Auto Configuration

jim stafford

Fall 2022 v2022-09-13: Built: 2022-12-07 06:12 EST

Table of Contents

Chapter 1. Introduction

Thus far we have focused on how to configure an application within the primary application module, under fairly static conditions, and applied directly to a single application.

However, our application configuration will likely be required to be:

- **dynamically determined** - Application configurations commonly need to be dynamic based on libraries present, properties defined, resources found, etc. at startup. For example, what database will be used when in development, integration, or production? What security should be enabled in development versus production areas?
- **modularized and not repeated** - Breaking the application down into separate components and making these components reusable in multiple applications by physically breaking them into separate modules is a good practice. However, that leaves us with the repeated responsibility to configure the components reused. Many times there could be dozens of choices to make within a component configuration and the application can be significantly simplified if an opinionated configuration can be supplied based on the runtime environment of the module.

If you find yourself needing configurations determined dynamically at runtime or find yourself solving a repeated problem and bundling that into a library shared by multiple applications, you are going to want to master the concepts within Spring Boot's Auto-configuration capability that will be discussed here. Some of these Auto-configuraton capabilities mentioned can be placed directly into the application while others are meant to be placed into separate Auto-configuration modules called "starter" modules that can come with an opinionated, default way to configure the component for use with as little work as possible.

1.1. Goals

The student will learn to:

- Enable/disable `@Configuration` classes and `@Bean` factories based on condition(s) at startup
- Create Auto-configuration/Starter module(s) that establish necessary dependencies and conditionally supplies beans
- Resolve conflicts between alternate configurations
- Locate environment and condition details to debug Auto-configuration issues

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. Enable a `@Component`, `@Configuration` class, or `@Bean` factory method based on the result of a condition at startup
2. Create Spring Boot Auto-configuration/Starter module(s)
3. Bootstrap Auto-configuration classes into applications using a `spring.factories` metadata file
4. Create a conditional component based on the presence of a property value

5. Create a conditional component based on a missing component
6. Create a conditional component based on the presence of a class
7. Define a processing dependency order for Auto-configuration classes
8. Access textual debug information relative to conditions using the `debug` property
9. Access web-based debug information relative to conditionals and properties using the Spring Boot Actuator

Ref: Creating Your Own Auto-configuration

Chapter 2. Review: Configuration Class

As we have seen earlier, `@Configuration` classes are how we bootstrap an application using Java classes. They are the modern alternative to the legacy XML definitions that basically do the same thing — define and configure beans.

`@Configuration` classes can be the `@SpringBootApplication` class itself. This would be appropriate for a small application.

Configuration supplied within @SpringBootApplication Class

```
@SpringBootApplication
//==> wraps @EnableAutoConfiguration
//==> wraps @SpringBootConfiguration
//            ==> wraps @Configuration
public class SelfConfiguredApp {
    public static final void main(String...args) {
        SpringApplication.run(SelfConfiguredApp.class, args);
    }

    @Bean
    public Hello hello() {
        return new StdOutHello("Application @Bean says Hey");
    }
}
```

2.1. Separate @Configuration Class

`@Configuration` classes can be broken out into separate classes. This would be appropriate for larger applications with distinct areas to be configured.

```
@Configuration(proxyBeanMethods = false)
public class AConfigurationClass {
    @Bean
    public Hello hello() {
        return new StdOutHello("...");
    }
}
```

`@Configuration` classes are commonly annotated with the `proxyMethods=false` attribute that tells Spring it need not create extra proxy code to enforce normal, singleton return of the created instance to be shared by all callers since `@Configuration` class instances are only called by Spring. The javadoc for the annotation attribute describes the extra and unnecessary work saved.

Chapter 3. Conditional Configuration

We can make `@Bean` factory methods (or the `@Component` annotated class) and entire `@Configuration` classes dependent on conditions found at startup. The following example uses the @ConditionalOnProperty annotation to define a `Hello` bean based on the presence of the `hello.quiet` property equaling the value `true`.

Property Condition Example

```
...
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class StarterConfiguredApp {
    public static final void main(String...args) {
        SpringApplication.run(StarterConfiguredApp.class, args);
    }

    @Bean
    @ConditionalOnProperty(prefix="hello", name="quiet", havingValue="true") ①
    public Hello quietHello() {
        return new StdOutHello("(hello.quiet property condition set, Application @Bean
says hi)");
    }
}
```

① @ConditionalOnProperty annotation used to define a `Hello` bean based on the presence of the `hello.quiet` property equaling the value `true`

3.1. Property Value Condition Satisfied

The following is an example of the property being defined with the targeted value.

