Hands-On Spring Boot in 3 Weeks

Andy Olsen

andyo@olsensoft.com

Course Contents - Week 1

1. Introduction to Spring Boot
2. Creating a Simple Spring Boot App
3. Creating a Spring Boot Web App
4. Beans and Dependency Injection
5. Injection Techniques
6. Configuration Classes
7. Spring Boot Techniques

Course Contents - Week 2

8. Integrating with Data Sources
9. Querying and Modifying Entities
10. Spring Data Repositories
11. Implementing a Simple REST Service
12. Implementing a Full REST Service
13. Consuming REST Services

Course Contents - Week 3

14. Messaging with Kafka
15. Containerizing a Spring Boot App
16. Spring Cloud Microservices
17. Authentication using OAuth2
18. Testing Spring Boot Applications

Full Materials Online

- You can get full materials for this course online
  - https://github.com/andyolsen/spring-boot-in-3-weeks

- What's there?
  - Slides and demos
  - Weekly assignments and full solutions

Introduction to Spring Boot

1. Overview of Spring Boot
2. Tooling up

Section 1. Overview of Spring Boot

- What is Spring Boot?
- Getting Spring Boot
- Spring Boot documentation
- What can you do with Spring Boot?
- Spring Boot in the enterprise

What is Spring Boot?

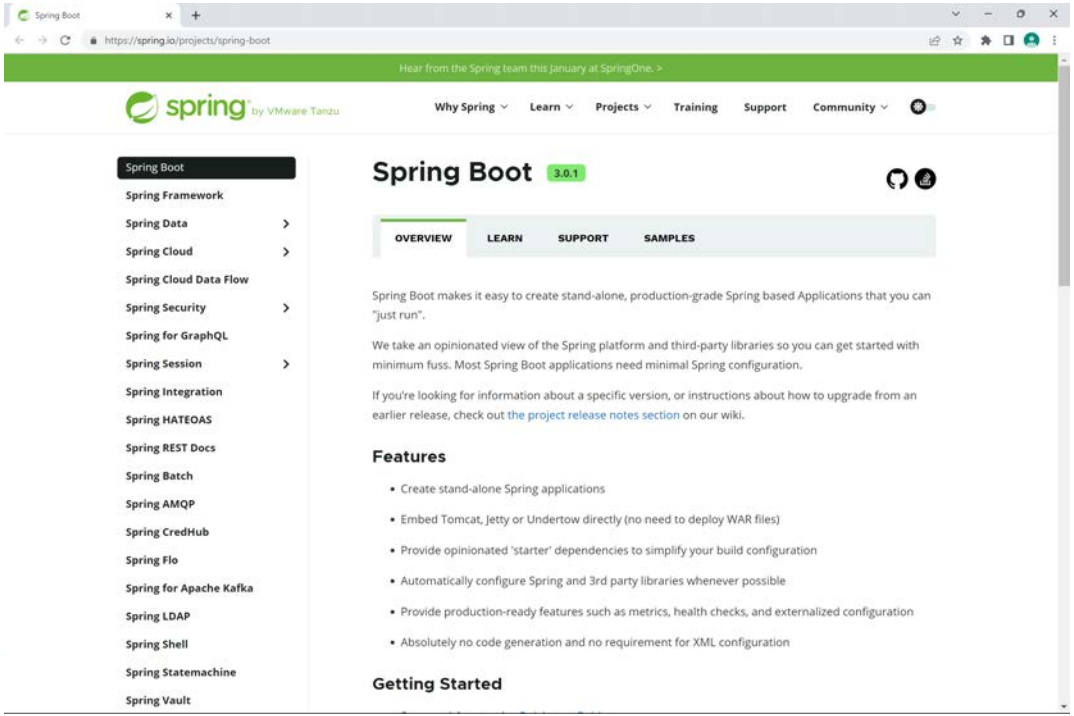
- Spring Boot is a popular Java-based framework from Pivotal
  - Simplifies creating and running enterprise applications

- Allows you to create completely standalone applications
  - Applications can contain built-in servers, e.g. Tomcat or Jetty
  - No need for an external web server host to run on
  - You can just run a Spring Boot application as a regular Java app

```
java –jar MySpringBootApp.jar
```

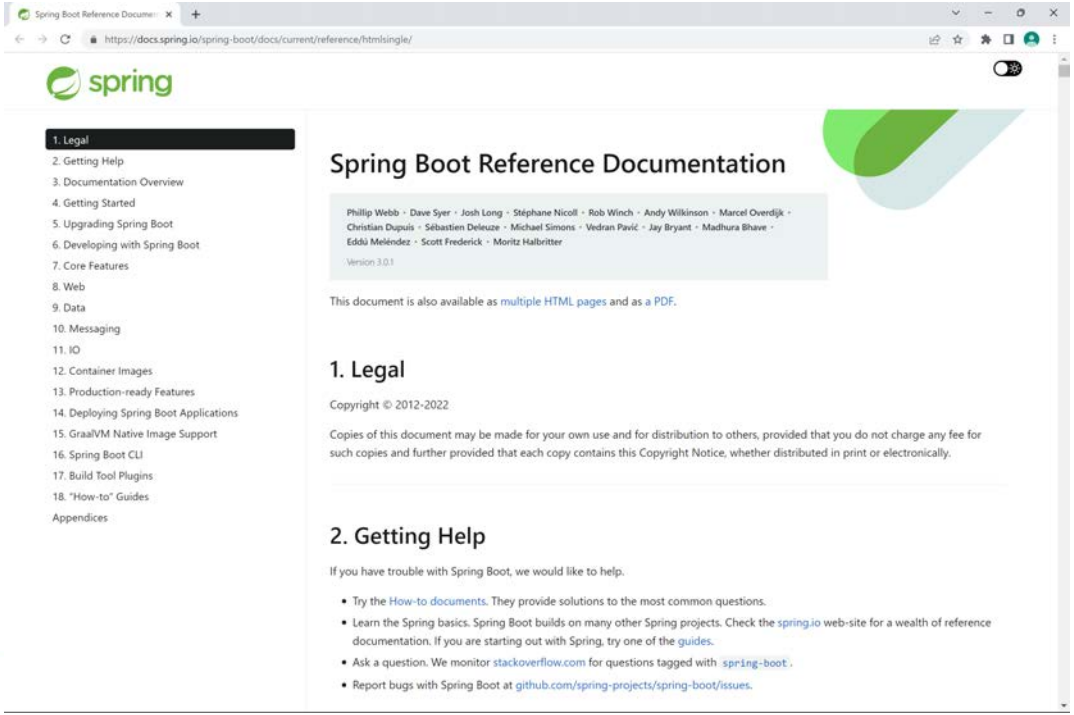
Getting Spring Boot

- You can get the Spring Boot project here:
  - https://spring.io/projects/spring-boot



