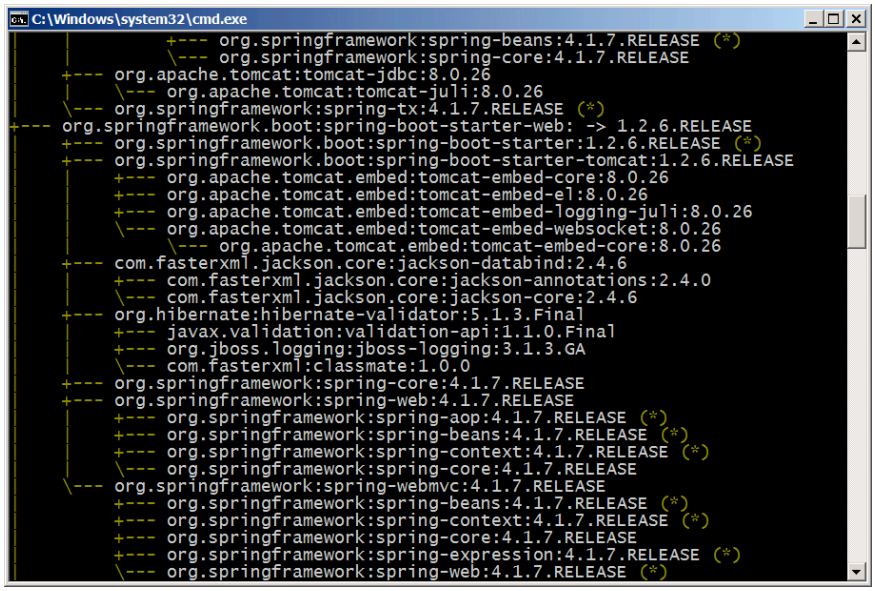
```
buildscript {
    ext {
        springBootVersion = '1.2.6.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersi
        classpath('io.spring.gradle:dependency-management-plugin:0.5.2.RELEASE')
    }
}
```

The key is the dependencies in the buildscript that enable Spring Boot starter dependencies based on the Spring Boot version.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Viewing Dependencies

- Gradle and Maven scripts let you view libraries so you can see what the starter dependencies brought in and the library versions



The above screenshot used this command to generate the output:

```
gradlew dependencies
```

Maven provides the the dependency:tree goal that acts in a similar fashion to the Gradle output shown above.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Eliminating Starter Dependencies

- Starter dependencies bring in commonly used transitive dependencies
- For example, **starter-web** brings in Spring MVC and embedded Tomcat
- You can remove dependencies that your app doesn't need to reduce the size of the executable JAR or WAR

```
compile('org.springframework.boot:spring-boot-starter-web') {
//     exclude group: 'org.hibernate'
       exclude module: 'hibernate-validator'
}
```

The full name of a dependency includes group, module and version, e.g.:

```
org.hibernate:hibernate-validator:5.1.3.Final
```

You can exclude using either the group, which eliminates all modules in that group, or via the module name. Note that you can specify both the group and module name to be more specific.

In Maven, you can use the "exclusions" element to eliminate transitive dependencies.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a Specific Library

- Due to organizational policy, you might wish to use specific versions of some libraries other than what starters provide
- In Gradle, to use a newer version, simply specify it, but to use an older version, first exclude the older version, then specify the newer

```
compile('org.springframework.boot:spring-boot-starter-jdbc') {
  exclude module: 'snakeyaml'
}

compile('org.yaml:snakeyaml:1.8')
```

```
C:\Users\jbarnum\advspring\workspace\lab02>gradlew dependencies | find "snake"
|    |    |    \--- org.yaml:snakeyaml:1.14 -> 1.8
|    |    \--- org.yaml:snakeyaml:1.14 -> 1.8
```


One reason you might need to use specific versions is for security reasons - your organization might vet libraries to ensure they have no vulnerabilities that could affect your application.

Note that if you override the choices made by Spring Boot's starters, the onus is on you to ensure that the versions you choose work well together. One of the big benefits of the starters is that you know that the Spring Boot team has tested the chosen versions together.

In Maven, you just need to specify the specific library version using with a "dependency" element.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Auto Configuration

- In traditional Spring applications, developers needed to provide extensive configuration either using XML, annotations or Java configuration
- Spring Boot eliminates the need for most configuration, freeing the developer to concentrate on application code

```
1    <bean id="studentDataSource"
2      class=
3    "org.springframework.jdbc.datasource.DriverManagerDataSource">
4      <property name="driverClassName"
5        value="org.hsqldb.jdbcDriver"/>
6      <property name="url"
7        value="jdbc:hsqldb:hsql://localhost/"/>
8      <property name="username"
9        value="SA"/>
10     <property name="password"
11       value=""/>
12   </bean>
```


Over the years, Spring has striven to reduce the amount of configuration required using annotations and auto-wiring. Spring Boot continues that trend.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

How Auto Configuration Works

- Essentially, Spring Boot makes configuration decisions based on what it finds on the application's **classpath**
- For example, if you have a dependency for the HSQLDB database, Spring Boot will automatically configure a database connection
- For auto config to work, your application's "main" class should be annotated with **@SpringBootApplication**

The @SpringBootApplication annotation combines @Configuration, @ComponentScan and @EnableAutoConfiguration.

