



# Spring Boot – A Closer Look

---

Discovering how properties and auto-configuration simplifies Spring application development

## Objectives

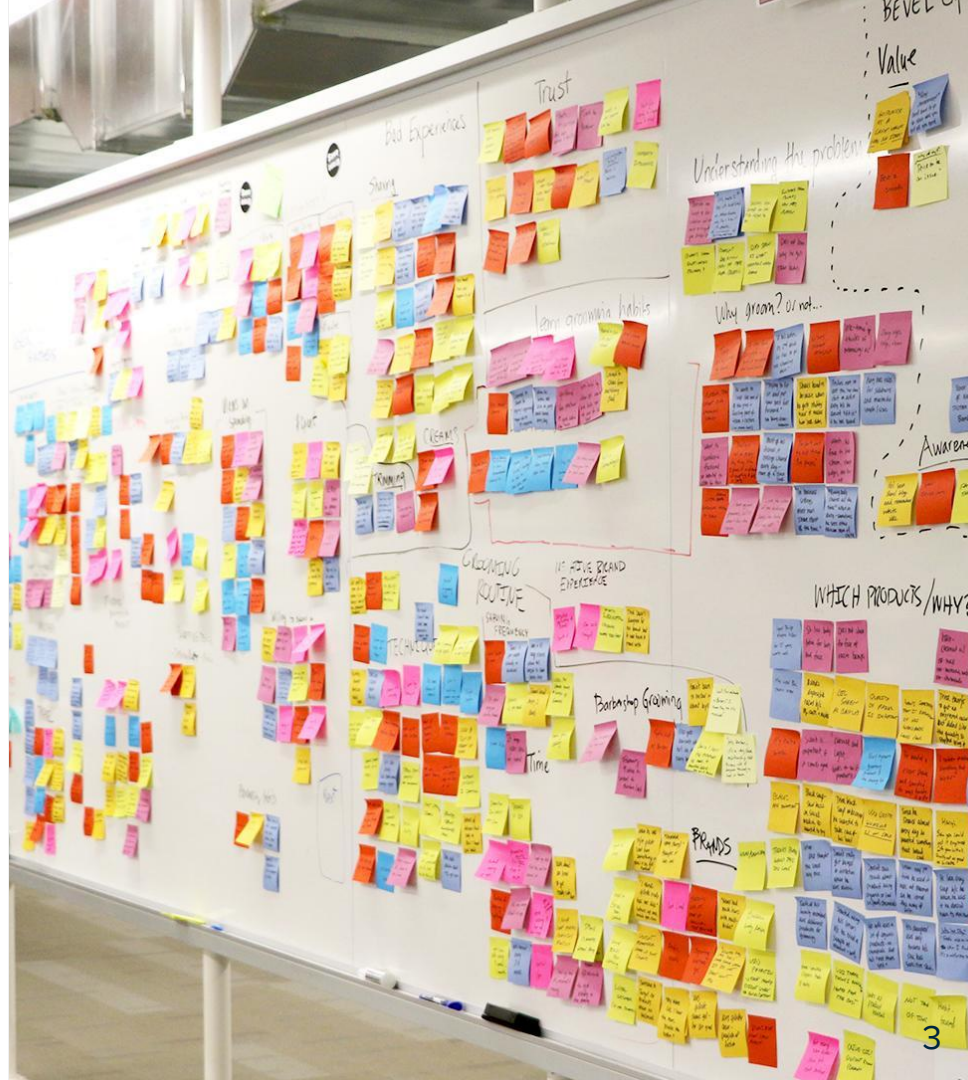
---

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



## ***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 - 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

# 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



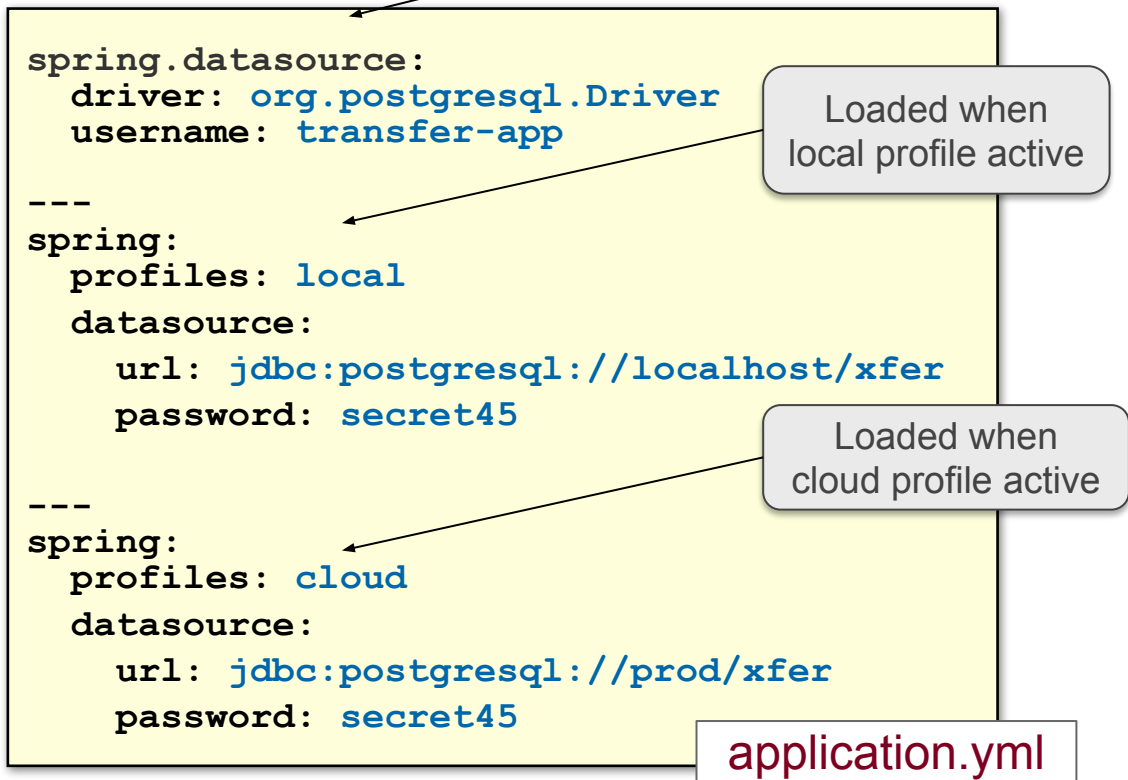
Requires snakeyaml.jar, provided by spring-boot-starter.

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





# 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.

# Agenda

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



# 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
```

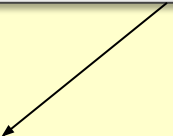
# Use @EnableConfigurationProperties

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

```
@Configuration