Property Value Condition Satisfied Result

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --hello.quiet=true ①
...
(hello.quiet property condition set, Application @Bean says hi) World ②
```

① matching property supplied using command line

② satisfies property condition in `@SpringBootApplication`

The `(parentheses)` is trying to indicate a *whisper*. `hello.quiet=true` property turns on this behavior.

3.2. Property Value Condition Not Satisfied

The following is an example of the property being missing. Since there is no `Hello` bean factory, we encounter an error that we will look to solve using a separate Auto-configuration module.

Property Value Condition Not Satisfied

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar ①
...
***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of constructor in info.ejava.springboot.examples.app.AppCommand required a
bean of type
  'info.ejava.examples.app.hello.Hello' that could not be found.

The following candidates were found but could not be injected: ②
    - Bean method 'quietHello' in 'StarterConfiguredApp' not loaded because
    @ConditionalOnProperty (hello.quiet=true) did not find property 'quiet'

Action:
Consider revisiting the entries above or defining a bean of type
  'info.ejava.examples.app.hello.Hello' in your configuration.
```

① property either not specified or not specified with targeted value

② property condition within `@SpringBootApplication` not satisfied

Chapter 4. Two Primary Configuration Phases

Configuration processing within Spring Boot is broken into two primary phases:

1. **User-defined configuration classes**
   - processed first
   - part of the application module
   - located through the use of a `@ComponentScan` (wrapped by `@SpringBootApplication`)
   - establish the base configuration for the application
   - fill in any fine-tuning details.
2. **Auto-configuration classes**
   - parsed second
   - outside the scope of the `@ComponentScan`
   - placed in separate modules, identified by metadata within those modules
   - enabled by application using `@EnableAutoConfiguration` (also wrapped by `@SpringBootApplication`)
   - provide defaults to fill in the reusable parts of the application
   - use User-defined configuration for details

Chapter 5. Auto-Configuration

An Auto-configuration class is technically no different than any other `@Configuration` class except that it is inspected after the User-defined `@Configuration` class(es) processing is complete and based on being named in a `META-INF/spring.factories` descriptor. This alternate identification and second pass processing allows the core application to make key directional and detailed decisions and control conditions for the Auto-configuration class(es).

The following Auto-configuration class example defines an **unconditional** `Hello` bean factory that is configured using a `@ConfigurationProperties` class.

Example Auto-Configuration Class

```
package info.ejava.examples.app.hello; ②
...

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean ①
    public Hello hello(HelloProperties helloProperties) {
        return new StdOutHello(helloProperties.getGreeting());
    }
}
```

① Example Auto-configuration class provides **unconditional** `@Bean` factory for `Hello`

② this `@Configuration` package is outside the default scanning scope of `@SpringBootApplication`

Auto-Configuration Packages are Separate from Application

Auto-Configuration classes are designed to be outside the scope of the `@SpringBootApplication` package scanning. Otherwise it would end up being a normal `@Configuration` class and processed within the main application JAR pre-processing.

```
package info.ejava.examples.app.config.auto;
@SpringBootApplication
```

```
package info.ejava.examples.app.hello; ①

@Configuration(proxyBeanMethods = false)
public class HelloAutoConfiguration {
```

① `app.hello` is not under `app.config.auto`

5.1. Supporting @ConfigurationProperties

This particular `@Bean` factory defines the `@ConfigurationProperties` class to encapsulate the details of configuring Hello. It supplies a default greeting making it optional for the User-defined configuration to do anything.

Example Auto-Configuration Properties Class

```
@ConfigurationProperties("hello")
@Data
@Validated
public class HelloProperties {
    @NotNull
    private String greeting = "HelloProperties default greeting says Hola!"; ①
}
```

① Value used if user-configuration does not specify a property value

5.2. Locating Auto Configuration Classes

Auto-configuration class(es) are registered with an entry within the `META-INF/spring.factories` file of the Auto-configuration class’s JAR. This module is typically called an "auto-configuration".