Spring Boot Documentation

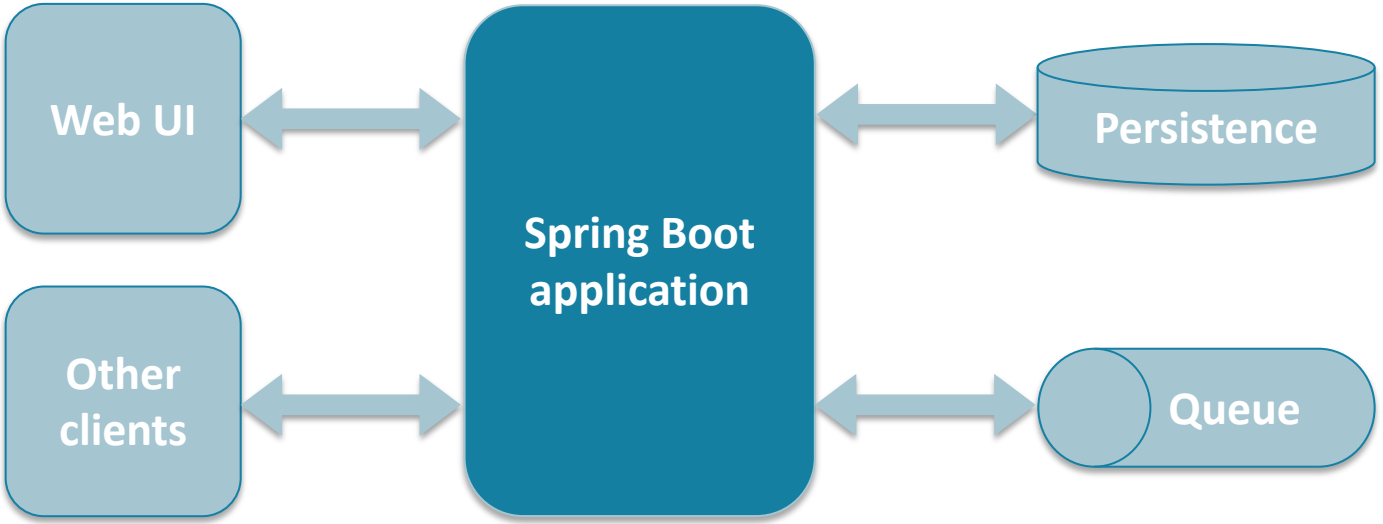
- Extensive Spring Boot documentation is available here:
  - https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/



What Can You Do With Spring Boot?

- Spring Boot has extensive APIs that address all aspects of contemporary systems
  - REST services and Web applications
  - SQL and NoSQL data sources
  - Messaging, e.g. with Kafka
  - Spring Batch
  - Spring Cloud
  - Web sockets
  - Plus testing, security, logging, etc.

Spring Boot in the Enterprise



Section 2: Tooling Up

- Overview
- Setting up the Java Development Kit (JDK)
- Setting up IntelliJ IDEA Ultimate Edition

Overview

- We're going to use the following tools during this video series:
  - Java Standard Edition Development Kit (JDK) version 17
  - IntelliJ IDEA Ultimate Edition

- Let's see how to set up these tools…

Setting up the Java Development Kit

- You can install the Oracle JDK, which is available here:
  - https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html
- Alternatively, you can install OpenJDK, which is available here:
  - https://openjdk.java.net/install/
- You must also set two environment variables:
  - `JAVA_HOME` - must point to your JDK folder
  - `PATH` - must include your JDK binary folder

Setting up IntelliJ IDEA Ultimate Edition

- You can install IntelliJ IDEA Ultimate Edition from here:
  - https://www.jetbrains.com/idea/
- Make sure you install the Ultimate Edition
  - You can install a free 30-day trial if you like
- We'll use IntelliJ IDEA to create and run Spring Boot apps

Summary

- Overview of Spring Boot
- Tooling up

Creating a Simple Spring Boot App

1. Creating a Spring Boot app in IntelliJ
2. Understanding the application

1. Creating a Spring Boot App in IntelliJ

- Overview of IntelliJ
- Creating a Spring Boot project
- Specifying project dependencies

Overview of IntelliJ

- IntelliJ IDEA Ultimate has excellent support for Spring
  - Spring Boot and Spring Framework

- IntelliJ Java dependencies:
  - JDK (e.g. JDK 17)
  - Set `JAVA_HOME` to the JDK folder
  - Set `PATH` to include the JDK binary folder

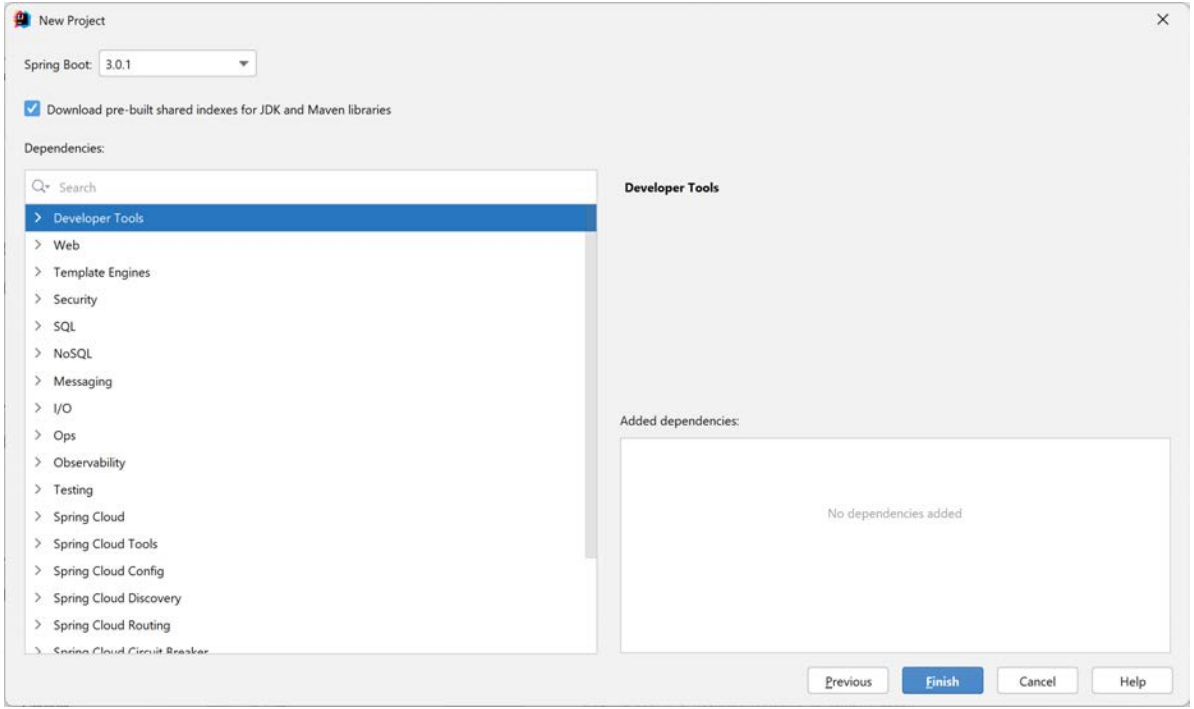
Creating a Spring Boot Project

- Start IntelliJ, click New Project, and select Spring Initializr

- Specify project info as follows:
  - Enter a suitable project name and location
  - Language - Java
  - Type - Maven
  - Enter a suitable group ID, artifact ID, and package name
  - Project SDK - Java 17
  - Java version - 17
  - Packaging - Jar
  - Then click Next

Specifying Project Dependencies

- In the next window you can add dependencies to your project
  - We don't need any dependencies yet, so just click Finish

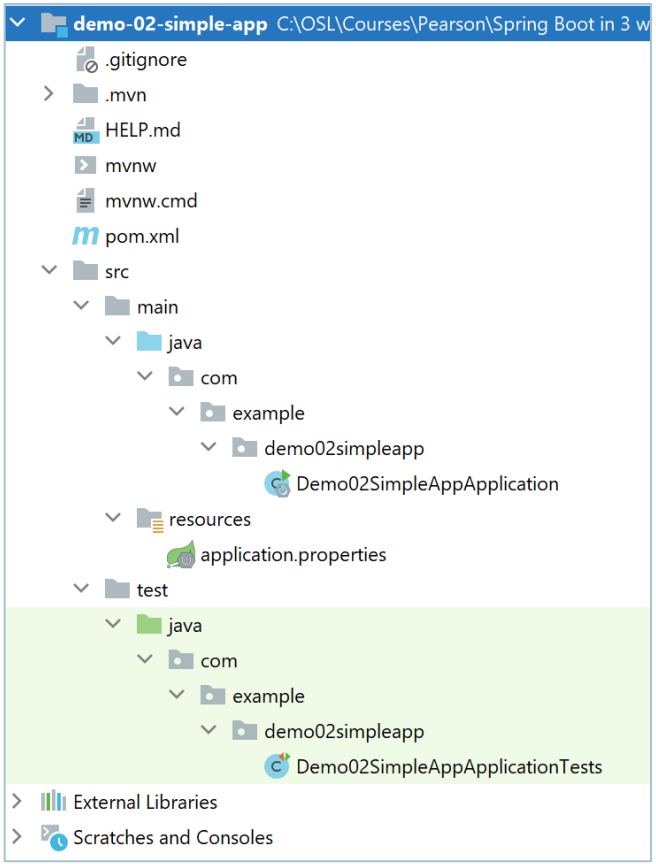


2. Understanding the Application

- Understanding the application structure
- Understanding the Maven POM file
- Understanding the application code
- Running the application
- Viewing the application output