@EnableConfigurationProperties(ConnectionSettings.class)
public class RewardsClientConfiguration {

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

Populated with all  
properties starting with  
"rewards.client"



# 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 somehow enables auto-configuration - How?

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



# 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)

# @Conditional Annotations

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

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

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



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

# 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

# Auto-Configuration Factories

- **@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

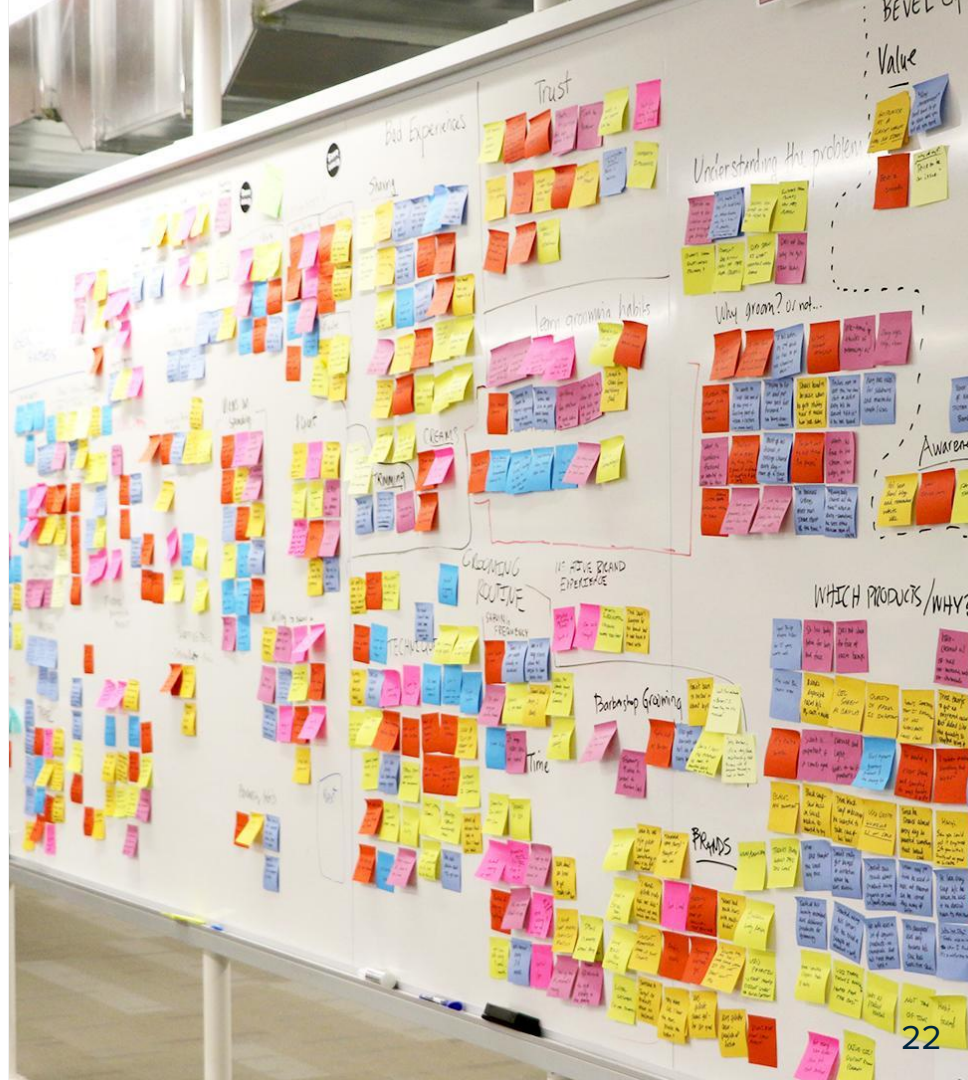
# Exploring Auto-configuration classes in *spring.factories*