Auto-configuration Module JAR

```
$ jar tf target/hello-starter-*-SNAPSHOT-bootexec.jar | egrep -v
'/$|maven|MANIFEST.MF'
META-INF/spring.factories ①
META-INF/spring-configuration-metadata.json ②
info/ejava/examples/app/hello/HelloAutoConfiguration.class
info/ejava/examples/app/hello/HelloProperties.class
```

① "auto-configuration" dependency JAR supplies `META-INF/spring.factories`

② `@ConfigurationProperties` class metadata generated by maven plugin for use by IDEs

It is common best-practice to host Auto-configuration classes in a separate module than the beans it configures. The `Hello` interface and `Hello` implementation(s) comply with this convention and are housed in separate modules.

5.3. META-INF/spring.factories Metadata File

The Auto-configuraton classes are registered using the property name equaling the fully qualified classname of the `@EnableAutoConfiguration` annotation and the value equaling the fully qualified classname of the Auto-configuration class(es). Multiple classes can be specified separated by commas as I will show later.

```
# src/main/resources/META-INF/spring.factories
```

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  info.ejava.examples.app.hello.HelloAutoConfiguration ①
```

① Auto-configuration class metadata registration

5.4. Spring Boot 2.7 AutoConfiguration Changes

Spring Boot 2.7 has announced:

- a new `@AutoConfiguration` annotation that is meant to take the place of using `@Configuration` on top-level classes
- the deprecation of `META-INF/spring.factories` in favor of `META-INF/spring/org.springframework.boot. autoconfigure.AutoConfiguration.imports`

> For backwards compatibility, entries in spring.factories will still be honored.
>
> — Spring.io, Spring Boot 2.7.0 M2 Release Notes -- Changes to Auto-configuration

See Spring.io Examples

5.5. Example Auto-Configuration Module Source Tree

Our configuration and properties class — along with the `spring.factories` file get placed in a separate module source tree.

Example Auto-Configuration Module Structure

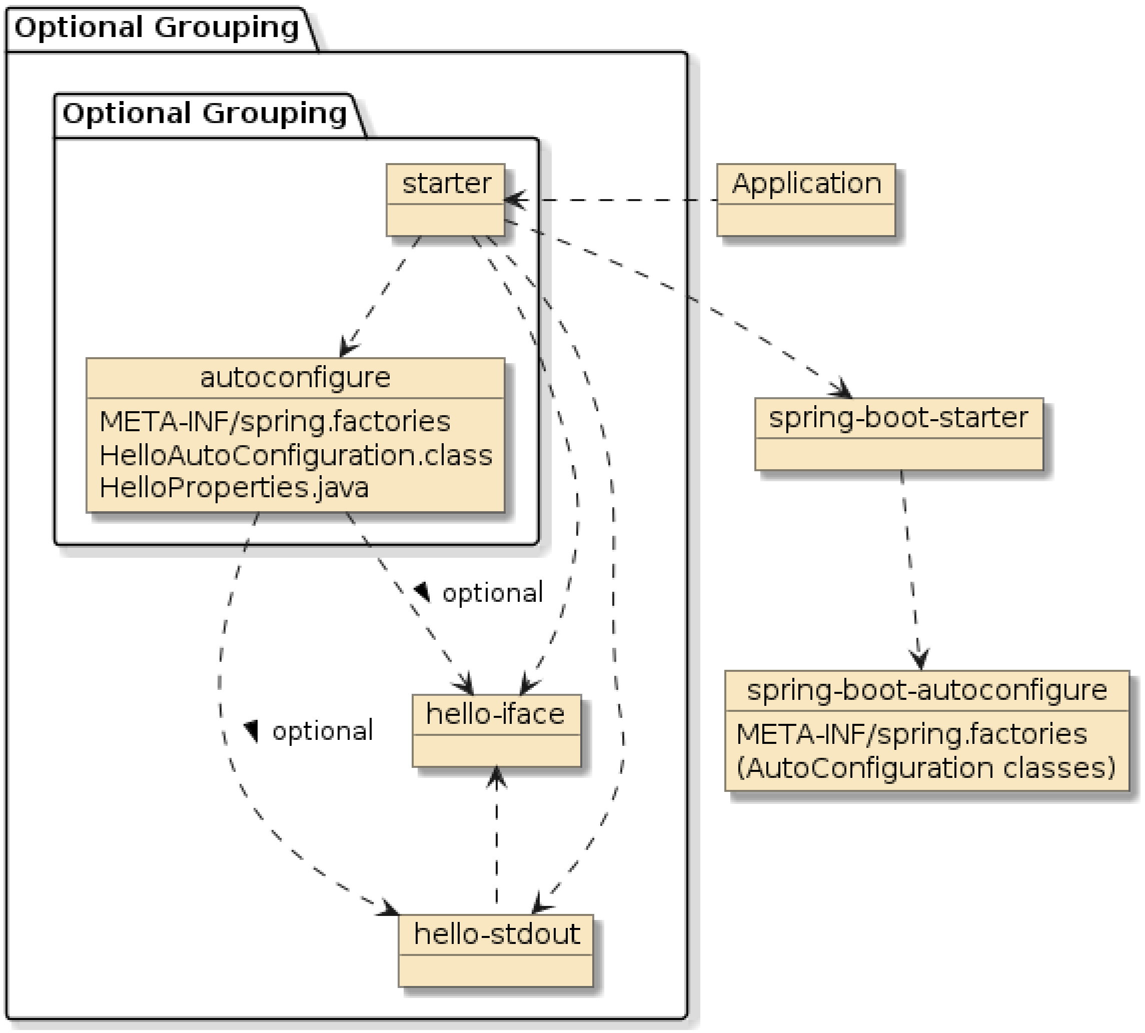
```
pom.xml
src
`-- main
    |-- java
    |   `-- info
    |       `-- ejava
    |           `-- examples
    |               `-- app
    |                   `-- hello
    |                       |-- HelloAutoConfiguration.java
    |                       `-- HelloProperties.java
    `-- resources
        `-- META-INF
            `-- spring.factories
```

5.6. Auto-Configuration / Starter Roles/Relationships

Modules designed as starters can have varying designs with the following roles carried out:

- Auto-configuration classes that conditionally wire the application

- An opinionated starter with dependencies that trigger the Auto-configuration rules



5.7. Example Starter Module pom.xml

The module is commonly termed a `starter` and will have dependencies on

- `spring-boot-starter`
- the service interface
- one or more service implementation(s) and their implementation dependencies

Example Auto-Configuration pom.xml Snippet

