Pivotal

Leveraging Spring Boot Starters and Auto-configuration

Discovering how starters and auto-configuration simplifies Spring application development

Objectives

After completing this lesson, you should be able to

- Utilize Spring Boot Starters to configure a project's dependencies
- Utilize auto-configuration to simplify project configuration and initialization
- Describe the behavior of various configuration elements, such as *@SpringBootApplication*
- Override default configuration

Four Key Boot Features Used in Last Lab

- Starters
  - Configure dependencies quickly
- Auto-configuration
  - Opinionated defaults enable rapid bootstrapping of project
- Configuration properties
  - One mechanism for tailoring configuration
- **`CommandLineRunner & ApplicationRunner`**
  - Easy way to invoke logic after ApplicationContext is loaded

We will explore these features in-depth in this section

Agenda

- **Starters and BOMs**
- Auto-Configuration
- Configuration Properties
- Overriding Configuration
- Running an Application
- Bonus
  - Advanced Properties
  - Fine Tuning Logging
  - YAML for Configuration

Spring Boot Needs Dependencies

- Spring Boot relies on analyzing the classpath
  - If you forget a dependency, Spring Boot can't configure it
  - A dependency management tool is recommended
  - Spring Boot parent and starters make it much easier
- Spring Boot works with Maven, Gradle, Ant/Ivy
  - *Our content here will only show Maven*

Spring Boot Parent POM

- Defines key versions of dependencies and Maven plugins
  - Uses a **`dependencyManagement`** section internally

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.3.RELEASE</version>
</parent>
```

Defines properties for dependencies, for example:
`${spring.version} = 5.0.7.RELEASE`

Spring Boot *"Starter"* Dependencies

- Easy way to bring in multiple coordinated dependencies
  - Including *"Transitive"* Dependencies

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

Version not needed!
Defined by parent

Resolves ~ 16 JARs!

spring-boot-.jar*
spring-core-.jar*
spring-context-.jar*
spring-aop-.jar*
spring-beans-.jar*
aopalliance-.jar*
...

Test *"Starter"* Dependencies

- Common test libraries

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

Resolves

spring-test-.jar*
junit-.jar*
mockito-.jar*
…

Available Starter POMs

- Not essential but *strongly* recommended
- Coordinated dependencies for common Java enterprise frameworks
  - Pick the starters you need in your project
- To name a few:
  - **`spring-boot-starter-jdbc`**
  - **`spring-boot-starter-data-jpa`**
  - **`spring-boot-starter-web`**
  - **`spring-boot-starter-batch`**

See: Spring Boot Reference, Starter POMs
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter

Spring Boot Developer Tools

- A set of tools to help make Spring Boot development easier
  - Automatic restart - any time a class file changes (on re-compile)
  - Additional features supporting remote application execution from IDE, global devtool settings
- Note the pattern for artifactId is different

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
  </dependency>
</dependencies>
```

Agenda

- Starters and BOMs
- **Auto-Configuration**
- Configuration Properties
- Overriding Configuration
- Running an Application
- Bonus
  - Advanced Properties
  - Fine Tuning Logging
  - YAML for Configuration

Spring Boot @EnableAutoConfiguration

- `@EnableAutoConfiguration` annotation on a Spring Java configuration class
  - Spring Boot automatically creates the beans it thinks you need
  - Usually based on classpath contents, can be easily overridden

