

Section 70: Spring Boot - Overview

0 / 7 | 52min

The Problem

- Building a Spring application is really HARD!!!

Q: What Maven archetype to use?

And that's
JUST the basics
for getting started

Q: Which Maven dependencies do I need?

Q: How do I set up configuration (xml or Java)?

Q: How do I install the server? (Tomcat, JBoss etc...)

The Problem

- Tons of configuration

Very error-prone

Easy to make
a simple mistake

```
<!-- Step 1: Configure Spring MVC Dispatcher Servlet -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<!-- Step 2: Set up URL mapping for Spring MVC Dispatcher Servlet -->
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- Step 3: Add support for component scanning -->
<context:component-scan base-package="com.luv2code.springsecurity.demo">
<!-- Step 4: Add support for conversion, formatting and localization -->
<mvc:annotation-driven>
<!-- Step 5: Define Spring MVC view resolver -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp" />

```

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages="com.luv2code.springsecurity.demo")
public class DemoAppConfig {

    // define a bean for ViewResolver

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/view/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

Spring Boot is the solution:

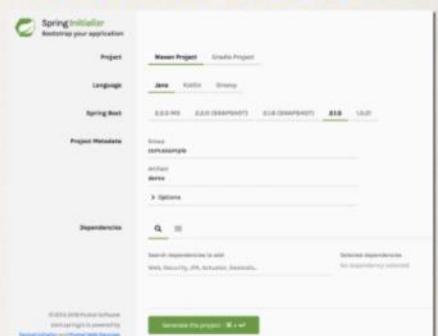
- Make it easier to get started with spring development.
- Minimize the amount of manual configuration
- Perform auto-configuration based on props files and JAR Classpath.
- Help to resolve dependency conflicts (Maven or Gradle)
- Provide an embedded HTTP server so you can get started quickly.
 - Out of box we can make use of Tomcat, Jetty, Undertow,...

Spring boot provides the spring initializer, which helps to quickly create a starter Spring Project.

Spring Initializr

- Quickly create a starter Spring project
- Select your dependencies
- Creates a Maven/Gradle project
- Import the project into your IDE
 - Eclipse, IntelliJ, NetBeans etc ...

<http://start.spring.io>



Spring Boot Embedded Server

- Provide an embedded HTTP server so you can get started quickly
 - Tomcat, Jetty, Undertow, ...
- No need to install a server separately



Running Spring Boot Apps

- Spring Boot apps can be run standalone (includes embedded server)
- Run the Spring Boot app from the IDE or command-line



Deploying Spring Boot Apps

- Spring Boot apps can also be deployed in the traditional way
- Deploy **WAR file** to an external server: Tomcat, JBoss, WebSphere etc ...



➤ Spring Boot Initializer Demo

1. Configure our project at Spring Initializer website

Visit the website: <https://start.spring.io/>

2. Download the zip file
3. Unzip the file
4. Import Maven project into the IDE.

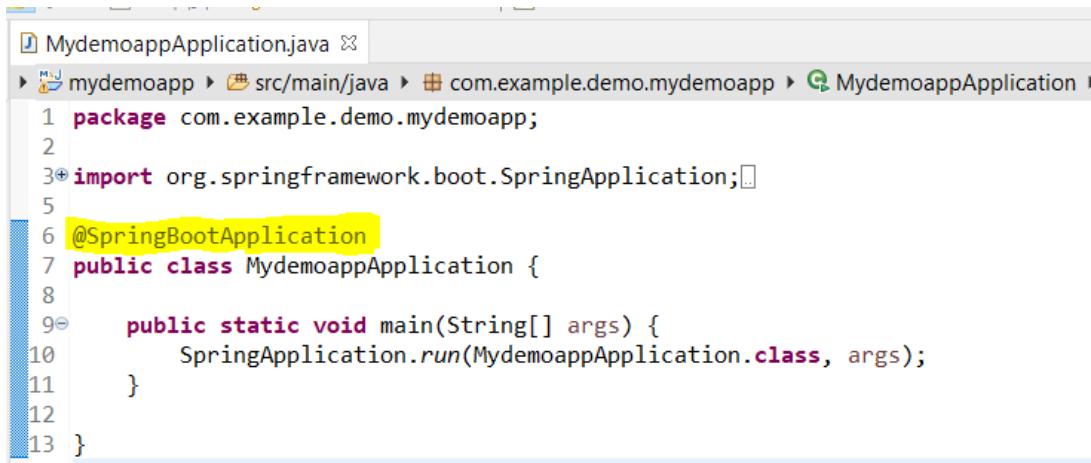
We will go to the file annotated with `@SpringBootApplication` annotation and run as **java application**.

Spring Boot FAQ #1

Q: Does Spring Boot replace Spring MVC, Spring REST etc ...?