```
    <groupId>info.ejava.examples.app</groupId>
    <artifactId>hello-starter</artifactId>

    <dependencies>
        <dependency> ①
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <!-- commonly declares dependency on interface module -->
        <dependency> ②
            <groupId>${project.groupId}</groupId>
            <artifactId>hello-service-api</artifactId>
            <version>${project.version}</version>
        </dependency> ②
        <!-- hello implementation dependency -->
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>hello-service-stdout</artifactId>
            <version>${project.version}</version>
        </dependency>
```

① dependency on `spring-boot-starter` define classes pertinent to Auto-configuration

② `starter` modules commonly define dependencies on interface and implementation modules

5.8. Example Starter Implementation Dependencies

The rest of the dependencies have nothing specific to do with Auto-configuration or starter modules and are there to support the module implementation.

Example Starter pom.xml Implementation Dependencies

```
        <dependency> ①
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency> ①
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
        </dependency>

        <!-- creates a JSON metadata file describing @ConfigurationProperties -->
        <dependency> ①
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>
    </dependencies>
```

① these dependencies are part of optional implementation detail having nothing to do with Auto-configuration topic

5.9. Application Starter Dependency

The application module declares dependency on the starter module containing or having a dependency on the Auto-configuration artifacts.

Application Module Dependency on Starter Module

```
<!-- takes care of initializing Hello Service for us to inject -->
```

```
<dependency>
    <groupId>${project.groupId}</groupId> ①
    <artifactId>hello-starter</artifactId>
    <version>${project.version}</version> ①
</dependency>
```

① For this example, the application and starter modules share the same `groupId` and `version` and leverage a `${project}` variable to simplify the expression. That will likely not be the case with most starter module dependencies and will need to be spelled out.

5.10. Starter Brings in Pertinent Dependencies

The starter dependency brings in the Hello Service interface, targeted implementation(s), and some implementation dependencies.

Application Module Transitive Dependencies from Starter

```
$ mvn dependency:tree
...
[INFO] +- info.ejava.examples.app:hello-starter:jar:6.0.1-SNAPSHOT:compile
[INFO] |  +- info.ejava.examples.app:hello-service-api:jar:6.0.1-SNAPSHOT:compile
[INFO] |  +- info.ejava.examples.app:hello-service-stdout:jar:6.0.1-SNAPSHOT:compile
[INFO] |  +- org.projectlombok:lombok:jar:1.18.10:provided
[INFO] |  \- org.springframework.boot:spring-boot-starter-validation:jar:2.7.0:compile
...
```

Chapter 6. Configured Application

The example application contains a component that requests the greeter implementation to say hello to "World".

Injection Point for Auto-configuration Bean