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class Application {
    public static void main(String[] args) {
            SpringApplication.run(Application.class, args);
    }
}
```

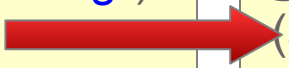
No packages specified = scan current package and all sub-packages

`SpringApplication` is actually a Spring Boot class

Shortcut: @SpringBootApplication

- Very common to use `@EnableAutoConfiguration`, `@Configuration`, and `@ComponentScan` together

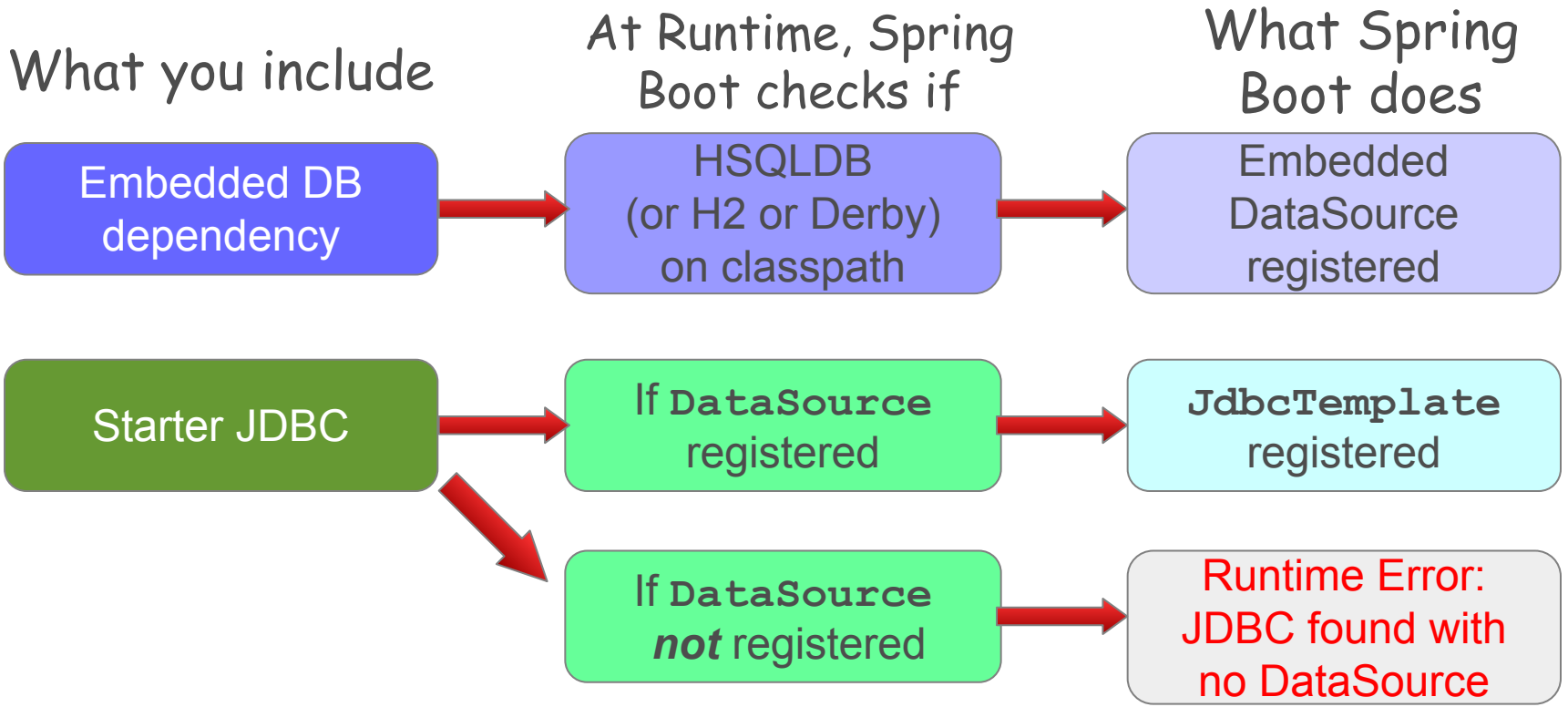
```
@SpringBootConfiguration
@ComponentScan("example.config")
@EnableAutoConfiguration
public class Application {
   ...
}
```