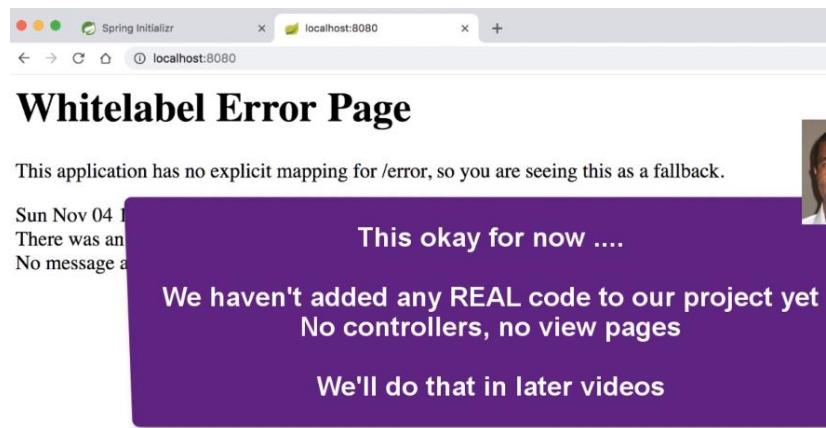
- No. Instead, Spring Boot actually uses those technologies





```
MydemoappApplication.java
mydemoapp > src/main/java > com.example.demo.mydemoapp > MydemoappApplication.java
1 package com.example.demo.mydemoapp;
2
3+ import org.springframework.boot.SpringApplication;[]
4
5
6 @SpringBootApplication
7 public class MydemoappApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(MydemoappApplication.class, args);
11     }
12
13 }
```

Output:



➤ Create a REST controller with Spring Boot



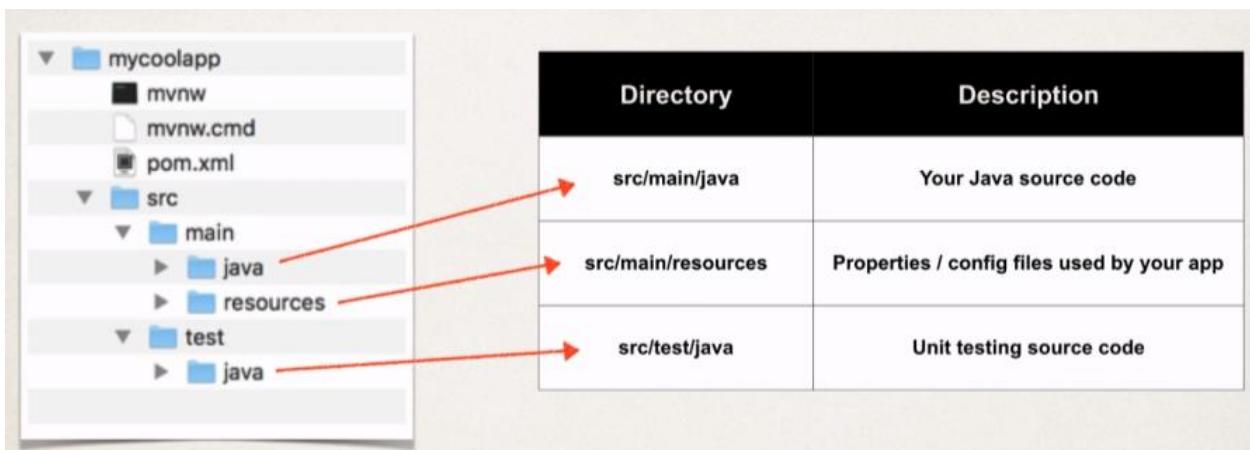
Output:

So, within no time we have successfully made our rest application and without performing any configuration.



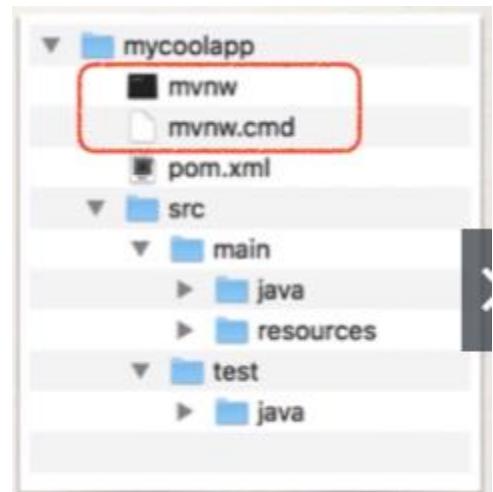
➤ Exploring Spring Boot Project Structure:

Spring boot make use of **Maven Standard Directory Structure**.



Maven Wrapper files:

- **mvnw** allows you to run a Maven Project.
- And there is no need to have Maven installed or present on your classpath.
- What they will do actually:
 - If correct version of Maven is NOT found on your computer
 - Then it will **Automatically Downloads** correct version and runs Maven.
- There are two files that are provided:
 - **mvnw.cmd** for MS windows.
 - **mvnw.sh** for linux/Mac
- If you already have Maven Installed previously.
- Then you can simply ignore/delete the **mvnw** files.



10. Spring Boot - Exploring the Spring Boot Project Structure - Part 1

Maven POM file

- **pom.xml** includes info that you entered

Spring Boot Starters
A collection of Maven dependencies (Compatible versions)

<groupId>com.luv2code.springboot.demo</groupId>
<dependency><groupId>org.springframework.boot</groupId><artifactId>spring-boot-starter-web</artifactId></dependency>

Save's the developer from having to list all of the individual dependencies

Also, makes sure you have compatible versions

framework, I root-starter, spring-web, spring-webmvc, hibernate-validator, tomcat, json, ...