```
import lombok.RequiredArgsConstructor;
...
@Component
@RequiredArgsConstructor ①
public class AppCommand implements CommandLineRunner {
    private final Hello greeter;

    public void run(String... args) throws Exception {
        greeter.sayHello("World");
    }
}
```

① lombok is being used to provide the constructor injection

6.1. Review: Unconditional Auto-Configuration Class

This starter dependency is bringing in a `@Bean` factory to construct an implementation of `Hello`.

Example Auto-Configuration Class

```
package info.ejava.examples.app.hello;
...

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean
    public Hello hello(HelloProperties helloProperties) { ①
        return new StdOutHello(helloProperties.getGreeting());
    }
}
```

① Example Auto-configuration configured by `HelloProperties`

6.2. Review: Starter Module Default

The starter dependency brings in an Auto-configuration class that instantiates a `StdOutHello` implementation configured by a `HelloProperties` class.

Review: Auto-configuration class` Configuration Properties

```
@ConfigurationProperties("hello")
```

```
@Data
@Validated
public class HelloProperties {
    @NotNull
    private String greeting = "HelloProperties default greeting says Hola!"; ①
}
```

① `hello.greeting` default defined in `@ConfigurationProperties` class of starter/autoconfigure module

6.3. Produced Default Starter Greeting

This produces the default greeting

Example Application Execution without Satisfying Property Condition

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar
...
HelloProperties default greeting says Hola! World
```

6.4. User-Application Supplies Property Details

Since the Auto-configuration class is using a properties class, we can define properties (aka "the details") in the main application for the dependency module to use.

application.properties

```
#appconfig-autoconfig-example application.properties
#uncomment to use this greeting
hello.greeting: application.properties Says - Hey
```

Runtime Output with hello.greeting Property Defined

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar
...
application.properties Says - Hey World ①
```

① auto-configured implementation using user-defined property

Chapter 7. Auto-Configuration Conflict

7.1. Review: Conditional @Bean Factory

We saw how we could make a `@Bean` factory in the User-defined application module conditional (on the value of a property).

Conditional @Bean Factory

```
@SpringBootApplication
public class StarterConfiguredApp {
...
    @Bean
    @ConditionalOnProperty(prefix = "hello", name = "quiet", havingValue = "true")
    public Hello quietHello() {
        return new StdOutHello("(hello.quiet property condition set, Application @Bean
says hi)");
    }
}
```

7.2. Potential Conflict

We also saw how to define `@Bean` factory in an Auto-configuration class brought in by starter module. We now have a condition where the two can cause an ambiguity error that we need to account for.

Example Output with Bean Factory Ambiguity

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --hello.quiet=true ①
...
***************************
APPLICATION FAILED TO START
***************************
Description:

Parameter 0 of constructor in info.ejava.examples.app.config.auto.AppCommand
   required a single bean, but 2 were found:
    - quietHello: defined by method 'quietHello' in
         info.ejava.examples.app.config.auto.StarterConfiguredApp
    - hello: defined by method 'hello' in class path resource
         [info/ejava/examples/app/hello/HelloAutoConfiguration.class]

Action:

Consider marking one of the beans as @Primary, updating the consumer to accept
multiple beans,
or using @Qualifier to identify the bean that should be consumed
```

① Supplying the `hello.quiet=true` property value causes two `@Bean` factories to chose from

7.3. @ConditionalOnMissingBean

One way to solve the ambiguity is by using the @ConditionalOnMissingBean annotation — which defines a condition based on the absence of a bean. Most conditional annotations can be used in both the application and autoconfigure modules. However, the `@ConditionalOnMissingBean` and its sibling @ConditionalOnBean are special and meant to be used with Auto-configuration classes in the autoconfigure modules.

Since the Auto-configuration classes are processed after the User-defined classes — there is a clear point to determine whether a User-defined `@Bean` factory does or does not exist. Any other use of these two annotations requires careful ordering and is not recommended.

@ConditionOnMissingBean Auto-Configuration Example

```
...
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean ①
    public Hello hello(HelloProperties helloProperties) {
        return new StdOutHello(helloProperties.getGreeting());
    }
}
```

① `@ConditionOnMissingBean` causes Auto-configured `@Bean` method to be inactive when `Hello` bean already exists

7.4. Bean Conditional Example Output

With the `@ConditionalOnMissingBean` defined on the Auto-configuration class and the property condition satisfied, we get the bean injected from the User-defined `@Bean` factory.

Runtime with Property Condition Satisfied

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --hello.quiet=true
...
(hello.quiet property condition set, Application @Bean says hi) World
```

With the property condition not satisfied, we get the bean injected from the Auto-configuration `@Bean` factory. Wahoo!

Runtime with Property Condition Not Satisfied

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar
```

```
...
application.properties Says - Hey World
```

Chapter 8. Resource Conditional and Ordering

We can also define a condition based on the presence of a resource on the filesystem or classpath using the @ConditionOnResource. The following example satisfies the condition if the file hello.properties exists in the current directory. We are also going to order our Auto-configured classes with the help of the @AutoConfigureBefore annotation. There is a sibling @AutoConfigureAfter annotation as well as a AutoConfigureOrder we could have used.

Example Condition on File Present and Evaluation Ordering

```
...
import org.springframework.boot.autoconfigure.AutoConfigureBefore;
import org.springframework.boot.autoconfigure.condition.ConditionalOnResource;

@ConditionalOnResource(resources = "file:./hello.properties") ①
@AutoConfigureBefore(HelloAutoConfiguration.class) ②
public class HelloResourceAutoConfiguration {
    @Bean
    public Hello resourceHello() {
        return new StdOutHello("hello.properties exists says hello");
    }
}
```

① Auto-configured class satisfied only when file hello.properties present

② This Auto-configuration class is processed prior to HelloAutoConfiguration

8.1. Registering Second Auto-Configuration Class

This second Auto-configuration class is being provided in the same, hello-starter module, so we need to update the Auto-configuration property within the META-INF/spring.factories file. We do this by listing the full classnames of each Auto-configuration class, separated by comma(s).

hello-starter spring.factories

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  info.ejava.examples.app.hello.HelloAutoConfiguration, \ ①
  info.ejava.examples.app.hello.HelloResourceAutoConfiguration
```

① comma separated

8.2. Resource Conditional Example Output

The following execution with hello.properties present in the current directory satisfies the condition, causes the @Bean factory from HelloAutoConfiguration to be skipped because the bean already exists.

Resource Condition Satisfied

```
$ echo hello.greeting: hello.properties exists says hello World > hello.properties
$ cat hello.properties
hello.greeting: hello.properties exists says hello World

$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar
...
hello.properties exists says hello World
```

- when property file is not present
  - `@Bean` factory from `HelloAutoConfiguration` used since neither property or resource-based conditions satisfied

Resource Condition Not Satisfied

```
$ rm hello.properties
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar
...
application.properties Says - Hey World
```

Chapter 9. @Primary

In the previous example I purposely put ourselves in a familiar situation to demonstrate an alternative solution if appropriate. We will re-enter the ambiguous match state if we supply a `hello.properties` file and the `hello.quiet=true` property value.

Example Ambiguous Conditional Match

```
$ touch hello.properties
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --hello.quiet=true
...
***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of constructor in info.ejava.examples.app.config.auto.AppCommand required
a single bean,
  but 2 were found:
    - quietHello: defined by method 'quietHello' in
info.ejava.examples.app.config.auto.StarterConfiguredApp
    - resourceHello: defined by method 'resourceHello' in class path resource
      [info/ejava/examples/app/hello/HelloResourceAutoConfiguration.class]


Action:

Consider marking one of the beans as @Primary, updating the consumer to accept
multiple beans,
or using @Qualifier to identify the bean that should be consumed
```

This time — to correct — we want the resource-based `@Bean` factory to take priority so we add the @Primary annotation to our highest priority `@Bean` factory. If there is a conflict — this one will be used.