The screenshot shows an IDE window titled "spring-boot-autoconfigure-2.0.3.RELEASE.jar" with the "spring.factories" file open. The file contains a list of auto-configuration classes, each followed by a backslash. The class `org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration` is highlighted with a red box. The IDE interface includes a sidebar with "Project", "Structure", and "Web" views, and a bottom status bar showing "Frameworks Detected: Web, JPA frameworks are detected. // Configure (today 11:22 AM)".

```
60 org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\n61 org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\n62 org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\n63 org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\n64 org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\n65 org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\n66 org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration,\n67 org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\n68 org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\n69 org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\n70 org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\n71 org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\n72 org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\n73 org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\n74 org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\n75 org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\n76 org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\n77 org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\n78 org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\n79 org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\n80 org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\n81 org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\n82 org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\n83 org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\n84 org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\n85 org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\n86 org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\n87 org.springframework.boot.autoconfigure ldap.embedded.EmbeddedLdapAutoConfiguration,\n88 org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\n89 org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\n90 org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\n91 org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\n92 org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
```

# Agenda

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



# 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 pre-defined `@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

## 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 Spring Boot Specific Properties



- Use `application.properties` / `yaml`
  - Read early enough to affect all auto-configuration possibilities
  - Some properties cannot be set later
    - 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 generally 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();
}
```

### 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*

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

application.properties

## 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/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

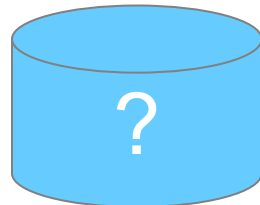
Adds Jetty

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=
```

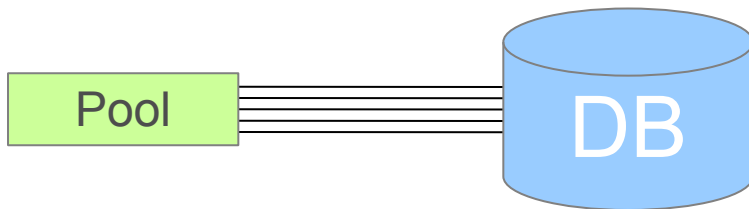
application.properties

## 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



# Agenda

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



# 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()

# Summary

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

A man and a woman are sitting at a desk in an office, looking at a computer monitor. The man is on the left, wearing a light-colored t-shirt, and the woman is on the right, wearing a dark top. They are both looking at the screen with interest. The background is slightly blurred, showing other office equipment and a person in the distance.

---

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

---

**Lab project:**  
**32-jdbc-autoconfig**

**Anticipated Lab time:**  
**45 Minutes**



# Agenda

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

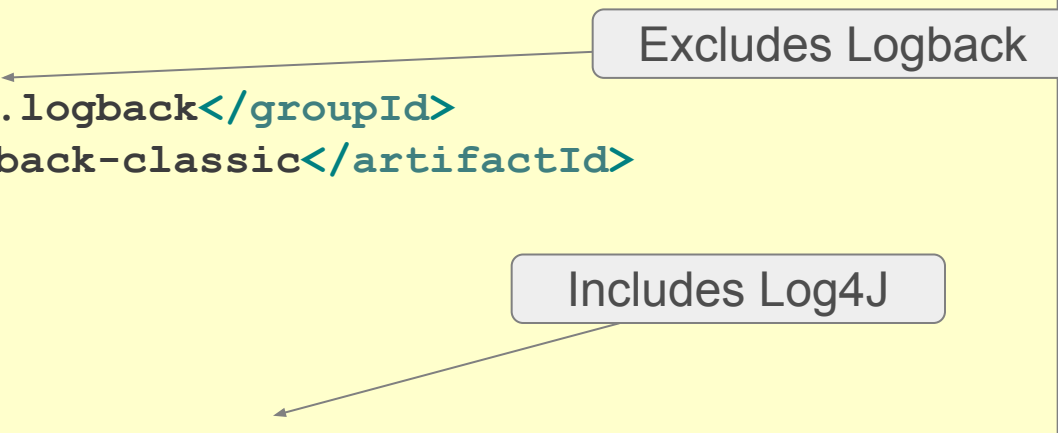


# Logging frameworks

- 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>
```



Excludes Logback

Includes Log4J

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

# 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

- 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>
```



# Agenda

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



# 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>
```