Understanding the Application Structure

- The generated application is a regular Maven project



Understanding the Maven POM File

- Here are the relevant sections in the Maven pom file:

```
<project … >
    …
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.1</version>
        <relativePath/>
    </parent>

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

Parent POM

Spring Boot dependency

Spring Boot test dependency

pom.xml

Understanding the Application Code

- Here's the generated application code:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Demo02SimpleAppApplication {

    public static void main(String[] args) {
        SpringApplication.run(Demo02SimpleAppApplication.class, args);
    }
}
```

Demo02SimpleAppApplication.java

- `@SpringBootApplication` is equivalent to:
  - `@Configuration`
  - `@EnableAutoConfiguration`
  - `@ComponentScan`

Running the Application

- You can build/run the app via the `mvn` command in the project root directory as follows:

```
mvn spring-boot:run
```

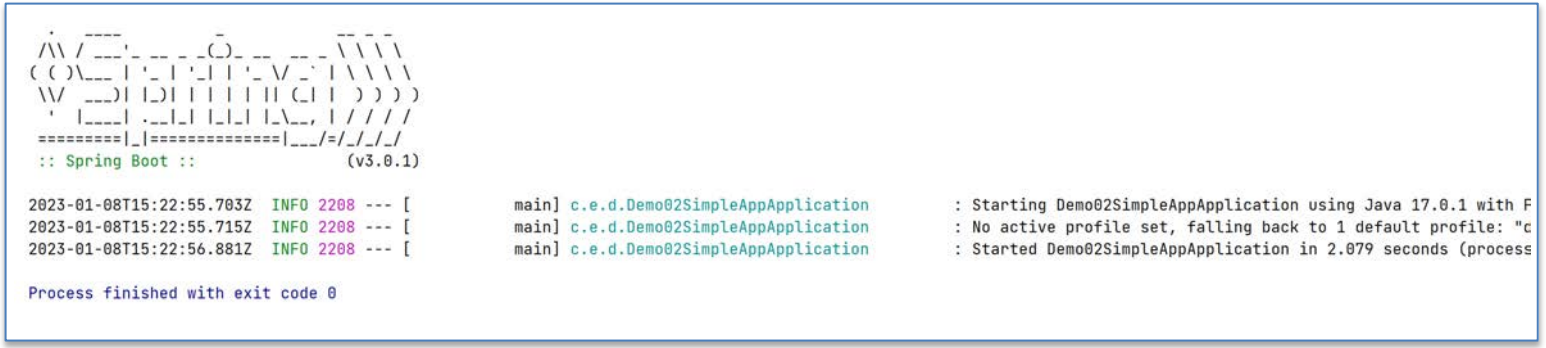
- If you don't have Maven installed separately on your machine, you can run the following command instead:

```
mvnw spring-boot:run
```

- It's also possible to run the application directly in IntelliJ
  - Right-click in the main class, and click Run

Viewing the Application Output

- This is the application output
  - Displays a "Spring Boot" banner
  - The app terminates immediately, because it's so simple 😊



Summary

- Creating a Spring Boot app in IntelliJ
- Understanding the application

Creating a Spring Boot Web App

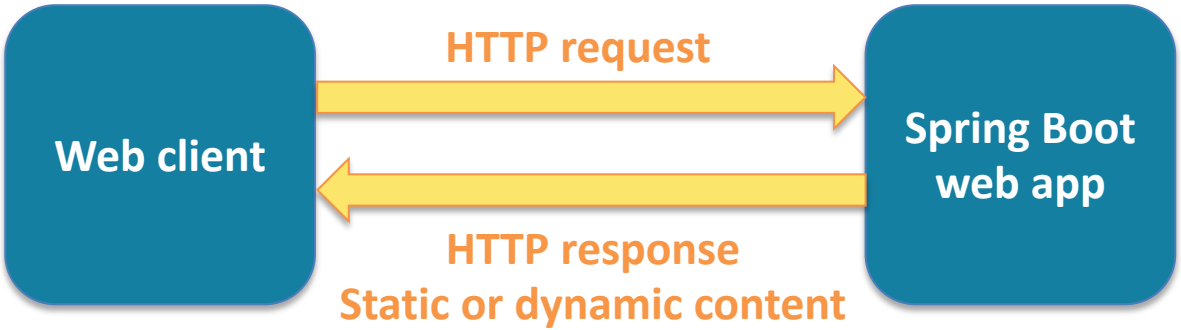
1. Creating a web app project in IntelliJ
2. Understanding the web app
3. Defining application properties

1. Creating a Web App Project in IntelliJ

- Overview
- Creating a Spring Boot web project
- Specifying project dependencies

Overview

- Spring Boot applications are typically "web apps"
  - Listen for HTTP requests from web client (e.g. a browser)
  - Return static or dynamic content



- We'll see how to return **static content** for now
  - Later we'll see how to return dynamic content, via REST services

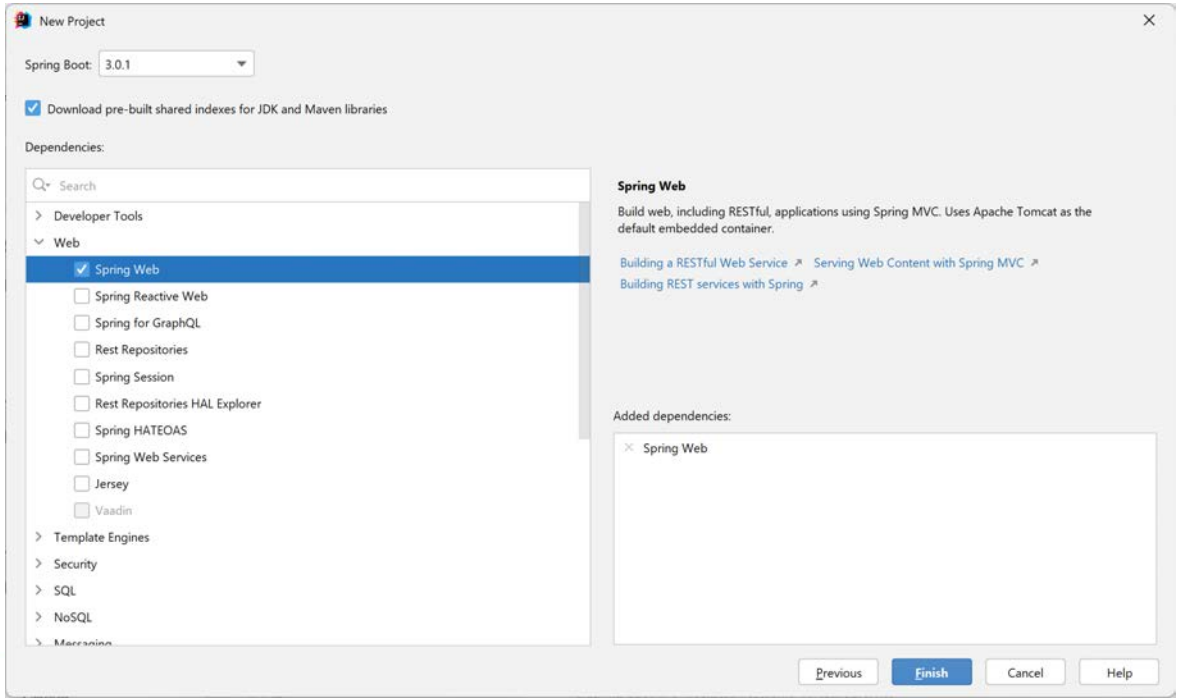
Creating a Spring Boot Web Project

- Start IntelliJ, click New Project, and select Spring Initializr

- Specify project info as follows:
  - Enter a suitable project name and location
  - Language - Java
  - Type - Maven
  - Enter a suitable group ID, artifact ID, and package name
  - Project SDK - Java 17
  - Java version - 17
  - Packaging - Jar
  - Then click Next