```
...
import org.springframework.context.annotation.Primary;

@ConditionalOnResource(resources = "file:./hello.properties")
@AutoConfigureBefore(HelloAutoConfiguration.class)
public class HelloResourceAutoConfiguration {
    @Bean
    @Primary //chosen when there is a conflict
    public Hello resourceHello() {
        return new StdOutHello("hello.properties exists says hello");
    }
}
```

9.1. @Primary Example Output

This time we avoid the error with the same conditions met and one of the `@Bean` factories listed as `@Primary` to resolve the conflict.

Ambiguous Choice Resolved thru @Primary

```
$ cat hello.properties
hello.greeting: hello.properties exists says hello World
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --hello.quiet=true ①
...
hello.properties exists says hello World
```

① `@Primary` condition satisfied overrides application `@Bean` condition

Chapter 10. Class Conditions

There are many conditions we can add to our `@Configuration` class or methods. However, there is an important difference between the two.

- class conditional annotations prevent the entire class from loading when not satisfied
- `@Bean` factory conditional annotations allow the class to load but prevent the method from being called when not satisfied

This works for missing classes too! Spring Boot parses the conditional class using ASM to detect and then evaluate conditions prior to allowing the class to be loaded into the JVM. Otherwise we would get a `ClassNotFoundException` for the import of a class we are trying to base our condition on.

10.1. Class Conditional Example

In the following example, I am adding @ConditionalOnClass annotation to prevent the class from being loaded if the implementation class does not exist on the classpath.

```
...
import info.ejava.examples.app.hello.stdout.StdOutHello; ②
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(StdOutHello.class) ②
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public Hello hello(HelloProperties helloProperties) {
        return new StdOutHello(helloProperties.getGreeting()); ①
    }
}
```

① `StdOutHello` is the implementation instantiated by the `@Bean` factory method

② HelloAutoConfiguration.class will not get loaded if `StdOutHello.class` does not exist

The `@ConditionOnClass` accepts either a class or string expression of the fully qualified classname. The sibling @ConditionalOnMissingClass accepts only the string form of the classname.

Spring Boot Autoconfigure module contains many examples of real Auto-configuration classes

Chapter 11. Excluding Auto Configurations

We can turn off certain Auto-configured classes using the

- `exclude` attribute of the `@EnableAutoConfiguration` annotation
- `exclude` attribute of the `@SpringBootApplication` annotation which wraps the `@EnableAutoConfiguration` annotation

```
@SpringBootApplication(exclude = {})
// ==> wraps @EnableAutoConfiguration(exclude={})
public class StarterConfiguredApp {
    ...
}
```

Chapter 12. Debugging Auto Configurations

With all these conditional User-defined and Auto-configurations going on, it is easy to get lost or make a mistake. There are two primary tools that can be used to expose the details of the conditional configuration decisions.

12.1. Conditions Evaluation Report

It is easy to get a simplistic textual report of positive and negative condition evaluation matches by adding a `debug` property to the configuration. This can be done by adding `--debug` or -Ddebug to the command line.

The following output shows only the positive and negative matching conditions relevant to our example. There is plently more in the full output.

12.2. Conditions Evaluation Report Example

Conditions Evaluation Report Snippet

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar --debug | less
...
============================
CONDITIONS EVALUATION REPORT
============================

Positive matches: ①
-----------------
   HelloAutoConfiguration matched:
      - @ConditionalOnClass found required class
'info.ejava.examples.app.hello.stdout.StdOutHello' (OnClassCondition)

   HelloAutoConfiguration#hello matched:
      - @ConditionalOnBean (types: info.ejava.examples.app.hello.Hello;
SearchStrategy: all) did not find any beans (OnBeanCondition)

Negative matches: ②
-----------------
   HelloResourceAutoConfiguration:
      Did not match:
         - @ConditionalOnResource did not find resource 'file:./hello.properties'
(OnResourceCondition)

   StarterConfiguredApp#quietHello:
      Did not match:
         - @ConditionalOnProperty (hello.quiet=true) did not find property 'quiet'
(OnPropertyCondition)
```

① Positive matches show which conditionals are activated and why

② Negative matches show which conditionals are not activated and why

12.3. Condition Evaluation Report Results

The report shows us that

- `HelloAutoConfiguration` class was enabled because `StdOutHello` class was present
- `hello @Bean` factory method of `HelloAutoConfiguration` class was enabled because no other beans were located
- entire `HelloResourceAutoConfiguration` class was not loaded because file `hello.properties` was not present
- `quietHello @Bean` factory method of application class was not activated because `hello.quiet` property was not found

12.4. Actuator Conditions

We can also get a look at the conditionals while the application is running for Web applications using the Spring Boot Actuator. However, doing so requires that we transition our application from a command to a Web application. Luckily this can be done technically by simply changing our starter in the pom.xml file.