Maven POM file

- Spring Boot Maven plugin

To package executable jar
or war archive

Can also easily run the app

The diagram shows a file tree for a Maven project named 'mycoolapp'. The structure includes a 'mvnw' script, a 'mvnw.cmd' batch file, a 'pom.xml' file, and source code directories 'src/main/java', 'src/main/resources', and 'src/test/java'. To the right of the file tree is the content of the 'pom.xml' file, which defines a Spring Boot Maven plugin configuration. Below the file tree, a red box contains the command 'mvn package'. To the right of the 'pom.xml' content, two terminal command boxes show how to build and run the application: '\$./mvnw package' and '\$./mvnw spring-boot:run'.

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

Can also just use:

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

\$./mvnw package

\$./mvnw spring-boot:run

If maven is already installed in our system, then we can just use the below two commands:

```
mvn package
```

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

Main Spring boot application class Created by Spring Initializer:

The screenshot shows the code for 'MycoolappApplication.java'. It includes imports for 'com.luv2code.springboot.demo.mycoolapp', 'org.springframework.boot.SpringApplication', and 'org.springframework.boot.autoconfigure.SpringBootApplication'. The class is annotated with '@SpringBootApplication'. A green callout box highlights the '@SpringBootApplication' annotation and lists its three benefits: Auto configuration, Component scanning, and Additional configuration.

```
File: MycoolappApplication.java

package com.luv2code.springboot.demo.mycoolapp;

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

@SpringBootApplication
public class MycoolappApplication {

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

Enables

Auto configuration
Component scanning
Additional configuration

Annotations

- `@SpringBootApplication` is composed of the following annotations:

Annotation	Description
<code>@EnableAutoConfiguration</code>	Enables Spring Boot's auto-configuration support
<code>@ComponentScan</code>	Enables component scanning of current package Also recursively scans sub-packages
<code>@Configuration</code>	Able to register extra beans with <code>@Bean</code> or import other configuration classes

File: MycoolappApplication.java

```
package com.luv2code.springboot.demo.mycoolapp;

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

@SpringBootApplication
public class MycoolappApplication {

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

Behind the scenes ...

Creates application context and registers all beans

Starts the embedded server Tomcat etc...

Bootstrap your Spring Boot application

More on Component Scanning

Best Practice

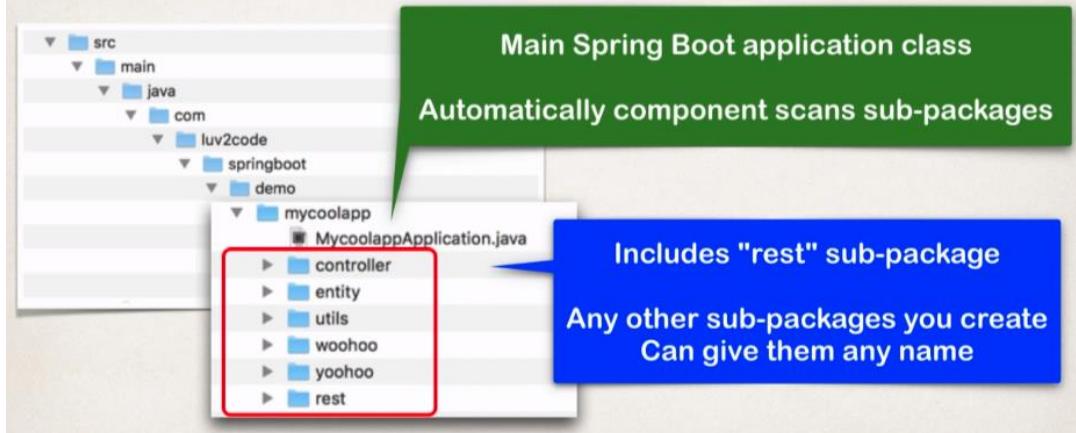
Place your main application class in the root package above your other packages

- This implicitly defines a base search package
 - Allows you to leverage default component scanning
 - No need to explicitly reference the base package name

Common pitfall in traditional Spring apps

Anyone use the wrong package name before in traditional Spring apps???

More on Component Scanning



- Default scanning is fine if everything is under
 - `com.luv2code.springboot.demo.mycoolapp`
- But what about my other packages?
 - `org.acme.iot.utils`
 - `edu.cmu.wean`

Explicitly list
base packages to scan

```
package com.luv2code.springboot.demo.mycoolapp;  
...  
@SpringBootApplication(  
    scanBasePackages={"com.luv2code.springboot.demo.mycoolapp",  
    "org.acme.iot.utils",  
    "edu.cmu.wean"})  
public class MycoolappApplication {  
    ...  
}
```

➤ Application Properties:

By default, Spring Boot will load properties from: `application.properties`



Can add Spring Boot properties
`server.port=8585`

Also add your own custom properties
`coach.name=Mickey Mouse`

How to read data from `Application.properties` file:

We can actually read this by making use of **injection**.