```
@SpringBootApplication
(scanBasePackages="example.config")
public class Application {
   ...
}
```

@SpringBootConfiguration simply extends **@Configuration** – see **@SpringBootTest** for why.

Auto-Configuration: Examples



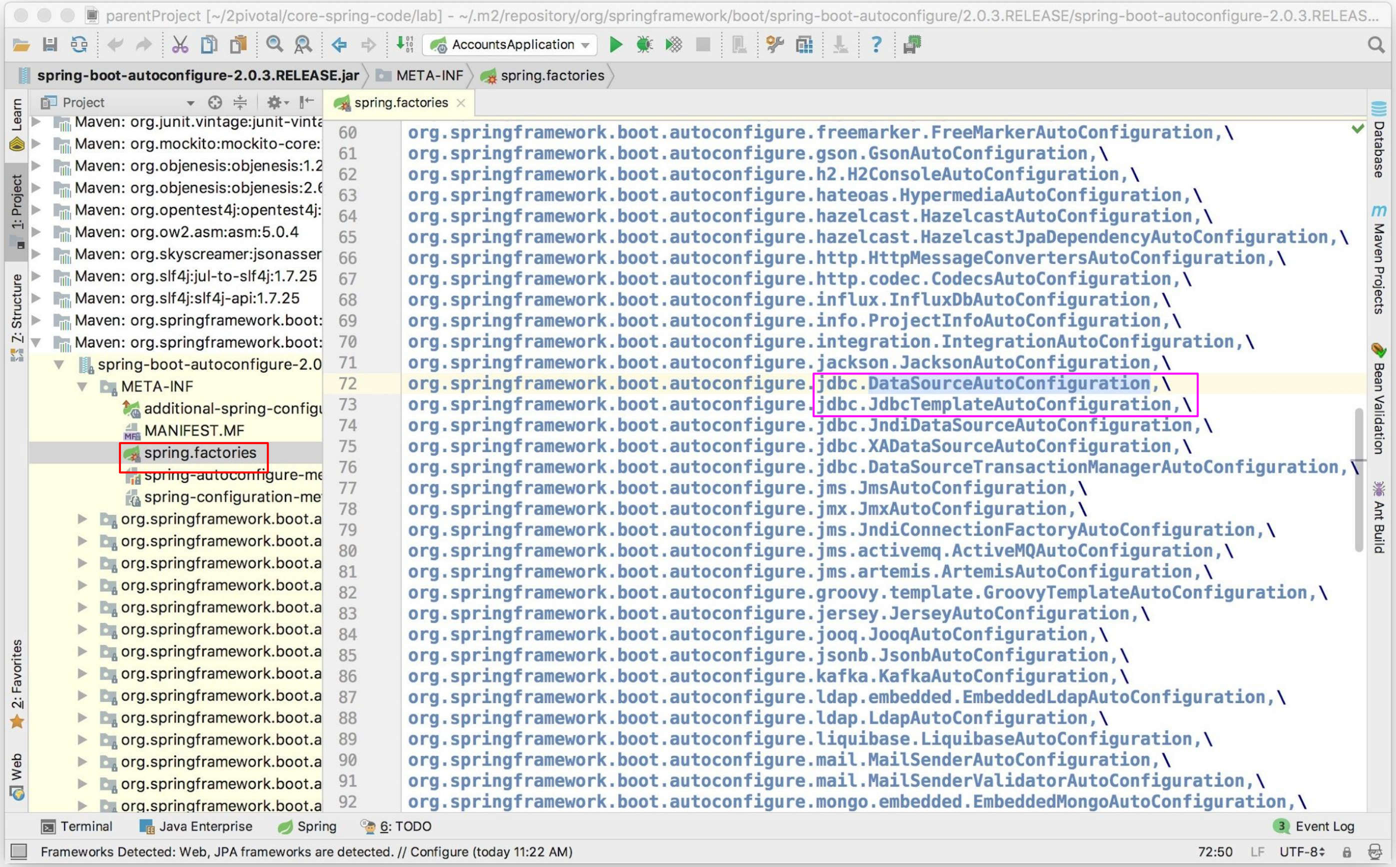
Auto-Configuration Factories

- ***@EnableAutoConfiguration*** reads the ***spring-boot-autoconfigure/META-INF/spring.factories***

- The ***spring.factories*** file contains a list of auto-configuration classes (****AutoConfiguration***) that have all the logic to be executed accordingly to the dependencies that an application has in the classpath

- Auto-configuration classes in the ***spring.factories*** file get processed after application defined configuration classes are processed

Exploring Auto-configuration classes in *spring.factories*



How Does Auto-Configuration Work?

- Extensive use of *pre-written* `@Configuration` classes
- Configuration of beans based on on
  - The contents of the classpath
  - Properties you have set
  - Beans already defined (or not defined)

- `@Profile` is an example of conditional configuration
  - Spring Boot takes this idea to the next level

@Conditional Annotations

- Allow conditional bean creation
  - Only create if other beans exist (or don't exist)

```
@Bean
@ConditionalOnBean(name={"dataSource"})
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
}
```

  - Or by type: @ConditionalOnBean(DataSource.class)
- Many others:
  - @ConditionalOnClass, @ConditionalOnProperty, ...
    @ConditionalOnMissingBean, @ConditionalOnMissingClass

Leverages @Conditional added in Spring 4.0

What are Auto-Configuration Classes?

- Pre-written Spring configurations
  - **`org.springframework.boot.autoconfigure`** package
  - See **`spring-boot-autoconfigure`** JAR file
    - Best place to check what they exactly do

```
@Configuration
public class DataSourceAutoConfiguration {
    …
    @Conditional(...)
    @ConditionalOnMissingBean(DataSource.class, ..)
    @Import({EmbeddedDataSourceConfiguration.class})
    protected static class EmbeddedDatabaseConfiguration { ... }
    …
}
```

Spring Boot defines many of these configurations. They activate in response to dependencies on the classpath

Agenda

- Starters and BOMs
- Auto-Configuration
- **Configuration Properties**
- Overriding Configuration
- Running an Application
- Bonus
  - Advanced Properties
  - Fine Tuning Logging
  - YAML for Configuration

Configuration Properties
Using *application.properties (or application.yml) file*

- Developers commonly externalize properties to files
  - Easily consumable via **`@PropertySource`**
  - But developers name / locate their files different ways

- Spring Boot looks for **`application.properties`**
  - *Many* properties exist to control auto-configuration
  - Can put *any* properties you need in here
    - Boot will automatically find and load them
  - Available to **`Environment`** and **`@Value`** in usual way

See *Appendix A* of Spring Boot documentation:
http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

Example: External Database

- Configuring an *external* database
  - Such as MySQL
  - Make sure project defines JDBC driver dependency

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.schema=/testdb/schema.sql
spring.datasource.data=/testdb/data.sql
```

Example: Controlling Logging Level

- Boot can control the logging level
  - Just set it in **`application.properties`**
- Works with most logging frameworks
  - Java Util Logging, Logback, Log4J, Log4J2

```
logging.level.org.springframework=DEBUG
logging.level.com.acme.your.code=INFO
```

application.properties

Try to stick to SLF4J in the application.
The *advanced* section covers how to change the logging framework

Where to Define Properties

- Spring Boot properties
  - Use **`application.properties`**
    - Read early by Spring Boot since they affect auto-configuration
    - Some properties can *only* be set there
      - Such as logging levels

- Your application properties
  - *Can* be in **`application.properties`**
    - But add comments to identify Boot vs custom properties
  - *Or* use your own properties files with **`@PropertySource`**

Agenda

- Starters and BOMs
- Auto-Configuration
- Configuration Properties
- **Overriding Configuration**
- Running an Application
- Bonus
  - Advanced Properties
  - Fine Tuning Logging
  - YAML for Configuration

Controlling What Spring Boot Does

- There are several options
  - Set some of Spring Boot's properties
  - Define certain beans yourself so Spring Boot won't
  - Explicitly disable some auto-configuration
  - Change dependencies

1. Set some of Spring Boot's properties

- Spring Boot looks for **`application.properties`** in these locations (in this order):
  - **`/config`** sub-directory of the working directory
  - The working directory
  - **`config`** package in the classpath
  - classpath root
- Creates a *PropertySource* based on these files
- Many, many configuration properties available

See *Appendix A* of Spring Boot documentation:
http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

2. Define certain beans yourself

- Normally beans you declare *explicitly* disable any auto-created ones.
  - *Example:* Your own **`DataSource`** stops Spring Boot creating a default **`DataSource`**
  - Bean name typically not relevant
  - Works with Java Config, Component Scanning

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder().
                                setName("RewardsDb").build();
}
```

3. Explicitly disable some auto-configuration

- Can disable some auto-configuration classes
  - If they don't suit your needs
- Via an annotation

```
@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)
public class ApplicationConfiguration {
    …
}
```

`@SpringBootApplication` also has the *exclude* attribute

- Or use *configuration*

application.properties

```
spring.autoconfigure.exclude=\
    org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

4a. Override Dependency Versions

- Spring Boot POMs preselect the versions of dependencies
  - Ensures the versions of all dependencies are consistent
  - Simplifies dependency management in many cases
- Should I override the version of a given dependency?
  - Use default pre-selected version unless there are compelling reasons such as
    - A bug in the given version
    - Compliance
    - Company policies/restrictions

4b. Override Dependency Versions

- Set the appropriate Maven property in your **`pom.xml`**

```
<properties>
    <spring.version>5.0.0.RELEASE</spring.version>
</properties>
```

- Check this POM to know all the properties names
  - https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-dependencies/pom.xml

This only works if you *inherit* from the parent. You need to redefine the artifact if you directly import the dependency

4c. Explicitly Substitute Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Excludes Tomcat

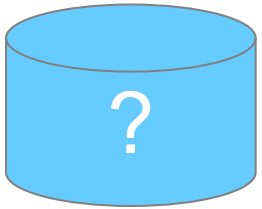
Adds Jetty

Jetty automatically detected and used!

Configuration Example: DataSource (1)

- A common example of how to control or override Spring Boot's default configuration

- Typical customizations
  - Use the predefined properties
  - Change the underlying data source connection pool implementation
  - Define your own DataSource bean (shown earlier)



Example: DataSource Configuration (2)

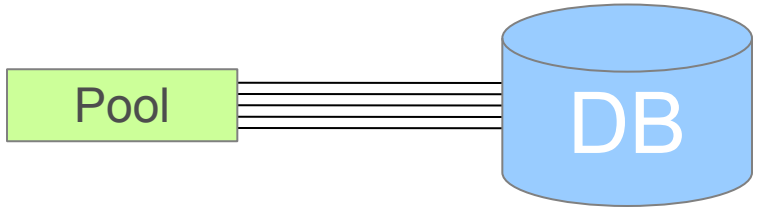
- Common properties configurable from properties file

```
spring.datasource.url=              # Connection settings
spring.datasource.username=
spring.datasource.password=
spring.datasource.driver-class-name=

spring.datasource.schema=           # SQL scripts to execute
spring.datasource.data=

spring.datasource.initial-size=     # Connection pool settings
spring.datasource.max-active=
spring.datasource.max-idle=
spring.datasource.min-idle=
```

Example: DataSource Configuration (3)

- Spring Boot creates a pooled **`DataSource`** by default
  - If a known pool dependency is available
    - *spring-boot-starter-jdbc* or *spring-boot-starter-jpa* starters try to pull in *a* connection pool by default
  - *Choices:* Tomcat, HikariCP, Commons DBCP 1 & 2
    - Set **`spring.datasource.type`** to pick a pool explicitly

Default pool:
- Spring Boot 1.x: Tomcat
- Spring Boot 2.x: Hikari



Agenda

- Starters and BOMs
- Auto-Configuration
- Configuration Properties
- Overriding Configuration
- **Running an Application**
- Bonus
  - Advanced Properties
  - Fine Tuning Logging
  - YAML for Configuration

`CommandLineRunner` and `ApplicationRunner`

- Offers a Spring-style entry point for running applications
  - Avoids having logic in **`main()`** method
- **`CommandLineRunner`**
  - Offers **`run()`** method, handling arguments as an array
- **`ApplicationRunner`**
  - Offers **`run()`** method, handling arguments as **`ApplicationArguments`**
  - A more sophisticated argument handling mechanism

Using `CommandLineRunner`

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    CommandLineRunner commandLineRunner(JdbcTemplate jdbcTemplate){
        String QUERY = "SELECT count(*) FROM T_ACCOUNT";

        return args -> System.out.println("Hello, there are "
                + jdbcTemplate.queryForObject(QUERY, Long.class)
                + " accounts");

    }
}
```

Special Spring Bean detected by Boot and invoked before returning from SpringApplication.run()

Fat JARs and the Spring Boot Plugin

- A “fat” JAR is a JAR that also contains all its dependencies
  - Can be run directly using **`java -jar`** command

- To create
  - Add plugin to your Maven POM or Gradle Build file
  - Build JAR in usual way
    - **`gradle assemble`** or **`mvn package`**
  - Creates two JARs
    - **`my-app.jar`** the executable “fat” jar
    - **`my-app.jar.original`** the “usual” jar

Spring Boot Plugin - Maven

- What it does
  - Extend **`package`** goal to create fat JAR
  - Add **`spring-boot:run`** goal to run your application

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Spring Boot Plugin - Gradle

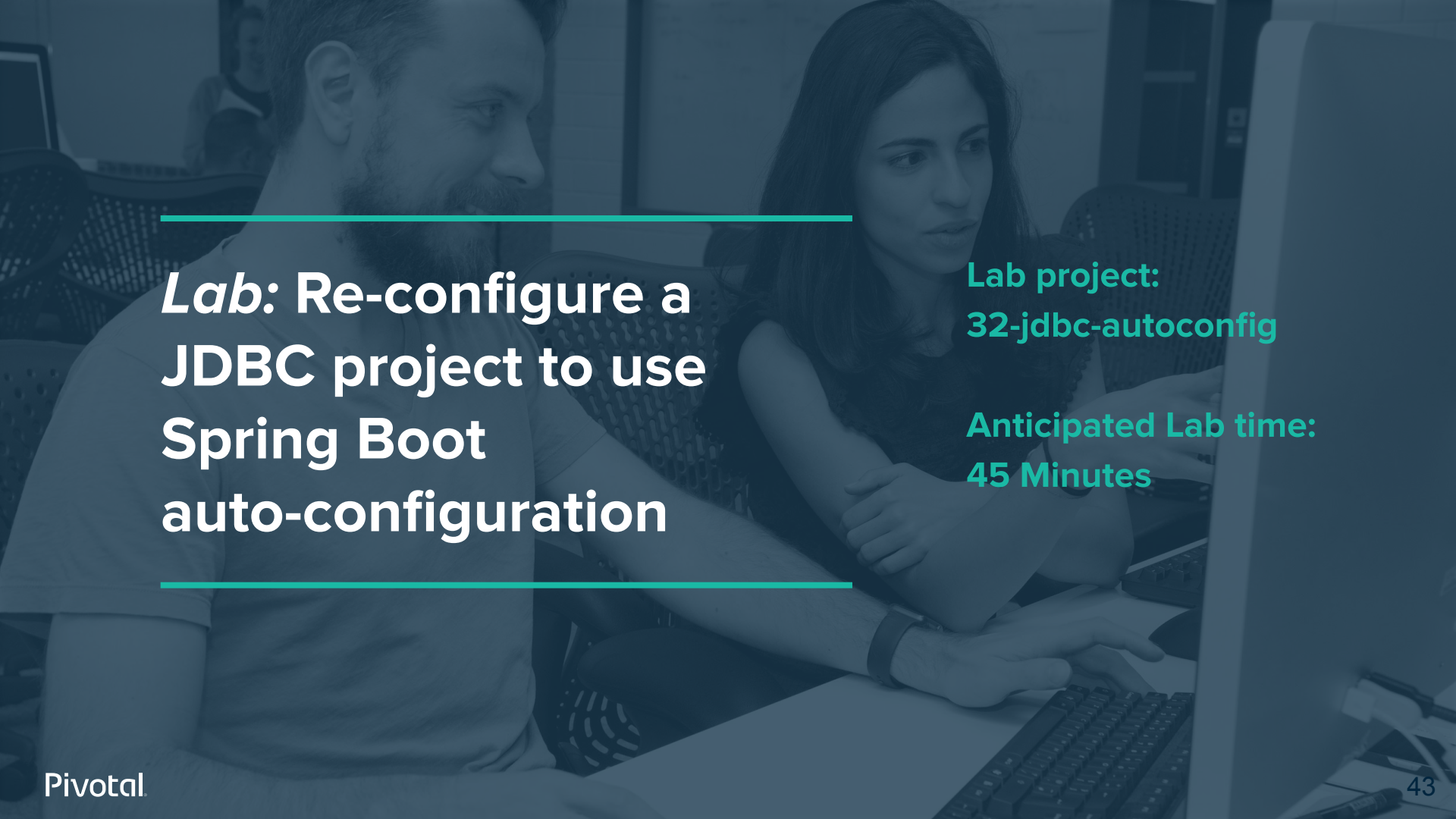
- What it does
  - Enable dependency management like Maven
    - Gradle does *not* provide this feature as standard
  - Extend the **`assemble`** task to create fat JAR
  - Add **`bootRun`** goal to run your application

```
plugins {
    id 'org.springframework.boot' version '2.0.3.RELEASE'
}

apply plugin: 'java'
apply plugin: 'io.spring.dependency-management'
```

Summary

We've discussed starters and auto-configuration:

- How do we get a predefined set of dependencies?
- What is auto-configuration?
- What are configuration properties? Why use them?
- How do you override Spring Boot's defaults
- What is the purpose of a `CommandLineRunner`?
- What is a "fat" jar?



Lab: Re-configure a JDBC project to use Spring Boot auto-configuration

Lab project:
32-jdbc-autoconfig

Anticipated Lab time:
45 Minutes

Agenda

- Starters and BOMs
- Auto-Configuration
- Configuration Properties
- Running an Application
- **Bonus**
  - **Advanced Properties**
  - Fine Tuning Logging
  - YAML for Configuration

Overriding Properties

Applies to Spring *or* Spring Boot

- Order of evaluation of the properties (non-exhaustive)
  - Command line arguments
  - Java system properties
  - OS environment variables
  - Property file(s) – including **`application.properties`**
- Can access any of them using **`@Value`** in the usual way
- *Recommendation:*
  - Use Property files to define defaults
  - Override *externally* using one of the other 3 options

The Problem with Property Placeholders

- Using property placeholders is sometimes cumbersome
  - Many properties, prefix has to be repeated

```
@Configuration
public class RewardsClientConfiguration {

    @Value("${rewards.client.host}") String host;
    @Value("${rewards.client.port}") int port;
    @Value("${rewards.client.logdir}") String logdir;
    @Value("${rewards.client.timeout}") int timeout;

    ...
}
```

Use @ConfigurationProperties

- **`@ConfigurationProperties`** on *dedicated* container bean
  - Will hold the externalized properties
  - Avoids repeating the prefix
  - Data-members automatically set from corresponding properties

```
@ConfigurationProperties(prefix="rewards.client")
public class ConnectionSettings {

    private String host;
    private int port;
    private String logdir;
    private int timeout;
    …    // getters/setters
}
```

```
rewards.client.host=192.168.1.42
rewards.client.port=8080
rewards.client.logdir=/logs
rewards.client.timeout=2000
```

example.properties

Use @EnableConfigurationProperties

- *@EnableConfigurationProperties* on configuration class
  - Specify and auto-inject the container bean

```
@Configuration
@EnableConfigurationProperties(ConnectionSettings.class)
public class RewardsClientConfiguration {
    // Spring initialized this automatically
    @Autowired ConnectionSettings connectionSettings;

    @Bean public RewardClient rewardClient() {
        return new RewardClient(
            connectionSettings.getHost(),
            connectionSettings.getPort(), ...
        );
    }
}
```

Relaxed Property Binding

- Flexible mapping between Java-style properties and environment variables
  - **`path`** equivalent to **`PATH`**
  - **`java.home`** equivalent to **`JAVA_HOME`**
  - Easy overriding of property without changing the name!
- Feature implemented by **`@ConfigurationProperties`**
  - Works for Spring Boot properties

```
@Configuration
class AppConfig {

   @Value("${java.home}")
   String javaInstallDir;
   ...
}
```

Agenda

- Starters and BOMs
- Auto-Configuration
- Configuration Properties
- Running an Application
- **Bonus**
  - Advanced Properties
  - **Fine Tuning Logging**
  - YAML for Configuration

Logging frameworks

- Spring Boot includes by default
  - SLF4J: logging facade
  - Logback: SLF4J implementation
- Best practice: stick to this in your application
  - Use the SLF4J abstraction the application code
- Other logging frameworks are supported
  - Java Util Logging, Log4J, Log4J2

Substituting Logging Libraries

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
    <exclusions>
        <exclusion>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

Excludes Logback

Includes Log4J

Logging Output

- Spring Boot logs by default to the console
- Can also log to rotating files
  - Specify file OR path in application.properties

```
# Use only one of the following properties

#  absolute or relative file to the current directory
logging.file=rewards.log

#  will write to a spring.log file
logging.path=/var/log/rewards
```

Spring Boot can also configure logging by using the appropriate configuration file of the underlying logging framework.

Agenda

- Starters and BOMs
- Auto-Configuration
- Configuration Properties
- Running an Application
- **Bonus**
  - Advanced Properties
  - Fine Tuning Logging
  - **YAML for Configuration**

What is YAML?

- *Yaml Ain't a Markup Language*
  - Recursive acronym
- Created in 2001
- Alternative to .properties files
  - Allows hierarchical configuration
- Java parser for YAML is called SnakeYAML
  - Must be in the classpath
  - Provided by spring-boot-starters

YAML for Properties

- Spring Boot support YAML for Properties
  - An alternative to properties files

```
database.host = localhost
database.user = admin
```

application.properties

equals