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

We also need to add a dependency on the `spring-boot-starter-actuator` module.

```
        <!-- added to inspect env -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
```

12.5. Activating Actuator Conditions

The Actuator, by default, will not expose any information without being configured to do so. We can show a JSON version of the Conditions Evaluation Report by adding the `management.endpoints.web.exposure.include` equal to the value `conditions`. I will do that on the command line here. Normally it would be in a profile-specific properties file appropriate for exposing this information.

Enable Actuator Conditions Report to be Exposed

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar \
  --management.endpoints.web.exposure.include=conditions
```

The Conditions Evaluation Report is available at the following URL: http://localhost:8080/actuator/conditions.

Example Actuator Conditions Report

```
{
"contexts": {
  "application": {
    "positiveMatches": {
        "HelloAutoConfiguration": [{
            "condition": "OnClassCondition",
            "message": "@ConditionalOnClass found required class
'info.ejava.examples.app.hello.stdout.StdOutHello'"
            }],
        "HelloAutoConfiguration#hello": [{
            "condition": "OnBeanCondition",
            "message": "@ConditionalOnBean (types:
info.ejava.examples.app.hello.Hello; SearchStrategy: all) did not find any beans"
            }],
...
,
    "negativeMatches": {
        "StarterConfiguredApp#quietHello": {
            "notMatched": [{
            "condition": "OnPropertyCondition",
            "message": "@ConditionalOnProperty (hello.quiet=true) did not find
property 'quiet'"
            }],
            "matched": []
            },
        "HelloResourceAutoConfiguration": {
            "notMatched": [{
            "condition": "OnResourceCondition",
            "message": "@ConditionalOnResource did not find resource
'file:./hello.properties'"
            }],
            "matched": []
            },
...
```

12.6. Actuator Environment

It can also be helpful to inspect the environment to determine the value of properties and which source of properties is being used. To see that information, we add `env` to the `exposure.include` property.

Enable Actuator Conditions Report and Environment to be Exposed

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar \
   --management.endpoints.web.exposure.include=conditions,env
```

12.7. Actuator Links

This adds a full `/env` endpoint and a view specific `/env/{property}` endpoint to see information for a specific property name. The available Actuator links are available at http://localhost:8080/actuator.

Actuator Links

```
{
    _links: {
        self: {
            href: "http://localhost:8080/actuator",
            templated: false
    },
        conditions: {
            href: "http://localhost:8080/actuator/conditions",
            templated: false
    },
        env: {
            href: "http://localhost:8080/actuator/env",
            templated: false
    },
        env-toMatch: {
            href: "http://localhost:8080/actuator/env/{toMatch}",
            templated: true
        }
    }
}
```

12.8. Actuator Environment Report

The Actuator Environment Report is available at http://localhost:8080/actuator/env.

Example Actuator Environment Report

```
{
activeProfiles: [ ],
propertySources: [{
        name: "server.ports",
        properties: {
            local.server.port: {
                value: 8080
                }
            }
    },
    {
        name: "commandLineArgs",
        properties: {
            management.endpoints.web.exposure.include: {
                value: "conditions,env"
                }
            }
    },
...
```

12.9. Actuator Specific Property Source

The source of a specific property and its defined value is available below the `/actuator/env` URI such that the `hello.greeting` property is located at http://localhost:8080/actuator/env/hello.greeting.

Example Actuator Environment Report for Specific Property

```
{
  property: {
  source: "applicationConfig: [classpath:/application.properties]",
  value: "application.properties Says - Hey"
},
...
```

12.10. More Actuator

We can explore some of the other Actuator endpoints by changing the include property to * and revisiting the main actuator endpoint. Actuator Documentation is available on the web.

Expose All Actuator Endpoints

```
$ java -jar target/appconfig-autoconfig-*-SNAPSHOT-bootexec.jar \
   --management.endpoints.web.exposure.include="*" ①
```

① double quotes ("") being used to escape * special character on command line

Chapter 13. Summary

In this module we:

- Defined conditions for `@Configuration` classes and `@Bean` factory methods that are evaluated at runtime startup
- Placed User-defined conditions, which are evaluated first, in with with application module
- Placed Auto-configuration classes in separate `starter` module to automatically bootstrap applications with specific capabilities
- Added conflict resolution and ordering to conditions to avoid ambiguous matches
- Discovered how class conditions can help prevent entire `@Configuration` classes from being loaded and disrupt the application because an optional class is missing
- Learned how to debug conditions and visualize the runtime environment through use of the `debug` property or by using the Actuator for web applications