The diagram illustrates the mapping between `application.properties` file entries and Java code. On the left, the `application.properties` file contains three entries: `server.port=8484`, `coach.name=Mickey Mouse`, and `team.name=The Mouse Crew`. A red arrow points from the `team.name` entry to the corresponding `@Value("${team.name}")` annotation in the `FunRestController` class on the right. The `FunRestController` class also includes `@RestController` and `private String coachName;` annotations.

```
# configure server port
server.port=8484

# configure my props
coach.name=Mickey Mouse
team.name=The Mouse Crew

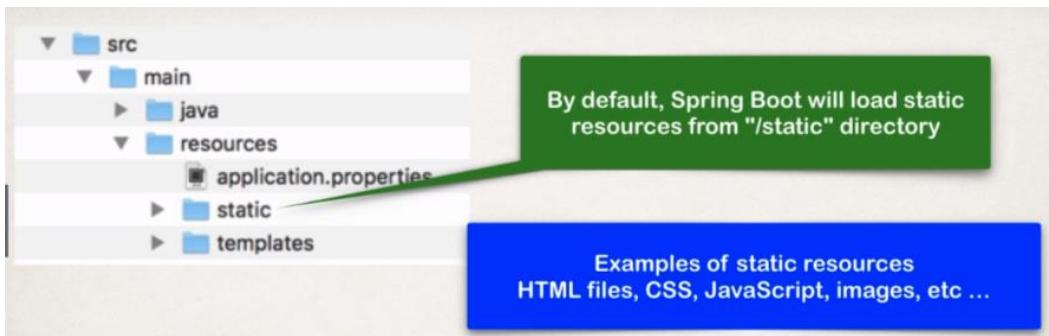
@RestController
public class FunRestController {

    @Value("${coach.name}")
    private String coachName;

    @Value("${team.name}")
    private String teamName;

    ...
}
```

➤ Static Content



WARNING:

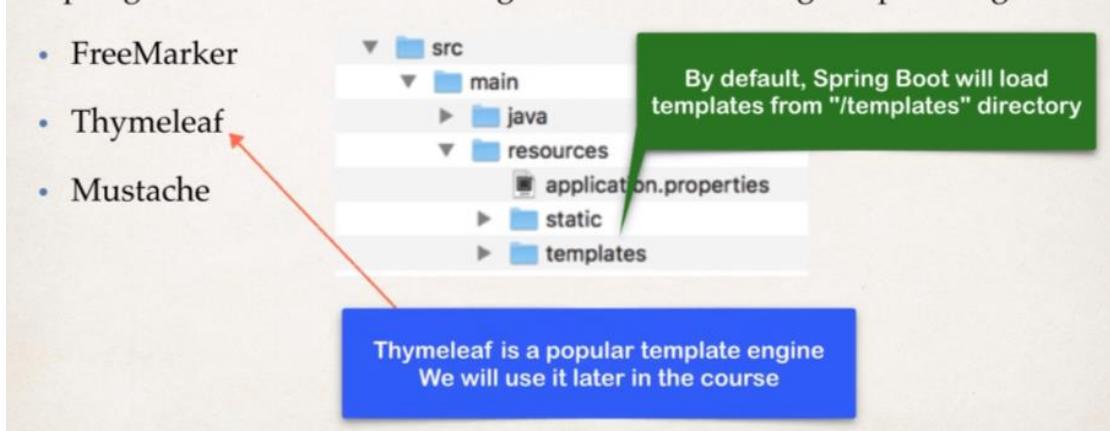
Do not use the `src/main/webapp` directory if your application is packaged as a JAR.

Although this is a standard Maven directory, it works only with WAR packaging.

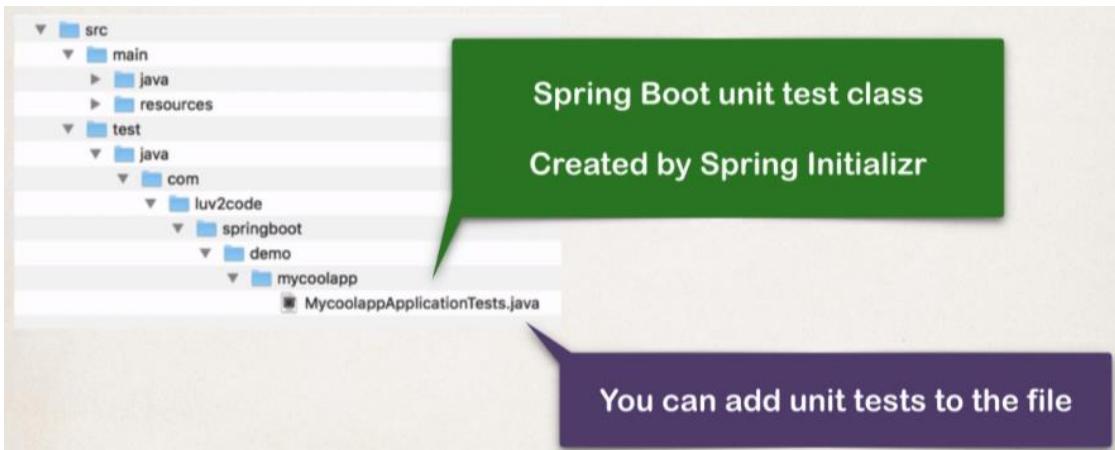
It is silently ignored by most build tools if you generate a JAR.

Templates

- Spring Boot includes auto-configuration for following template engines
 - FreeMarker
 - Thymeleaf
 - Mustache



Unit Tests



➤ Spring Boot Starters:

- A curated list of Maven Dependencies.
- A collection of dependencies grouped together.
- Tested and verified by the Spring Development team.
- These starters make it much easier for the developer to get started with Spring.
- Reduces the amount of Maven configuration that we need to do.