Behind the scenes, auto configuration works based on Spring Boot annotations such as @ConditionalOnMissingBean. For a good explanation on how this works, see: https://geowarin.github.io/understanding-spring-boot.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding Auto Config Externally

- You can override many auto configuration defaults by adding to the **application.properties** file
- For example, you can specify database connection information:

```
spring.datasource.url=jdbc:hsqldb:hsql://localhost/
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
```


Instead of writing simple Java properties files, you can also use a more readable format known as YAML (Yaml Ain't a Markup Language). You can read more about YAML at: http://yaml.org. Note that you need to have the Snake YAML library as a dependency for YAML to work.

You can see a list of common configuration properties at: http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

You can also override auto configuration defaults by writing your own Java configuration classes. For an example of providing database connection info, see: http://docs.spring.io/spring-boot/docs/current/reference/html/howto-data-access.html.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Executable JAR Deployment

- By default, *Initializr* generates build scripts that create a "fat" executable JAR that you can run to bootstrap the application
- If it's a Web application, executing the JAR launches an embedded Tomcat
- You can configure many server and Tomcat options via properties files

```
server.port=80

# enable SSL
server.port=443
server.ssl.key-store=classpath:mykeystore.jks
server.ssl.key-store-password=duke
server.ssl.key-password=mypassword
```

Executable JAR deployment is a good choice if you plan to run the application with a 'cloud' provider such as Heroku, or Docker.

You can also configure Spring Boot to use Jetty instead of Tomcat.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Building a WAR

- Executable JARs are easy, but some organizations use a traditional **application server** such as WebSphere or WebLogic
- Gradle scripts can use the **war** plugin to generate a WAR that you can deploy to an app server

```
apply plugin: 'war'

war {
    baseName = 'myapp'
    version = '0.0.1-SNAPSHOT'
}
```

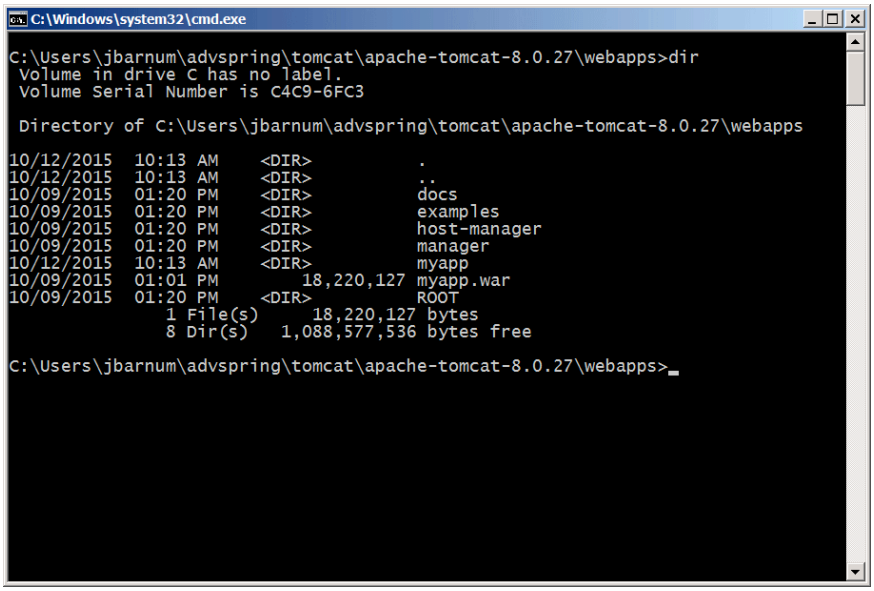

Maven scripts can change the "packaging" element from jar to war.

If you need to deploy an EAR instead of a WAR, it's a simple matter to package the WAR and an application.xml deployment descriptor into an EAR.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Deploying the WAR

- Deploying WARs is application-server specific - for Tomcat, just drop the WAR into TOMCAT_HOME/webapps



Note that depending on your application, you might need to do more work. For example, if you're using Spring MVC, Spring Boot executable JARs automatically enable the Spring 'dispatcher servlet', but if you're deploying a WAR, you will need to configure the servlet yourself, perhaps by writing entries in the web app's web.xml deployment descriptor.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About Spring Boot starter dependencies and how to override them
- About Spring Boot auto configuration and how to override it
- How to deploy Spring Boot applications as executable JARs or WARs

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum