Pivotal

Spring Boot – A Closer Look

Discovering how properties and auto-configuration simplifies Spring application development

Objectives

After completing this lesson, you should be able to

- Understand options for defining properties
- Utilize auto-configuration to simplify project configuration and initialization
- Override default configuration

Agenda

- **Properties**
- @ConfigurationProperties
- Auto-Configuration
- Overriding Configuration
- Running an Application
- Optional Topics

Pivotal

Externalized Properties: application.properties

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

- Spring Boot looks for **`application.properties`**
  - Available to **`Environment`** and **`@Value`** in usual way
  - Place *any* properties you need here
    - Boot will automatically find and load them

Location of application.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

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

application.properties

- Creates a *PropertySource* based on these files

Profiles and application.properties

- Spring Boot will look for profile-specific properties in files following *application-{profile}.properties* convention
  - Examples:
    - application-local.properties - local dev properties
    - application-cloud.properties - cloud properties
    - application.properties - always loaded

```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://localhost/transfer
db.user=transfer-app
db.password=secret45
```

application-local.properties

```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://prod/transfer
db.user=transfer-app
db.password=secret99
```

application-cloud.properties

Pivotal

YAML Support

- Spring Boot also supports YAML configuration
  - More concise, indented text format (similar to JSON)
- By default it looks for ***application.yml*** (same locations)
  - Indent must be 2 spaces
  - *Do not use tabs*