Let's consider an example of **Spring MVC**:

So, to build a Spring MVC app, you normally need dependencies;

- Spring MVC
- Hibernate Validator for some form validation
- And for a given web template, we need to put the dependency.

```
<!-- Spring MVC -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.2.RELEASE</version>
</dependency>

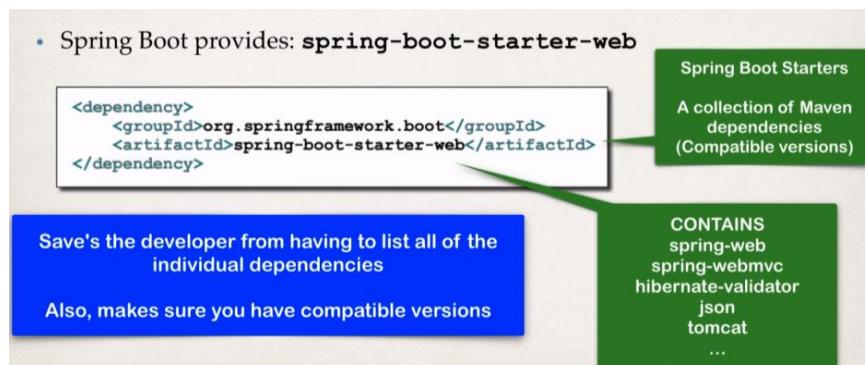
<!-- Form validation: Hibernate Validator -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>

<!-- Web template: JSP or Thymeleaf etc -->
<dependency>
    <!-- ... -->
</dependency>
```

Here is the solution:

Spring boot provides: **spring-boot-starter-web**

Add this Maven dependency to the Maven pom file. And this dependency entry actually contains other dependencies as it contains spring-web, spring-webmvc, hibernate-validator, json, tomcat.



So, in the spring initializer we need to select web dependency:

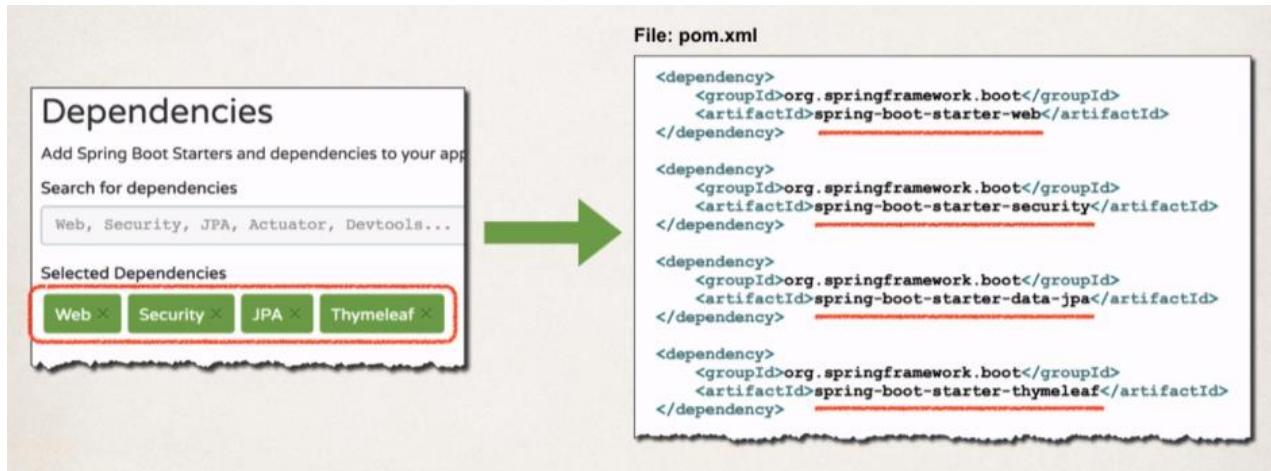
Spring Initializr

- In Spring Initializr, simply select **Web** dependency
- You automatically get **spring-boot-starter-web** in **pom.xml**

The screenshot shows the Spring Initializer interface. In the 'Dependencies' section, the 'Search for dependencies' input field contains 'web'. Below it, a list of starters is shown: 'Web' (highlighted with a red box), 'Reactive Web', and 'Reactive web development with Netty and Spring WebFlux'. The 'Web' option is described as 'Full-stack web development with Tomcat and Spring MVC'.

Another Example:

- Support we are building a Spring app that needs: Web, Security.
- Simply select the dependencies in the Spring initializer.
- It will add the appropriate Spring Boot starters to your pom.xml.



Spring Boot Starters

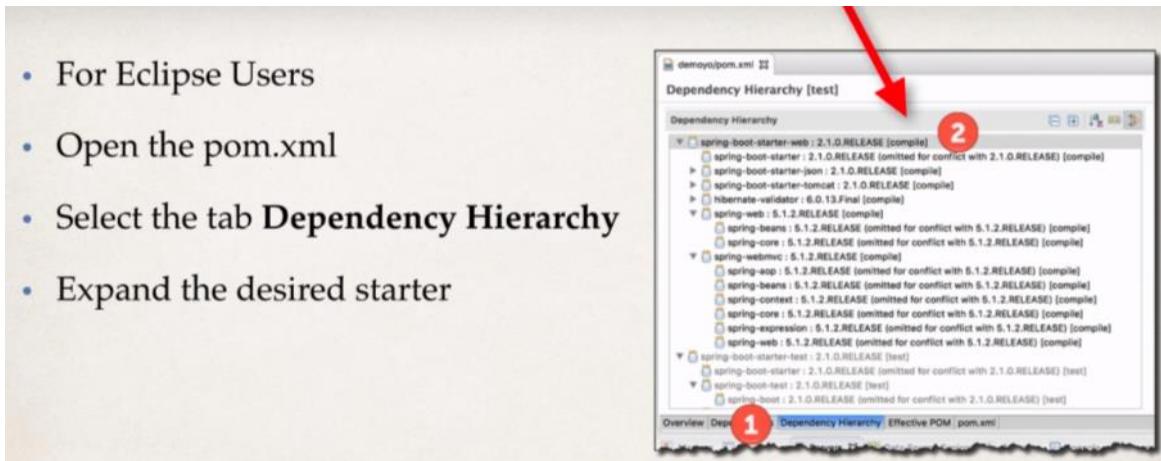
- There are 30+ Spring Boot Starters from the Spring Development team

Name	Description
spring-boot-starter-web	Building web apps, includes validation, REST. Uses Tomcat as default embedded server
spring-boot-starter-security	Adding Spring Security support
spring-boot-starter-data-jpa	Spring database support with JPA and Hibernate
...	

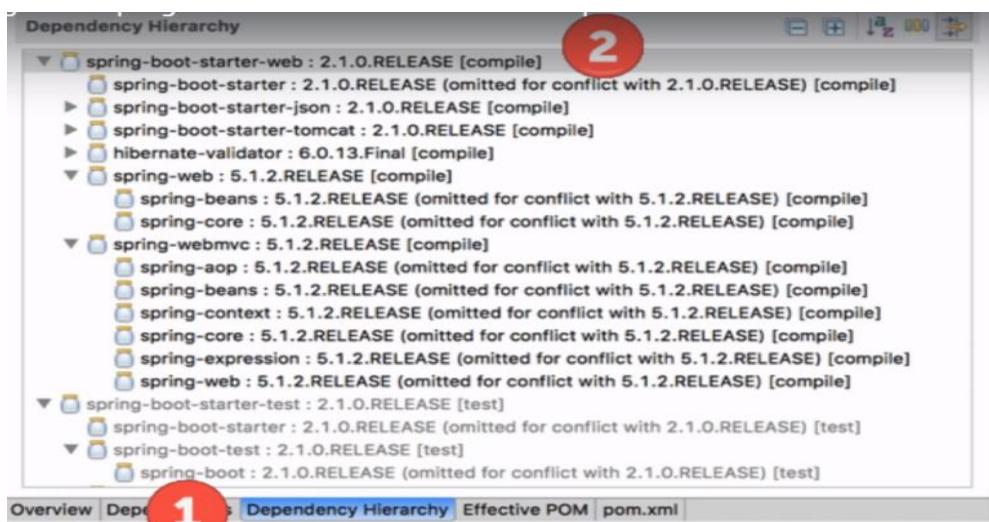
If we want a **full list** of Spring boot starters: [Link](#)

➤ What is in the Starter dependency that we have just into pom.xml?

- For Eclipse Users
- Open the pom.xml
- Select the tab **Dependency Hierarchy**
- Expand the desired starter



Eclipse Way:



➤ Spring Boot Starter Parent:

- Spring Boot provides a "Starter Parent"
- This is a special starter that provides Maven defaults

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.0.RELEASE</version>
  <relativePath/>
</parent>
```

Included in pom.xml
when using
Spring Initializr

So, the Maven Defaults are actually defined in the Starter Parent.

- We get the Default compiler level
- Also have support for UTF- 8 source coding
- Others..

To override a default, set as a property

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.0.RELEASE</version>
  <relativePath/>
</parent>

<properties>
  <java.version>12</java.version>
</properties>
```

Specify your
Java version

For ex in our project, we want to use java 12, or 15. then we can simply override that property.

For **spring-boot-starter-*** dependencies, no need to list version.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.0.RELEASE</version>
  <relativePath/>
</parent>

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

Specify version of
Spring Boot

Inherit version from
Starter Parent

No need to list individual versions
Great for maintenance!

Basically, we have here **spring-boot-starter-parent** where we specify the version that we are using and then for the actual dependencies, we simply inherit the version from the starter parent. So, there is no need to list individual versions and it's great for maintenance.

Starter parent also provides the default configuration of Spring boot plugin, in our build section of pom.xml file, we simply refer to the spring boot maven plugin and that's it. And now we can simply run it from our command line using the command:

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

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

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

Benefits of the Spring Boot Starter Parent

- Default Maven configuration: Java version, UTF-encoding etc
- Dependency management
 - Use version on parent only
 - **spring-boot-starter-*** dependencies inherit version from parent
- Default configuration of Spring Boot plugin

Section 71: Spring Boot - Spring Boot Dev Tools and Spring Boot Actuator

0 / 4 | 35min

The Problem:

When running Spring Boot applications.