```
database:
  host: localhost
  user: admin
```

application.yml

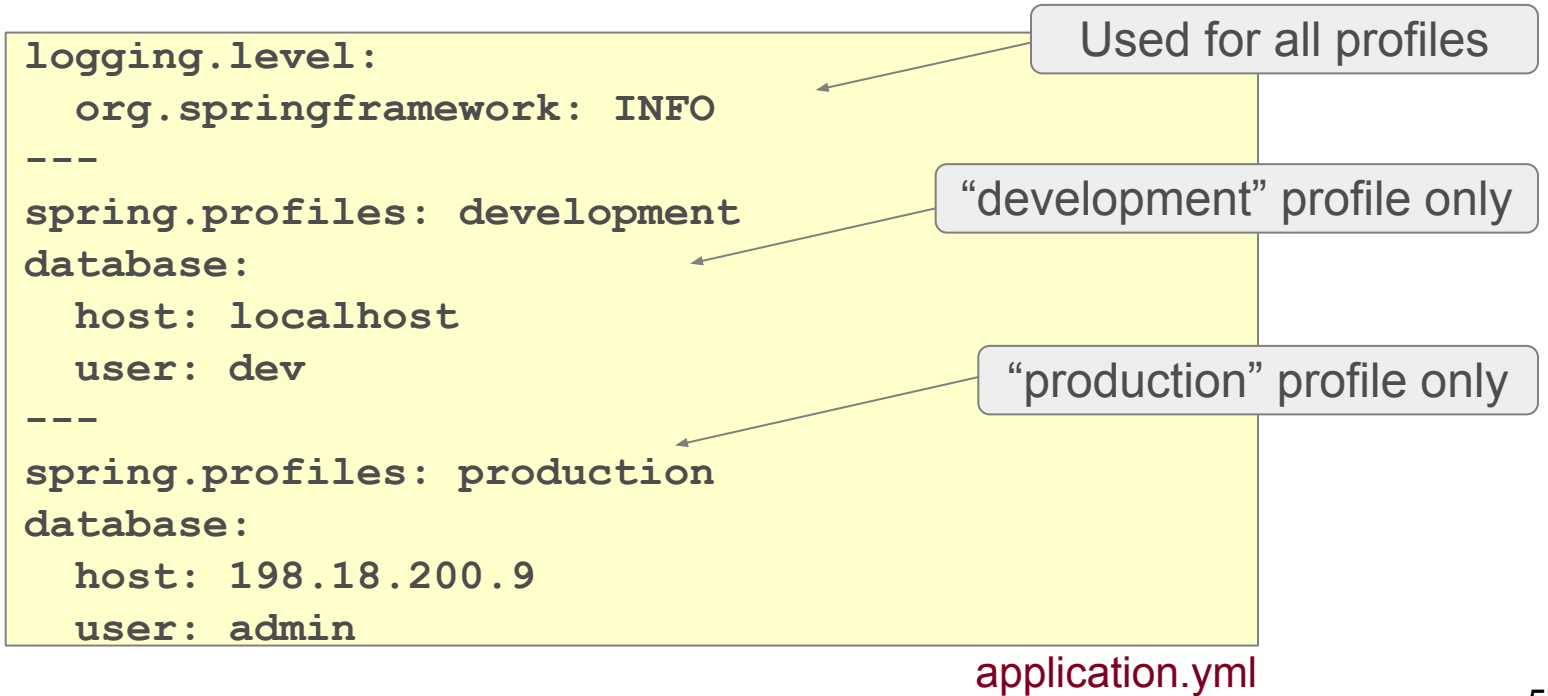
colon

- YAML is convenient for hierarchical configuration data
  - Spring Boot properties are organized in groups

Spring Boot *only* loads `application.yml` by default.

Multiple Profiles Inside a Single YAML File

- YAML file can contain configuration for multiple profiles
  - '---' implies a separation between profiles



Multiple Profiles – YML vs Properties

```
server:
  port: 9999
---
spring.profiles: development
database:
  host: localhost
  user: dev
---
spring.profiles: production
database:
  host: 198.18.200.9
  user: admin
```

application.yml

```
server.port=9999
```

application.properties

```
database.host=localhost:
database.user=dev
```

application-development.properties

```
database.host=198.18.200.9
database.user=admin
```

application-production.properties