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

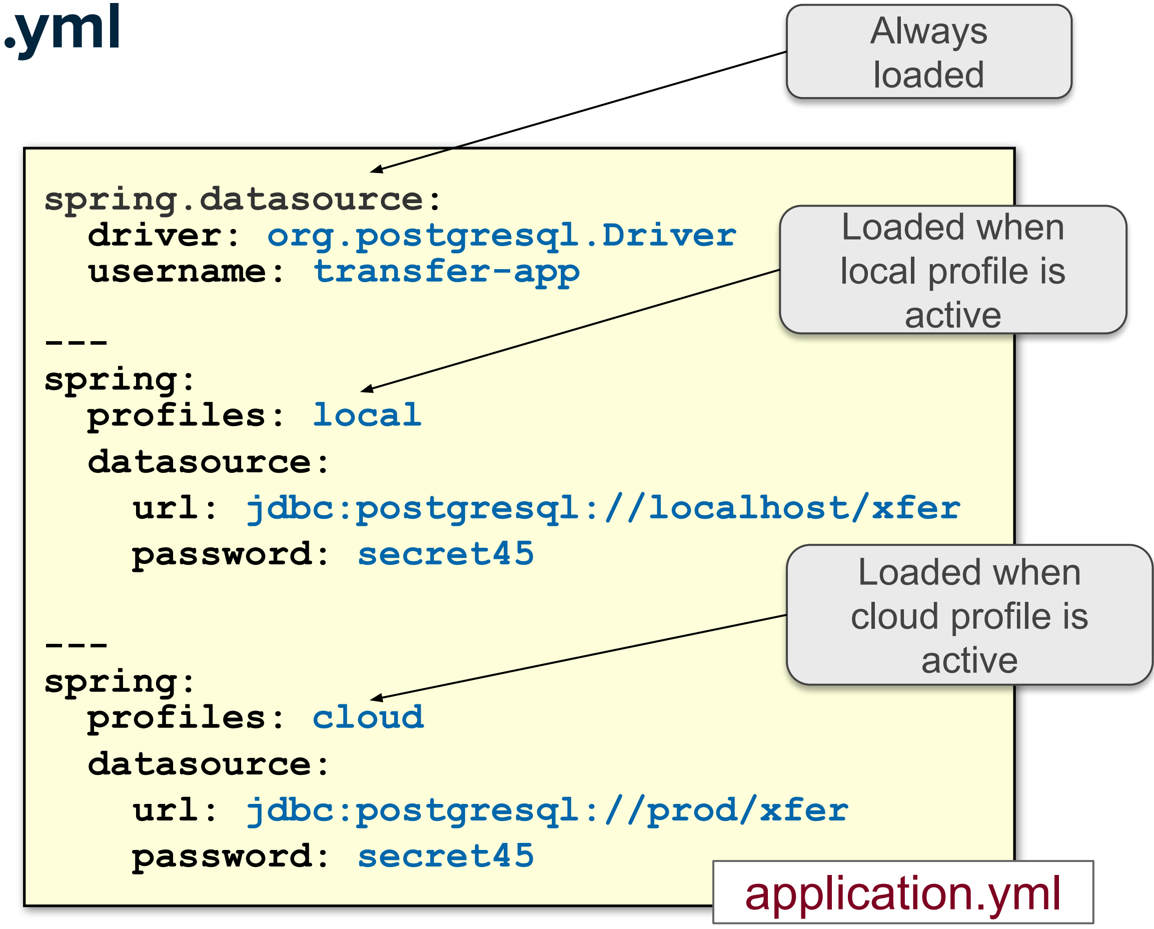
application.yml

Requires snakeyaml.jar, provided by spring-boot-starter.
Note: Spring framework and @PropertySource do not support YAML config files.

Profiles and application.yml

- Profile-specific properties can be placed in a single file.

  "---" indicates separate logical file



```
spring.datasource:
  driver: org.postgresql.Driver
  username: transfer-app

---
spring:
  profiles: local
  datasource:
    url: jdbc:postgresql://localhost/xfer
    password: secret45

---
spring:
  profiles: cloud
  datasource:
    url: jdbc:postgresql://prod/xfer
    password: secret45
```

Always loaded
Loaded when local profile is active
Loaded when cloud profile is active
application.yml

Precedence

- Spring Boot selects properties in the following order (simplified):

1. Devtools settings
2. @TestPropertySource and @SpringBootTest properties
3. Command line arguments
4. SPRING_APPLICATION_JSON (inline JSON properties).
5. ServletConfig / ServletContext parameters.
6. JNDI attributes from java:comp/env
7. Java System properties
8. OS environment variables
9. Profile-specific application properties
10. Application properties / YAML
11. @PropertySource files
12. SpringApplication.setDefaultProperties.

Pivotal

Agenda

- Properties
- **@ConfigurationProperties**
- Auto-Configuration
- Overriding Configuration
- Running an Application
- Optional Topics

Pivotal

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* 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
```

application.properties

Enable @ConfigurationProperties (3 Schemes)

- *@EnableConfigurationProperties* on configuration class
- *@ConfigurationPropertiesScan* on configuration class (Spring Boot 2.2.1+)
- *@Component* on configuration properties class

```
@Configuration
@EnableConfigurationProperties(ConnectionSettings.class)
public class RewardsConfiguration { .. }
```

```
@Configuration
@ConfigurationPropertiesScan
public class RewardsConfiguration { .. }
```

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

Relaxed Binding on @ConfigurationProperties

- `@ConfigurationProperties` beans utilize relaxed binding.

```
@ConfigurationProperties("rewards.client-connection")
public class ConnectionSettings {
    private String hostUrl;
    …
}
```

- Valid Matches:

```
rewards.clientConnection.hostUrl
rewards.client-connection.host-url
rewards.client_connection.host_url
REWARDS_CLIENTCONNECTION_HOSTURL
```

Agenda

- Properties
- @ConfigurationProperties
- **Auto-Configuration**
- Overriding Configuration
- Running an Application
- Optional Topics

Spring Boot @SpringBootApplication

- @SpringBootApplication or @EnableAutoConfiguration somehow enables auto-configuration - How?

```
@SpringBootApplication
public class Application {
   ...
}
```

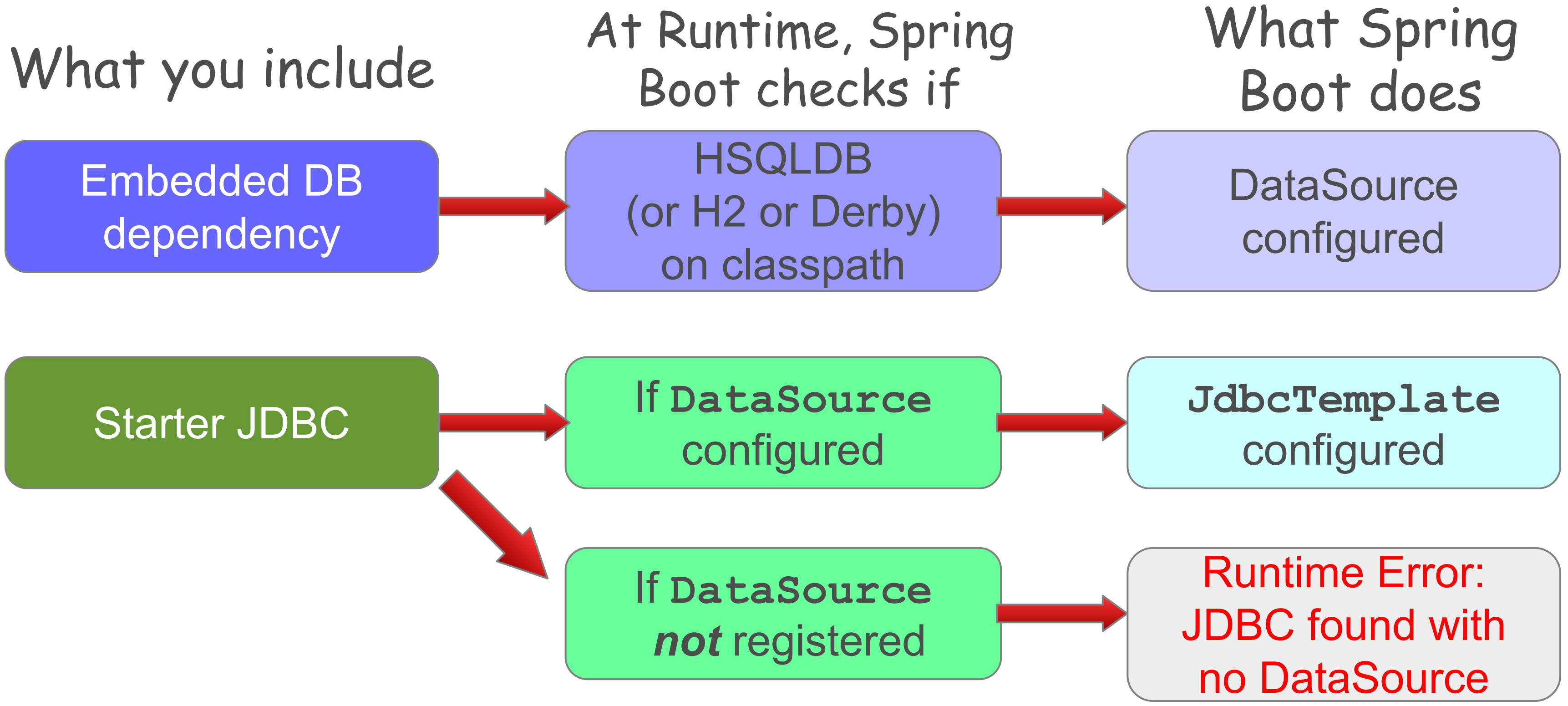
Pivotal

How Does Auto-Configuration Work?

- Extensive use of *pre-written* **`@Configuration`** classes

- Configuration of beans based on
  - The contents of the classpath
  - Properties you have set
  - Beans already defined (or not defined)

Examples of Auto-configuration: DataSource, JdbcTemplate



What you include
At Runtime, Spring Boot checks if
What Spring Boot does
Embedded DB dependency
HSQLDB (or H2 or Derby) on classpath
DataSource configured
Starter JDBC
If DataSource configured
JdbcTemplate configured
If DataSource *not* registered
Runtime Error: JDBC found with no DataSource

Pivotal

@Conditional Annotations

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

```
@Bean
@ConditionalOnBean(DataSource.class)
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
   return new JdbcTemplate(dataSource);
}
```

- Many others:
  - @ConditionalOnClass, @ConditionalOnProperty, ...
    @ConditionalOnMissingBean, @ConditionalOnMissingClass

Leverages @Conditional added in Spring 4.0. **`@Profile`** is an example of conditional configuration

Pivotal

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 { ... }
   …
}
```

Auto-configure DataSource bean only if it is not already configured

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

Pivotal

Where are these Auto-Configuration classes specified?

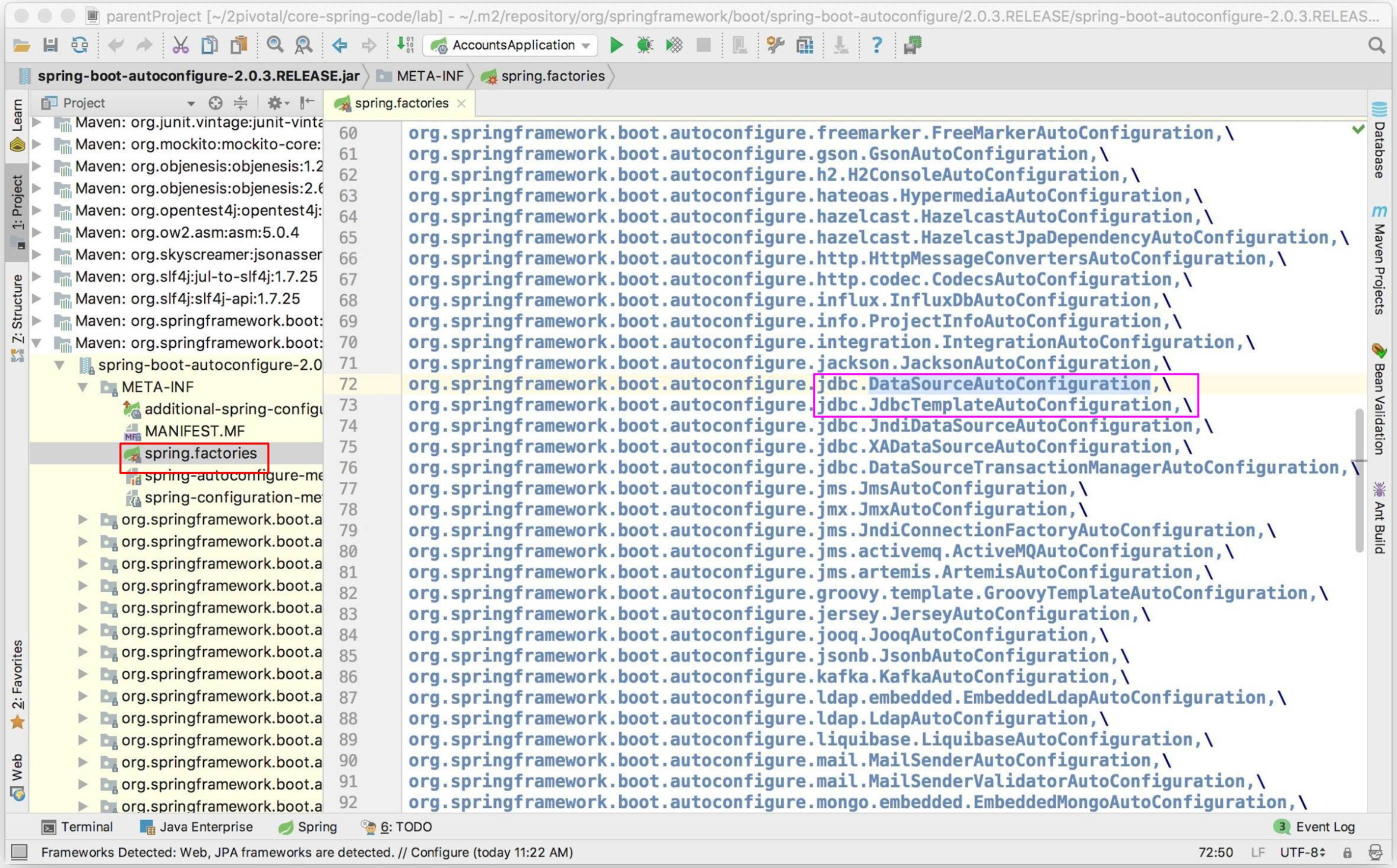
- *@EnableAutoConfiguration* reads a factories file
  - ***spring-boot-autoconfigure/META-INF/spring.factories***

- Lists the *AutoConfiguration* used by Boot.
  - May be overridden (covered next)

- Auto-configuration classes processed *after* explicitly created beans are defined
  - Beans you define always take precedence over AutoConfiguration

Pivotal

Exploring Auto-configuration classes in *spring.factories*


parentProject [~/2pivotal/core-spring-code/lab] - ~/.m2/repository/org/springframework/boot/spring-boot-autoconfigure/2.0.3.RELEASE/spring-boot-autoconfigure-2.0.3.RELEAS...
AccountsApplication
spring-boot-autoconfigure-2.0.3.RELEASE.jar
META-INF
spring.factories
Project
Maven: org.junit.vintage:junit-vinta
Maven: org.mockito:mockito-core:
Maven: org.objenesis:objenesis:1.2
Maven: org.objenesis:objenesis:2.6
Maven: org.opentest4j:opentest4j:
Maven: org.ow2.asm:asm:5.0.4
Maven: org.skyscreamer:jsonasser
Maven: org.slf4j:jul-to-slf4j:1.7.25
Maven: org.slf4j:slf4j-api:1.7.25
Maven: org.springframework.boot:
Maven: org.springframework.boot:
spring-boot-autoconfigure-2.0
META-INF
additional-spring-configu
MANIFEST.MF
spring.factories
spring-autoconfigure-me
spring-configuration-me
org.springframework.boot.a
Learn
1: Project
7: Structure
2: Favorites
Web
Database
Maven Projects
Bean Validation
Ant Build
Terminal
Java Enterprise
Spring
6: TODO
Event Log
72:50
LF
UTF-8

```
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\
org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
```

Frameworks Detected: Web, JPA frameworks are detected. // Configure (today 11:22 AM)

Agenda

- Properties
- @ConfigurationProperties
- Auto-Configuration
- **Overriding Configuration**
- Running an Application
- Optional Topics

Pivotal

Controlling Spring Boot Auto-Configuration

- Spring Boot is designed to make overriding easy.

- There are several options
  1. Set some of Spring Boot's properties
  2. Explicitly define beans yourself so Spring Boot won't
  3. Explicitly disable some auto-configuration
  4. Change dependencies

1. Set some of Spring Boot's properties

- Hundreds of pre-defined properties are available
  - Used by Spring-provided @ConfigurationProperties / *AutoConfiguration classes.

See *Common Application Properties* 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
```

Not required as it is autoconfigured typically

application.properties

Pivotal

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

Pivotal

Where to Define Spring Boot Specific Properties

- Use **`application.properties`** / **`yml`**
  - Read early enough to affect all auto-configuration possibilities
  - Some properties cannot be set in other property files
    - Such as logging levels
  - Remember: anything set in these files is easy to override

- Your own properties can be placed in any property source.

2. Explicitly define beans yourself

- Explicitly-defined beans disable auto-created ones.
  - *Example:* A **`DataSource`** you define stops Spring Boot creating a default **`DataSource`**
  - Auto-configuration generally based on bean type, not name.
  - Works regardless of how the bean is defined

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

Pivotal

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
```

Pivotal

4a. Override Dependency Versions

- Spring Boot POMs preselect the versions of dependencies
  - Ensures the versions of all dependencies are consistent
  - Simplifies dependency management in most cases

- Reasons to override default versions are:
  - 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/build.gradle

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

Pivotal

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

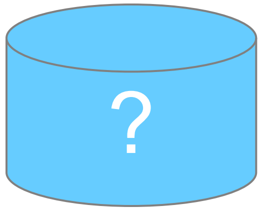
Jetty automatically detected and used!