If we make changes to our source code, then we have to manually restart our application.

Solution: Spring boot Dev Tools

- **Spring-boot-devtools** to the rescue!
- Simply add the dependency to your pom.xml.

- Adding the dependency to your POM file

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

Automatically restarts your application when code is updated

Spring boot Actuator [Java Project Link](#)

Problems:

- How can I monitor and Manage my application?
- How can I check the application health?
- How can I access application Metrics?

One Solution is: Spring boot Actuator

- It exposes endpoints to monitor and manage your application
- You easily get DevOps functionality
- Simply add the dependency to your POM file
- REST endpoints are automatically added to your application.

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

Enables
Spring Boot Actuator

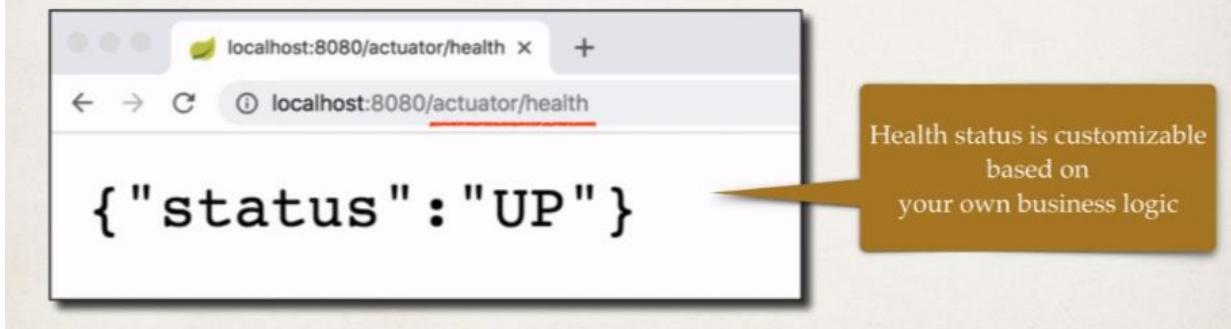
- Endpoints are prefixed with: **/actuator**

Name	Description
/health	Health information about your application
/info	Information about your project
...	

Remember:

You get these new REST endpoints for FREE!

- **/health** checks the status of your application
- Normally used by monitoring apps to see if your app is up or down



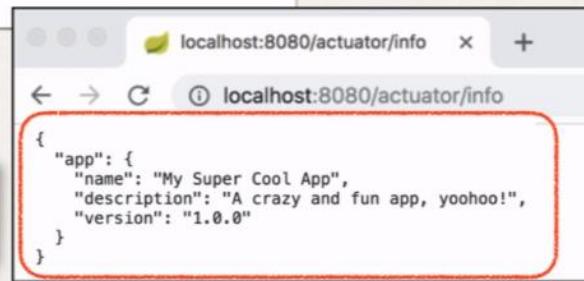
So, we can customize this "/info" endpoint.

- Update **application.properties** with your app info

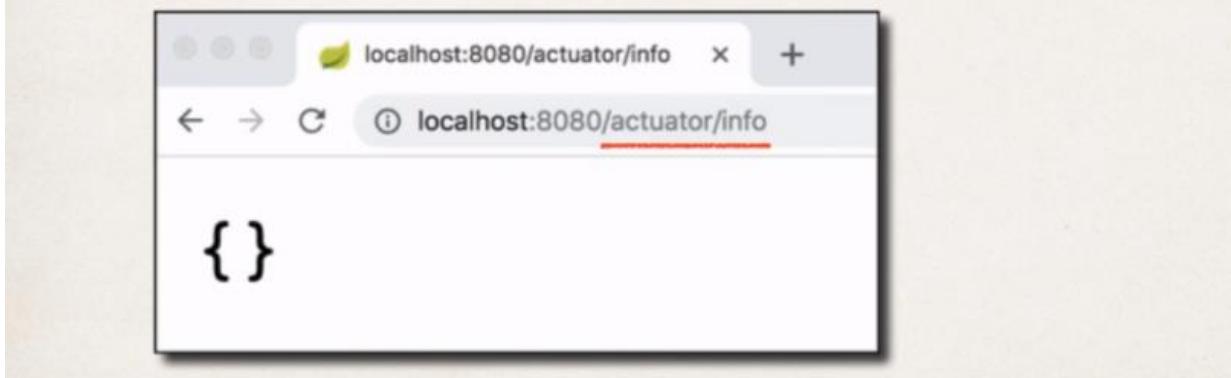
File: src/main/resources/application.properties

```
info.app.name=My Super Cool App
info.app.description=A crazy and fun app, yoohoo!
info.app.version=1.0.0
```

Properties starting with "info."
will be used by /info



- **/info** gives information about your application
- Default is empty



Spring Boot Actuator Endpoints

- There are 10+ Spring Boot Actuator endpoints

Name	Description
/auditevents	Audit events for your application
/beans	List of all beans registered in the Spring application context
/mappings	List of all @RequestMapping paths
...	