Specifying Project Dependencies

- In the next window you can add dependencies to your project
  - Expand **Web** and select **Spring Web**

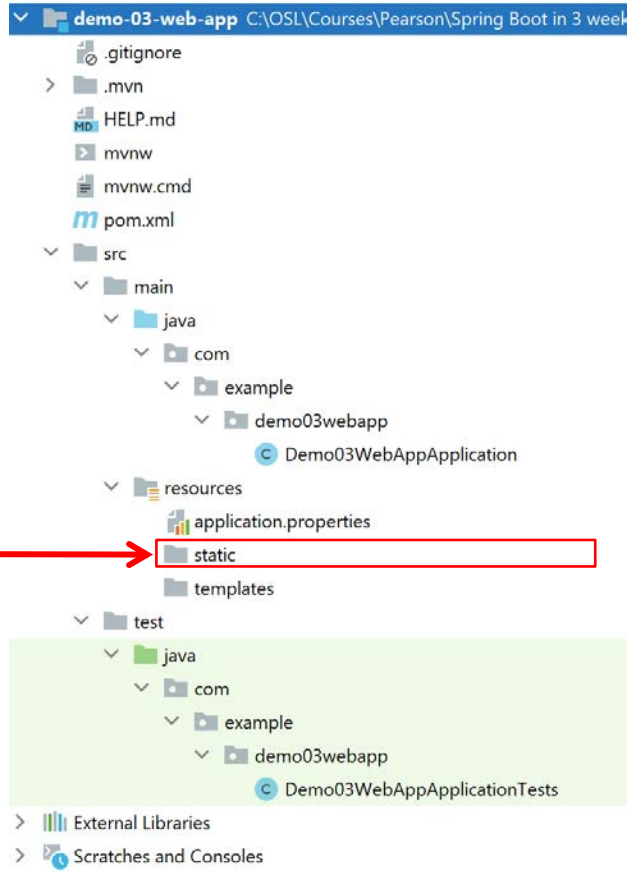


2. Understanding the Web Application

- Understanding the web application structure
- Understanding the Maven POM file
- Adding web content
- Running the application
- Pinging the application

Understanding the Web Application Structure

- The generated application is a regular Maven **web** project:



Put static web content here

Understanding the Maven POM File

- Here's the relevant section in the Maven POM file:

```
<project … >
    …

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

pom.xml

Adding Web Content

- You can add static web content in the following directory:
  - `src\main\resources\static`

- For example, add an `index.html` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home</title>
</head>
<body>
    Hello world!
</body>
</html>
```

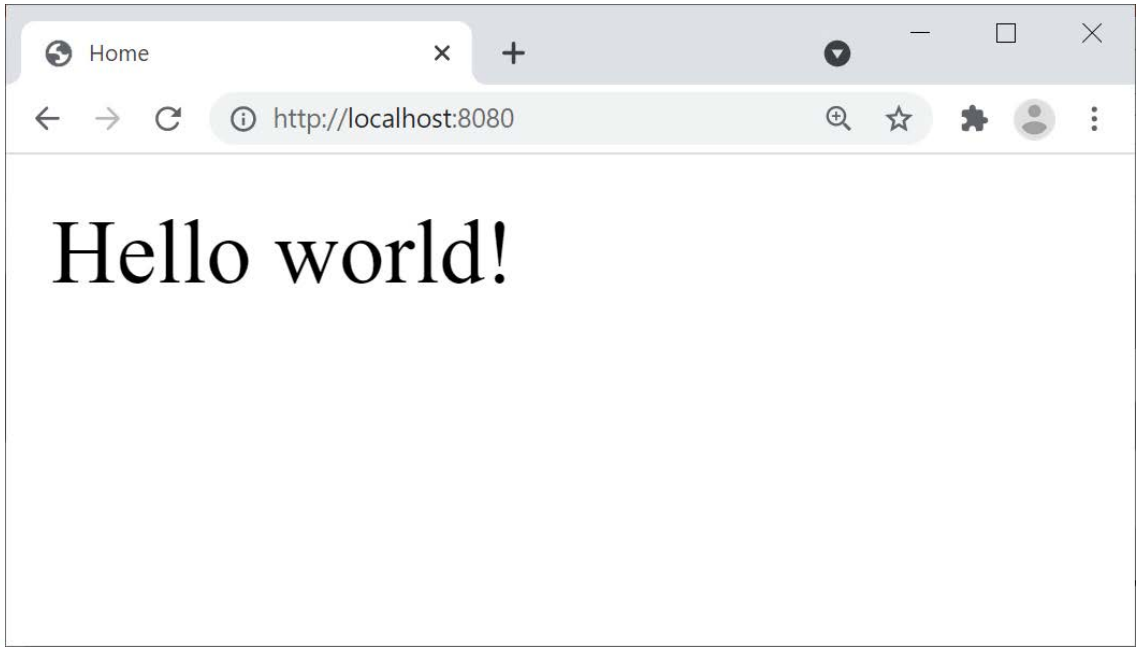
index.html

Running the Application

- To run the application:
  - Right-click the Java app file, then click Run
  - Compiles the code, bundles into a JAR, then runs the JAR
  - The application has an embedded Tomcat web server

Pinging the Application

- Open a browser and go to http://localhost:8080
  - Renders `index.html` (a standard welcome page in Java web)



3. Defining Application Properties

- Overview of application properties
- Editing application properties
- Restarting the application

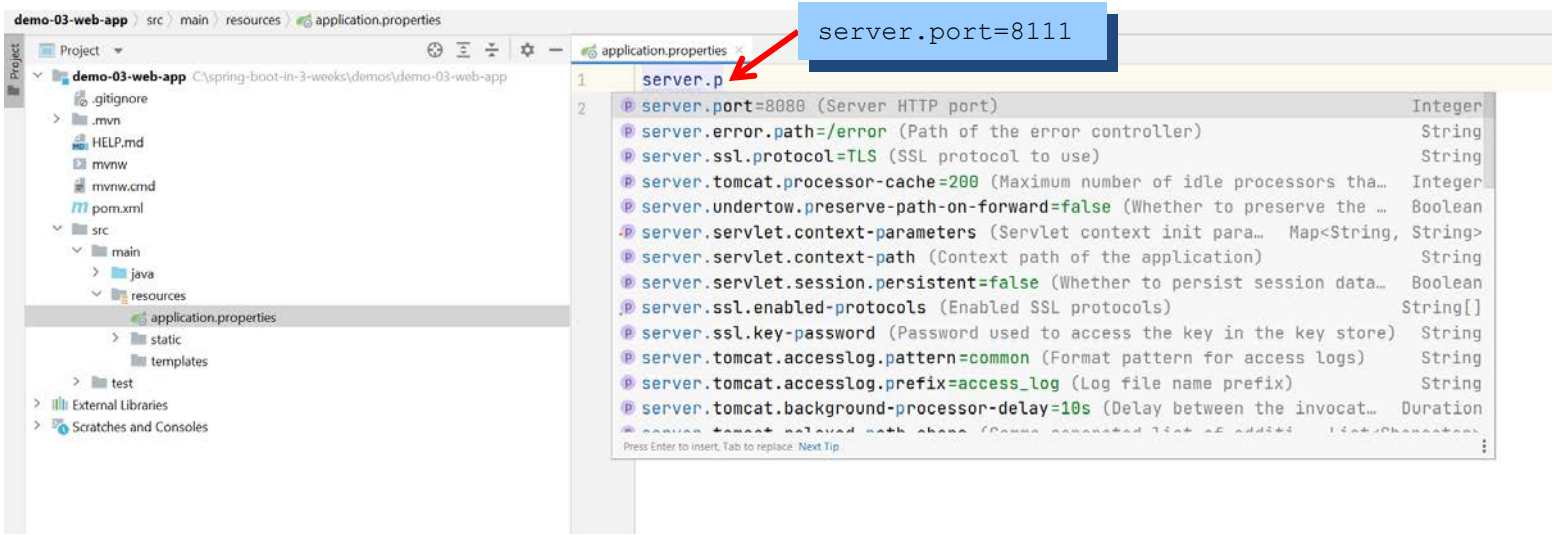
Overview of Application Properties

- Spring Boot applications have a standard text file named `application.properties`
  - Very important!
  - The recommended place to set application properties
  - i.e. name=value pairs

- You can also use YAML if you like
  - YAML = "YAML Ain't Markup Language"
  - More on YAML files later…

Editing Application Properties

- To help you edit `application.properties`, IntelliJ provides a Spring Properties Editor tool
  - Provides nice content assistance and error checking

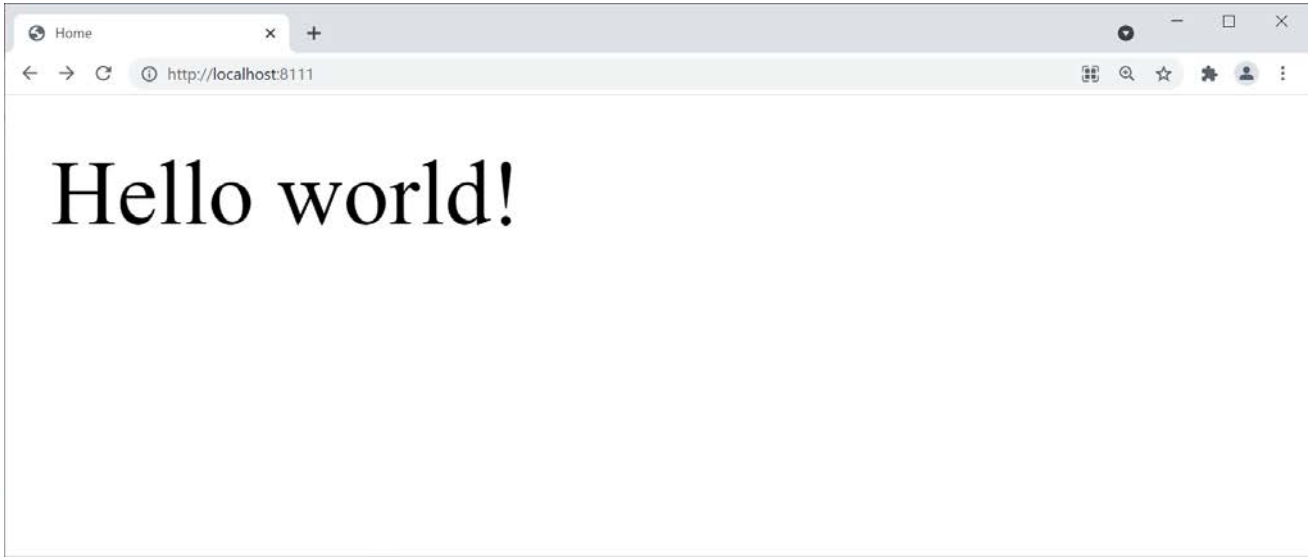


Restarting the Application

- Restart the application, and verify Tomcat starts on the new port number, 8111

```
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8111 (http) with context path ''
```

- Ping the Web server using the new port number, 8111



Summary

- Creating a web app project in IntelliJ
- Understanding the web app
- Defining application properties

Beans and Dependency Injection

1. Components and beans
2. A closer look at components and beans
3. Dependency injection
4. A closer look at dependency injection

1. Components and Beans

- Overview of components
- Defining a component
- Component scanning in Spring Boot
- Accessing a bean

Demo package: demo.beans

Overview of Components

- In Spring, a **component** is:
  - A class that Spring will automatically instantiate

- To define a component in Spring, annotate a class with any of the following annotations:
  - `@Component`
  - `@Service`
  - `@Repository`
  - `@Controller/@RestController`

Defining a Component

- Here's an example of how to define a component:

```
import org.springframework.stereotype.Component;

@Component
public class MyComponent {
    …
}
```

MyComponent.java

- Spring will automatically create an instance of this class
  - The instance is known as a "bean"

Component Scanning in Spring Boot

- When a Spring Boot app starts, it scans for component classes
  - It looks in the application class package, plus sub-packages

- You can tell Spring Boot to look elsewhere if necessary:

```
@SpringBootApplication( scanBasePackages={"mypackage1", "mypackage2"} )
public class Application {
    …
}
```

Accessing a Bean

- When a Spring Boot application starts up, it creates beans and stores them in the "application context"

- You can access beans in the application context as follows:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(Application.class, args);
        MyComponent bean = ctx.getBean(MyComponent.class);
        System.out.println(bean);
    }
}
```

Application.java

2. A Closer Look at Components and Beans

- Specifying a name for a component
- Understanding singleton scope
- Getting a singleton-scope bean
- Lazily instantiating a singleton bean
- Defining a different scope
- Getting prototype-scope beans

Specifying a Name for a Component

- Every bean has a name
  - By default, it's the name of the component class (with the first letter in lowercase)

- You can specify a different name for the bean as follows
  - When Spring creates a bean, it will be named `myCoolBean`

```
import org.springframework.stereotype.Component;

@Component("myCoolBean")
public class SomeComponent {
    …
}
```

Understanding Singleton Scope

- By default, Spring creates a single bean instance
  - i.e., the default scope is `"singleton"`

- You can annotate with `@Scope("singleton")` if you want to be explicit:

```
@Component
public class MySingletonComponent { … }
```

Equivalent

```
@Component
@Scope("singleton")
public class MySingletonComponent { … }
```

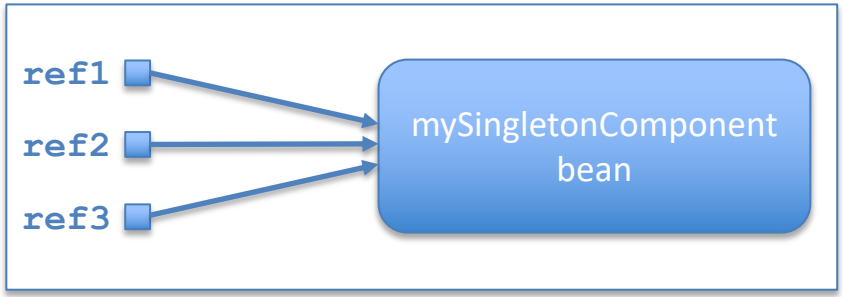
Getting a Singleton-Scope Bean

- Singleton beans are created at application start-up
  - For each call to `getBean()`, you get the same bean

```
ApplicationContext ctx = SpringApplication.run(Application.class, args);

MySingletonComponent ref1 = ctx.getBean(MySingletonComponent.class);
MySingletonComponent ref2 = ctx.getBean(MySingletonComponent.class);
MySingletonComponent ref3 = ctx.getBean(MySingletonComponent.class);
```

Application.java



Lazily Instantiating a Singleton Bean

- You can tell Spring to lazily instantiate a singleton bean
  - Annotate the component class with `@Lazy`

```
@Component
@Lazy
public class MySingletonComponent {
    …
}
```

- Avoids creating beans until needed
  - Speeds start-up time

Defining a Different Scope

- You can use `@Scope` to specify the scope for a bean:

```
@Component
@Scope("prototype")
public class MyPrototypeComponent { … }
```

MyPrototypeComponent.java

- There are several scopes available:
  - `"prototype"`
  - `"request"`
  - `"session"`
  - `"application"`

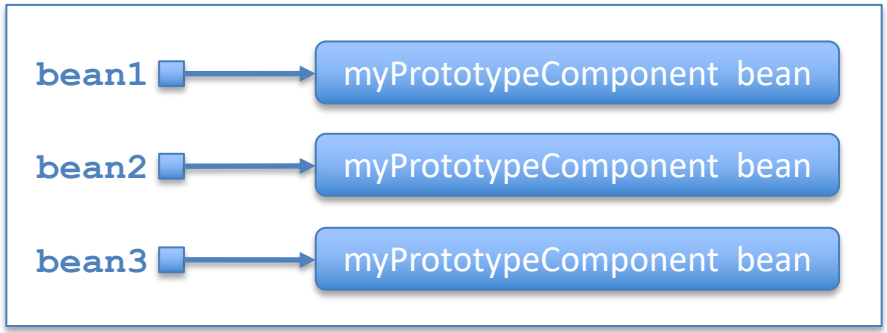
Getting Prototype-Scope Beans

- Consider this example of getting prototype beans
  - For each call to `getBean()`, Spring creates a new bean

```
ApplicationContext ctx = SpringApplication.run(Application.class, args);

MyPrototypeComponent bean1 = ctx.getBean(MyPrototypeComponent.class);
MyPrototypeComponent bean2 = ctx.getBean(MyPrototypeComponent.class);
MyPrototypeComponent bean3 = ctx.getBean(MyPrototypeComponent.class);
```

Application.java



3. Dependency Injection

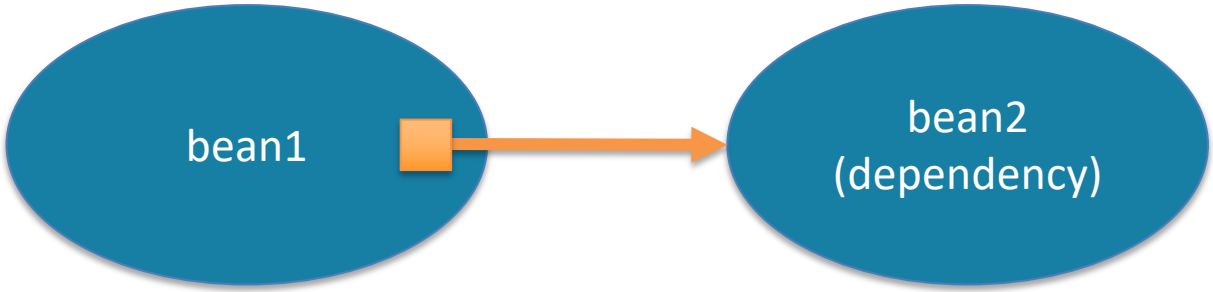
- Overview of dependency injection
- Injecting dependencies into fields
- Injecting dependencies into a constructor
- Fine-tuning autowiring

Demo package: demo.di

Overview of Dependency Injection

- Dependency Injection (DI) is a key Spring concept
  - Use configuration to describe dependencies between components

- Spring automatically injects dependencies into beans
  - This is known as "autowiring"



Injecting Dependencies into Fields

- If a bean has dependencies...
  - You can inject via `@Autowired`
- You can use `@Autowired` on a field
  - Spring injects a bean of the specified type into the field

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private BankRepository repository;
    …
}
```

BankServiceImpl.java

Injecting Dependencies into a Constructor

- You can also use `@Autowired` on a constructor
  - Spring will inject beans into all constructor parameters

```
@Service
public class BankServiceImpl implements BankService {

    private BankRepository repository;

    @Autowired
    public BankServiceImpl(BankRepository repository) {
        this.repository = repository;
    }
    …
}
```

BankServiceImpl.java

- Note: If a component only has one constructor, you can omit `@Autowired` (Spring autowires ctor params automatically)

Fine-Tuning Autowiring

- You can specify which bean instance to inject
  - Use `@Qualifier` to specify the bean name you want

```
@Autowired
@Qualifier("primaryRepository")
private BankRepository repository;
```

- You can mark an `@Autowired` member as optional
  - Set `required=false`

```
@Autowired(required=false)
private BankRepository repository;
```

4. A Closer Look at Dependency Injection

- Autowiring a collection
- Autowiring a map
- Injecting values into beans
- Specifying values in application properties
- Aside: Common application properties

Autowiring a Collection

- You can autowire a `Collection<T>`
  - Spring injects a collection of all the beans of type `T`

- Example
  - Autowire a collection of all beans that implement the `BankRepository` interface

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private Collection<BankRepository> repositories;
    …
}
```

Autowiring a Map

- You can also autowire a `Map<String,T>`
  - Spring injects a map indicating all beans of type `T`
  - Keys are bean names, values are bean instances

- Example
  - Autowire `BankRepository` names/beans

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private Map<String, BankRepository> repositoriesMap;
    …
}
```

Injecting Values into Beans

- You can inject values into beans, via `@Value`
  - Use `$` to inject an application property value
  - Use `#` to inject a general value via Spring Expression Language

```
import org.springframework.beans.factory.annotation.Value;
…

@Component
public class MyBeanWithValues {

    @Value("${name}")           // Inject value of "name" application property.
    private String name;

    @Value("#{ 5 * 7.5 }")      // Inject general Java value via SpEL.
    private double workingWeek;
    …
}
```

MyBeanWithValues.java

Specifying Values in Application Properties

- You can define values in the application properties file

```
name=John Smith
```

application.properties

- Here's how to access the bean in the main code

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {

        ApplicationContext ctx = SpringApplication.run(Application.class, args);
        …
        MyBeanWithValues beanWithValues = ctx.getBean(MyBeanWithValues.class);
        System.out.println(beanWithValues);
    }
}
```

Application.java

Aside: Common Application Properties

- Spring Boot defines lots of common application properties by default - you can see the full list here:
  - https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

- You can override any of these properties in your code
  - In `application.properties` or `application.yml`

Summary

- Components and beans
- A closer look at components and beans
- Dependency injection
- A closer look at dependency injection

Exercise

- Define another component class that implements the `BankRepository` interface
  - Name the class `BankRepositoryCheckedImpl`
  - The class rejects transactions greater than a threshold amount
  - Specify the threshold in `application.properties`
- Modify `BankServiceImpl` to make use of this new component class
- Write some code in `main()` to test the new functionality

Injection Techniques

1. Using Spring Expression Language (SpEL)
2. Working with command-line arguments

1. Using Spring Expression Language (SpEL)

- Overview of SpEL
- Simple SpEL example
- SpEL scalar expressions
- Using SpEL for collections
- Using SpEL for parameters

Demo package: demo.spel

Overview of SpEL

- Spring Expression Language (SpEL) is a Java-like syntax that you can use in various places in Spring:
  - In beans, via `@Value` annotations on fields
  - On parameters in autowired methods
  - Within XML configuration files
  - Etc.

Simple SpEL Example

- Here's a simple SpEL example in a Spring bean:

```
@Component
public class SpelBean {

    @Value("#{ 5 * 7.5 }")
    private double workingWeek;
    …
}
```

SpelBean.java

- Literals you can use in SpEL:
  - Strings enclosed in single quotes
  - Dates, numbers, booleans
  - null

SpEL Scalar Expressions

- You can create an object in a SpEL expression:

```
@Value("#{ new java.util.Date() }")
private Date timestamp;
```
SpelBean.java

- You can call a static method, using `T` to denote a type:

```
@Value("#{ T(java.lang.Math).random() * 100.0 }")
private int luckyNumber;
```
SpelBean.java

Using SpEL for Collections (1 of 2)

- SpEL can access items in arrays, collections, and maps:

```
@Value("#{ info.cities[9] }")
private String city;

@Value("#{ info.currencies['UK'] }")
private String currency;
```

SpelBean.java

```
@Component
public class Info {
    public List<String> cities() {…}
    public Map<String, String> currencies() {…}
}
```

Info.java

Using SpEL for Collections (2 of 2)

- SpEL has operators for processing collection items:

```
@Value("#{ info.cities.?[startsWith('B')] }")
private List<String> allBCities;

@Value("#{ info.cities.^[startsWith('B')] }")
private String firstBCity;

@Value("#{ info.cities.$[startsWith('B')] }")
private String lastBCity;

@Value("#{ info.cities.![toUpperCase()] }")
private List<String> upperCities;
```

SpelBean.java

Using SpEL for Parameters

- You can use SpEL for autowired method parameters:

```
@Component
public class SpelBean {

    @Autowired
    public void setUserName(@Value("#{systemProperties['user.name'] }") String n) {
        …
    }
```

SpelBean.java

2. Working with Command-Line Arguments

- Overview of command-line arguments
- Accessing command-line arguments
- Two types of command-line arguments
- Passing command-line arguments
- Accessing command-line arguments

Demo package: demo.commandlineargs

Overview of Command-Line Arguments

- Here's a reminder of how to "run" a Spring Boot app:

```
public static void main(String[] args) {

    ApplicationContext ctx = SpringApplication.run(Application.class, args);
    …
}
```

- Note we've passed `args` into `SpringApplication.run()`
  - This makes the command-line args available to your components

Accessing Command-Line Arguments

- You can autowire command-line args into a component:

```
@Component
public class MyBeanWithArgs {

    @Autowired
    public MyBeanWithArgs(ApplicationArguments args) {
        // You can access command-line arguments here…
    }
    …
}
```

MyBeanWithArgs.java

Two Types of Command-Line Arguments

- Spring Boot supports two types of command-line arguments:
- Option arguments, prefixed by −−

```
--target=windows --target=macOS --db=h2
```

- Non-option arguments

```
norway oslo krone 42
```

Passing Command-Line Arguments

- To pass command-line arguments using IntelliJ:
  - Click Run | Edit Configurations
  - Choose the module and class to run
  - Enter program arguments
  - Then run the configuration

Accessing Command-Line Arguments

- The `ApplicationArguments` class has various methods for accessing command-line arguments:
  - `getSourceArgs()`
  - `getOptionNames()`
  - `getOptionValues(optionName)`
  - `getNonOptionArgs()`

- Example:
  - See `MyBeanWithArgs.java`

Summary

- Using Spring Expression Language (SpEL)
- Working with command-line arguments

Exercise

- Define a component class named `Timestamp` with two methods
  - `creationDate()` – returns local creation date
  - `creationTime()` – returns local creation time
- Inject both these values into the existing `SpelBean` component class
- Add a method to `SpelBean` to display the date, time, or both depending on a command-line argument:
  - `displayTimestampMode=date|time|both`

Configuration Classes

1. Defining a config class and bean methods
2. Locating config classes and bean methods
3. Configuration techniques
4. Configuring bean dependencies

1. Defining a Config Class and Bean Methods

- Overview of configuration classes
- Defining a simple configuration class
- Accessing a bean

Demo package: demo.simpleconfiguration

Overview of Configuration Classes

- A configuration class is a special "factory" class in Spring Boot
  - Creates and initializes bean objects

- How to define a configuration class:
  - Annotate class with `@Configuration`
  - Annotate methods with `@Bean` and create/return objects

Defining a Simple Configuration Class

- Here's a simple configuration class:

```
@Configuration
public class MyConfig {

    @Bean
    public MyBean myBean() {
        MyBean b = new MyBean();
        b.setField1(42);
        b.setField2("wibble");
        return b;
    }
    …
```

MyConfig.java

- This example creates a bean:
  - Type of bean is `MyBean`
  - Name of bean is `"myBean"`

Accessing a Bean

- You can access beans as normal:

```
ApplicationContext ctx = SpringApplication.run(Application.class, args);

MyBean bean = ctx.getBean("myBean", MyBean.class);
System.out.println(bean);
```

Application.java

- You can also autowire beans as normal:

```
@Component
public class SomeComponent {

    @Autowired
    MyBean bean;
    …
}
```

SomeComponent.java

2. Locating Config Classes and Bean Methods

- Location of configuration classes
- Specifying different configuration locations
- Defining beans in the "application" class

Demo package: demo.configurationlocation

Location of Configuration Classes

- Configuration classes are special kinds of "components"

- When a Spring Boot application starts…
  - It scans for components and configuration classes
  - It "application" class package, plus sub-packages

Specifying Different Configuration Locations

- You can tell Spring Boot to look in alternative packages to find components and configuration classes

```
@SpringBootApplication( scanBasePackages={"mypackage1", "mypackage2"} )
public class Application {
    …
}
```

- See the following packages in the demo app:
  - `demo.configurationlocation.main` - app class
  - `demo.configurationlocation.config` - config class

Defining Beans in the "Application" Class (1 of 2)

- The `@SpringBootApplication` annotation is equivalent to:
  - `@Configuration`
  - `@EnableAutoConfiguration`
  - `@ComponentScan`

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

- This means the application class is also a "configuration" class
  - You can define `@Bean` methods in your application class
  - See example on next slide…

Defining Beans in the "Application" Class (2 of 2)

```
@SpringBootApplication(scanBasePackages="demo.configurationlocation.config")
public class Application {

    @Bean
    public LocalDateTime timestamp1() {
        return LocalDateTime.of(1997, 7, 2, 1, 5, 30);
    }

    @Bean
    public LocalDateTime timestamp2() {
        return LocalDateTime.of(1997, 7, 2, 1, 20, 0);
    }
    …
}
```

Application.java

3. Configuration Techniques

- Customizing bean names
- Looking-up named beans
- Lazily instantiating a singleton bean
- Setting the scope of a bean

Demo package: demo.techniques

Customizing Bean Names

- By default the bean name is the same as method name
  - You can specify different bean name(s), if you like

```
@Configuration
public class MyConfig {

    @Bean(name="cool-bean")
    public MyBean bean1() { return new MyBean(1111, aUUID); }

    @Bean(name = {"subsystemA-bean", "subsystemB-bean", "subsystemC-bean"})
    public MyBean bean2() { return new MyBean(2222, aUUID); }
    …
}
```

MyConfig.java

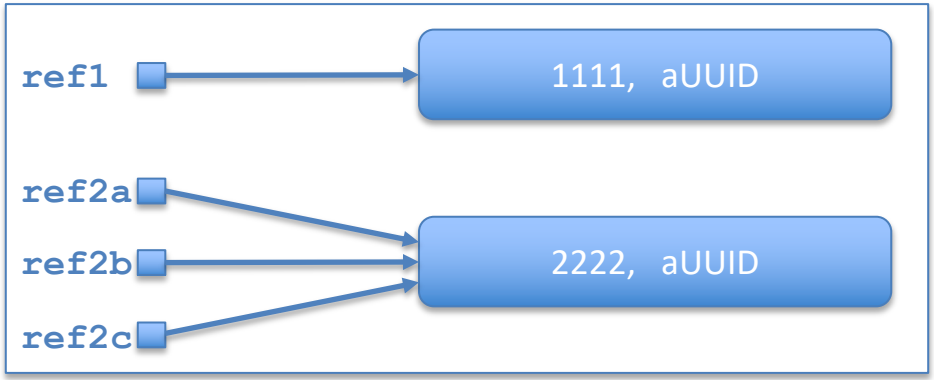
Looking-Up Named Beans

- Call `getBean()` and specify the bean name you want:

```
// Lookup 1st bean via its name.
MyBean ref1 = ctx.getBean("cool-bean", MyBean.class);

// Lookup 2nd bean via its various aliases.
MyBean ref2a = ctx.getBean("subsystemA-bean", MyBean.class);
MyBean ref2b = ctx.getBean("subsystemB-bean", MyBean.class);
MyBean ref2c = ctx.getBean("subsystemC-bean", MyBean.class);
```

Application.java



Lazily Instantiating a Singleton Bean

- You can set a bean to be lazily instantiated as follows:

```
@Configuration
public class MyConfig {

    @Bean(name="lazy-bean")
    @Lazy
    public MyBean bean3() { return new MyBean(3333, aUUID); }
    …
}
```

MyConfig.java

- Spring Boot will instantiate the bean "just in time"

Setting the Scope of a Bean

- You can set the scope of a bean as follows:

```
@Configuration
public class MyConfig {

    @Bean(name="proto-bean")
    @Scope("prototype")
    public MyBean bean4() { return new MyBean(4444, aUUID); }
    …
}
```

MyConfig.java

- Spring Boot will instantiate a new bean every time you autowire or call `ctx.getBean()`

4. Configuring Bean Dependencies

- Overview
- Configuring dependencies - technique 1
- Configuring dependencies - technique 2

Demo package: demo.dependencies

Overview

- Consider the following Java classes:

```
public class TransactionManager {

  DataSource dataSource;

  public void setDataSource(DataSource ds) {
    this.dataSource = ds;
  }
  …
}
```

TransactionManager.java

```
public class DataSource {

  private String connectionString;
  private int maxPoolSize;
    …
}
```

DataSource.java

- Note:
  - The `TransactionManager` references a `DataSource`

Configuring Dependencies - Technique 1

- You can configure dependencies as follows:

```
@Configuration
public class MyConfig {

    @Bean
    public DataSource dataSource() {
        DataSource ds = new DataSource();
        …
        return ds;
    }

    @Bean
    public TransactionManager transactionManager1() {
        TransactionManager txMgr = new TransactionManager();
        txMgr.setDataSource(dataSource());
        return txMgr;
    }
    …
}
```

MyConfig.java

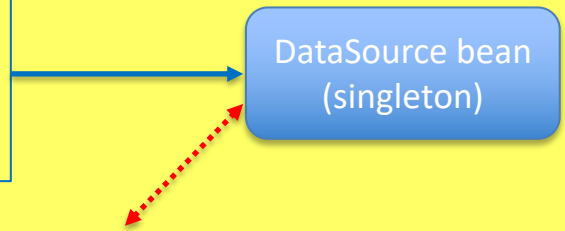
Configuring Dependencies - Technique 2

- Here's another way to configure dependencies:

```
@Configuration
public class MyConfig {

    @Bean
    public DataSource dataSource() {
        DataSource ds = new DataSource();
        …
        return ds;
    }

    @Bean
    public TransactionManager transactionManager3(DataSource ds) {
        TransactionManager txMgr = new TransactionManager();
        txMgr.setDataSource(ds);
        return txMgr;
    }
    …
}
```



MyConfig.java

Summary

- Defining a config class and bean methods
- Locating config classes and bean methods
- Configuration techniques
- Configuring bean dependencies

Exercise

- Define a Java class named `Transcript` as follows:
  - `log(String)` method, adds a message to a log transcript
  - `transcriptSize` property, max number of messages
  - `cyclic` property, indicates whether to clear log if full
- Create a `Transcript` bean and initialize as follows:
  - `transcriptSize = 5`
  - `cyclic = true`
- Inject the `Transcript` bean into another bean and log some messages

Spring Boot Techniques

1. Setting app properties at the command line
2. Specifying which properties file to use
3. Defining YAML properties files
4. Using Spring profiles

1. Setting App Properties at the Command Line

- Recap of application properties
- Source of external configuration
- Setting properties at the command line

Demo package: demo.commandlineproperties

Recap of Application Properties

- A Spring Boot application can define properties in an `application.properties` file:

```
name=John Smith
```

application.properties

- You can inject properties via `@Value("${propName}")`

```
@Component
public class MyBean1 {

    @Value("${name}")
    private String name;
    …
}
```

MyBean1.java

Source of External Configuration

- Spring Boot lets you define application properties in many places, such as:
  - Command-line arguments
  - Environment variable `SPRING_APPLICATION_JSON`
  - Operating system environment variables
  - Application properties outside your JAR
  - Application properties inside your JAR

Setting Properties at the Command Line

- If you define command-line args that start with `--`
  - Spring Boot converts them into application properties

- E.g., set the `name` property via a command-line arg:

```
--name="Mary Jones"
```

- Let's see an example in IntelliJ:
  - Edit configurations
  - Specify a command-line arg, as shown above
  - Run the configuration

2. Specifying which Properties File to Use

- Location of properties files
- Specifying a different properties file

Demo package: demo.differentpropertiesfile

Location of Properties Files

- `SpringApplication` looks in the following places to find properties files (highest priority first):
  - `/config` subdirectory of your Java app directory
  - Your Java app directory
  - `/config` package on classpath
  - Root package on classpath

Specifying a Different Properties File (1 of 2)

- You can tell Spring to use a different properties file:

```
@SpringBootApplication
public class Application {

    private static void demo2(String[] args) {

        System.setProperty("spring.config.name", "app2");
        ApplicationContext ctx = SpringApplication.run(Application.class, args);

        …
```

Application.java

```
name=Bill Jones
```

app2.properties

- Alternatively, you can set the `SPRING_CONFIG_NAME` environment variable

Specifying a Different Properties File (2 of 2)

- You can also use a command-line argument to specify which application properties file to use:

```
--spring.config.name=app2
```

- This enables you to specify a properties file as part of your overall CI/CD process
  - E.g. in a Jenkins build script

3. Defining YAML Properties Files

- Overview of YAML files
- Using YAML properties in beans - technique 1
- Using YAML properties in beans - technique 2

Demo package: demo.yamlpropertiesfiles

Overview of YAML Files

- Spring Boot supports YAML as an alternative format for defining application properties:

```
contact:
    tel: 555-111-2222
    email: contact@mydomain.com
    web: http://mydomain.com
```

app3.yml

- YAML is convenient for specifying hierarchical config data

Using YAML Properties in Beans - Technique 1

- Here's one way to use YAML properties in a bean:

```
@Component
public class MyBean3a {

    @Value("${contact.tel}")
    private String tel;

    @Value("${contact.email}")
    private String email;

    @Value("${contact.web}")
    private String web;
    …
}
```

MyBean3a.java

Using YAML Properties in Beans - Technique

- Here's another way to use YAML properties in a bean:

```
@Component
@ConfigurationProperties(prefix="contact")
public class MyBean3b {

    private String tel;
    private String email;
    private String web;
    …
    // Plus getters and setters - these are essential!
 }
```

MyBean3b.java

- You also need this dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
</dependency>
```

pom.xml

4. Using Spring Profiles

- Overview
- Defining profile-specific components
- Defining profile-specific properties
- Setting the active profile

Demo package: demo.profiles

Overview

- Spring profiles provide a way to segregate parts of your application configuration
  - So, configuration is only available in certain environments

- For example:
  - "development" profile
  - "production" profile

Defining Profile-Specific Components

- You can annotate component classes with `@Profile`:

```
public interface MyBean4 {}
```

```
@Component
@Profile("development")
public class MyBean4Dev implements MyBean4 {

    @Override
    public String toString() { return "Hello from MyBean4Dev"; }
}
```

MyBean4Dev.java

```
@Component
@Profile("production")
public class MyBean4Prod implements MyBean4 {

    @Override
    public String toString() { return "Hello from MyBean4Prod"; }
}
```

MyBean4Prod.java

Defining Profile-Specific Properties

- You can also define profile-specific properties:

```
apiserver:
  address: 192.168.1.100
  port: 8080
---
spring:
  config:
    activate:
      on-profile: development
apiserver:
  address: 127.0.0.1
---
spring:
  config:
    activate:
      on-profile: production
apiserver:
  address: 192.168.1.120
```

Default values for properties

Properties for "development" profile

Properties for "production" profile

app4.yml

Setting the Active Profile

- You must tell Spring what is the active profile
  - Set the `spring.profiles.active` property

- To set the active profile via application properties:

```
spring.profiles.active=development                    app4.properties
```

- To set it at the command-line:

```
--spring.profiles.active=production
```

Summary

- Setting app properties at the command line
- Specifying which properties file to use
- Defining YAML properties files
- Using Spring profiles

Exercise

- Use profiles to define geography-specific properties:

Property	Value if "UK" profile	Value if "US" profile
txfmt.currency	GBP	USD
txfmt.dtformat	dd-MM-yyyy HH:mm:ss	MM-dd-yyyy HH:mm:ss

- Inject these values into a component class named `FinancialTransactionLogger`
  - Implement a `log()` method to output a formatted currency and timestamp - to format the timestamp, use `DateTimeFormatter.ofPattern(dtformat)`
- Set `spring.profiles.active` (hint, you can set comma-separated profiles)