Pivotal

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=
```

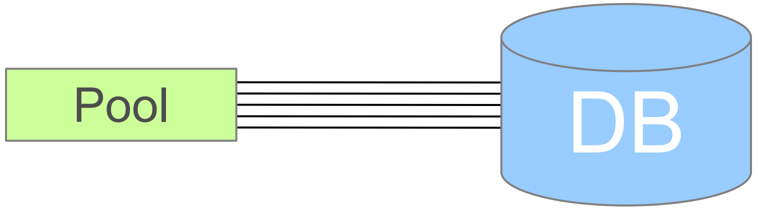
application.properties

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



Pivotal

Agenda

- Properties
- @ConfigurationProperties
- Auto-Configuration
- Overriding Configuration
- **Running an Application**
- Optional Topics

Pivotal

`CommandLineRunner` and `ApplicationRunner`

- Offers a Spring-style entry point for running applications
  - Avoids having business logic in the **`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`

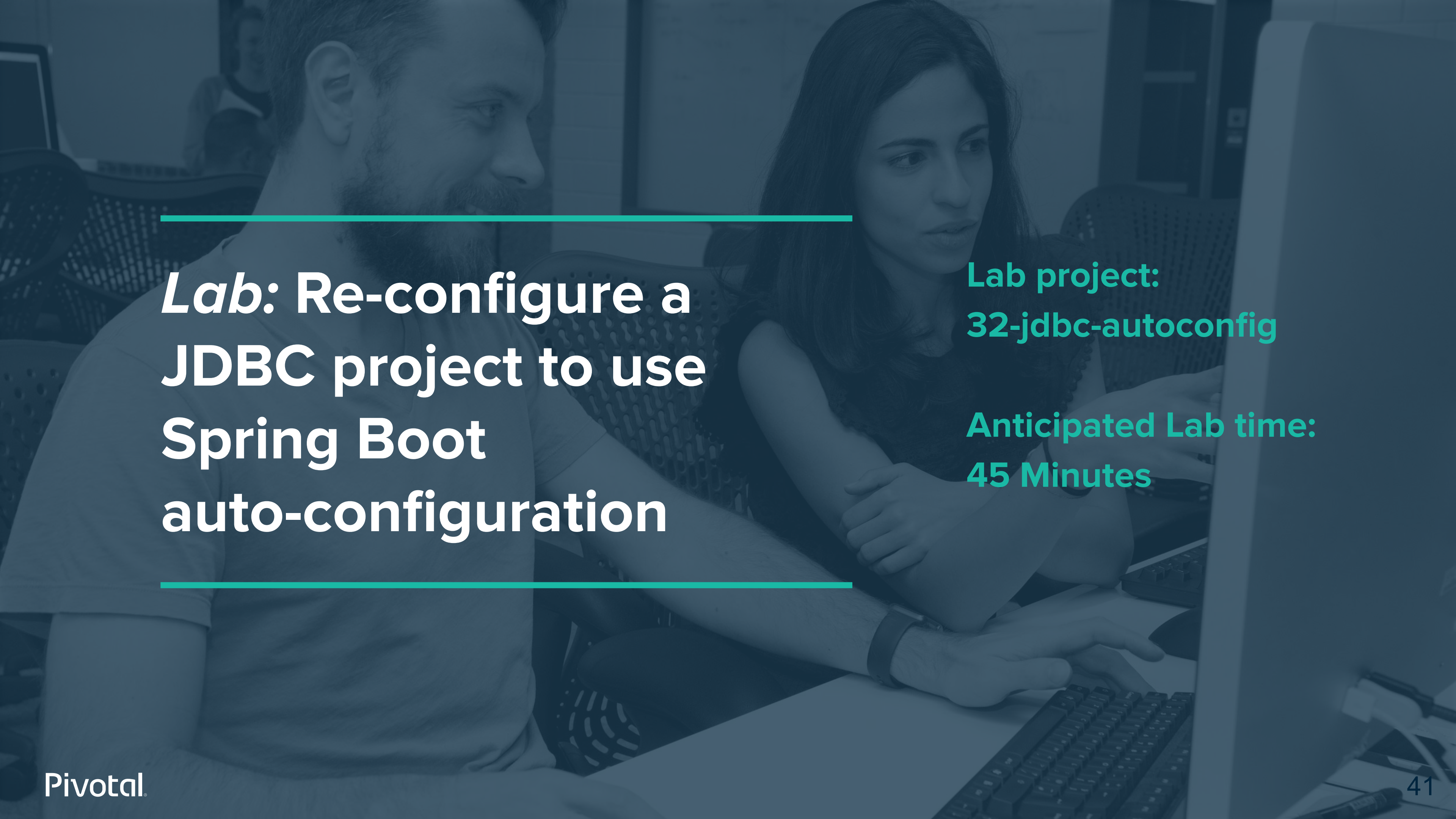
```
@SpringBootApplication
public class Application {

    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 Beans detected by Boot and invoked before returning from SpringApplication.run() but after all the beans are configured

Pivotal

Summary

- Properties can be set in application.properties / yml
- @ConfigurationProperties simplifies handling of large # of properties
- Spring Boot auto-configuration is enabled by @EnableAutoConfiguration
- AutoConfiguration can be controlled by
    1. Properties
    2. Explicit bean definition
    3. Disabling of auto configuration
    4. Altering dependency versions



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

Lab project:
32-jdbc-autoconfig

Anticipated Lab time:
45 Minutes

Pivotal

Agenda

- Properties
- @ConfigurationProperties
- Auto-Configuration
- Overriding Configuration
- Running an Application
- Optional Topics
  - **Fine-Tuning Logging**
  - Fully Executable JARs
  - DevTools

Pivotal

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.name=rewards.log

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

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

Pivotal

Agenda

- Properties
- @ConfigurationProperties
- Auto-Configuration
- Overriding Configuration
- Running an Application
- Optional Topics
  - Fine-Tuning Logging
  - **Fully Executable JARs**
  - DevTools

Variation: The Fully-Executable JAR

- A binary executable for UNIX-type systems.
  - Can be run directly from the command line
    - Without **`java -jar`**!
  - Ideal for use as OS-level service (init.d or systemd)

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <executable>true</executable>
    </configuration>
</plugin>
```

Pivotal

Agenda

- Properties
- @ConfigurationProperties
- Auto-Configuration
- Overriding Configuration
- Running an Application
- Optional Topics
  - Fine-Tuning Logging
  - Fully Executable JARs
  - **DevTools**

Pivotal

Spring Boot Developer Tools

- A set of tools to help make Spring Boot development easier
  - Automatic restart - any time a class changes (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>
```