Exposing Endpoints

- By default, only `/health` and `/info` are exposed
- To expose all actuator endpoints over HTTP

File: `src/main/resources/application.properties`

```
# Use wildcard "*" to expose all endpoints
# Can also expose individual endpoints with a comma-delimited list
#
management.endpoints.web.exposure.include=*
```

expose all endpoints

Development Process:

1. Edit `pom.xml` and add `spring-boot-starter-actuator`
2. View actuator endpoints for: `/health` and `/info`
3. Edit `application.properties` to customize `/info`.

```
application.properties
1 info.app.name=My Suuper Cool App
2 info.app.description=A crazy funu app, yoohoo!
3 info.app.version=1.0.0
4
```

```

{
  "app": {
    "name": "My Suuper Cool App",
    "description": "A crazy funu app, yoohoo!",
    "version": "1.0.0"
  }
}

```

Exposing all the other Endpoints of Spring Boot Actuator

```

application.properties
1 info.app.name=My Suuper Cool App
2 info.app.description=A crazy funu app, yoohoo!
3 info.app.version=1.0.0
4
5
6 #use Wildcard "*" to expose all endpoints
7 #can also expose individual endpoints with a comma-delimited list
8
9 management.endpoints.web.exposure.include=*

```

MydemoappApplication (2) [Java Application]

```

2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.boot.SpringApplication : Starting MydemoappApplication using Java 14.0.1 on LAPTOP-HL8QE68V with PID 14080
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.boot.SpringApplication : No active profile set, falling back to default profiles: default
2023-09-12 14:01:25.525  INFO 14080 --- [           main] c.a.t.web.embedded.servlet.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.b.w.embedded.tomcat.TomcatEmbeddedService : Starting service [Tomcat]
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.a.catalina.startup.EmbeddedServletEngine : Starting Servlet engine: [Apache Tomcat/9.0.45]
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.b.w.embedded.tomcat.TomcatContext : Initializing Spring embedded WebApplicationContext
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.b.w.embedded.tomcat.TomcatTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.b.w.embedded.tomcat.TomcatServer : LiveReload server is running on port 35729
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Exposing 13 endpoint(s) beneath base path '/actuator'
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.boot.SpringApplication : Tomcat started on port(s): 8080 (http) with context path ''
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.boot.SpringApplication : Started MydemoappApplication in 0.864 seconds (JVM running for 287.709)
2023-09-12 14:01:25.525  INFO 14080 --- [           main] o.s.b.w.embedded.tomcat.TomcatContext : Condition evaluation delta:

```

We can test then using endpoints:

<http://localhost:8080/actuator/mappings>

<http://localhost:8080/actuator/beans>

<http://localhost:8080/actuator/threaddump>

Get A List of Beans

- Access `http://localhost:8080/actuator/beans`

```
{  
    "contexts": {  
        "application": {  
            "beans": {  
                "endpointCachingOperationInvokerAdvisor": {  
                    "aliases": [],  
                    "scope": "singleton",  
                    "type": "org.springframework.boot.actuate.endpoint.invoker.cache.CachingOperationInvokerAdvisor",  
                    "resource": "class path resource [org/springframework/boot/actuate/au...",  
                    "dependencies": [  
                        "environment"  
                    ]  
                },  
                "defaultServletHandlerMapping": {  
                    "aliases": [],  
                    "scope": "singleton",  
                    "type": "org.springframework.web.servlet.HandlerMapping",  
                    "resource": "class path resource [org/springframework/boot/autoconfig...",  
                    "dependencies": []  
                },  
                "org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfigura..."  
            }  
        }  
    }  
}
```

What about security??

We'll add security
in later videos

Spring Boot Actuator Security:

Previously we learn how to expose all the spring boot actuator endpoints and we must be thinking about security,

We may not want to expose all of the endpoint information to anyone.

We will add Spring Security to project and endpoints are secured.

We need spring-boot-starter-security dependency.

Secured Endpoints:

- Now when we access: /actuator/beans
- Spring security will prompt for login

We can also override default username and generated password:

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

Enable Spring Security

- Spring Security will prompt for login

The screenshot shows a "Please sign in" form with fields for "Username" and "Password". A purple callout box says "Default user name: user". Below the form is a terminal window showing log output:

```
11-02 21:05:57.074 INFO 24986 --- [main] .s.s.UserDetailsSe
Using generated security password: 78fd68a6-c190-421d-934b-df7852fc7dc2
```

A red callout points to the password in the log with the text "Check console logs for password".

In the `application.properties` file:

```
spring.security.user.name=scott
spring.security.user.password=tiger
```

Customizing Spring Security

- You can customize Spring Security for Spring Boot Actuator
 - Use a database for roles, encrypted passwords etc ...
- Follow the same techniques

```
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private DataSource securityDataSource;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.jdbcAuthentication().dataSource(securityDataSource);
    }
}
```

A red callout points to the `securityDataSource` field with the text "Read security info from database". Another red callout points to the `Security` word with the text "Security".

Customizing Spring Security

- You can customize Spring Security for Spring Boot Actuator
 - Use a database for roles, encrypted passwords etc ...
- Follow the same techniques as regular Spring Security

```
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {  
  
    @Autowired  
    private DataSource securityDataSource;  
  
    @Override  
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
        auth.jdbcAuthentication().dataSource(securityDataSource);  
    }  
}  
  
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/actuator/**").hasRole("ADMIN")  
        ...  
}
```

Only authorize ADMIN users

Excluding Endpoints

- To exclude `/health` and `/info`

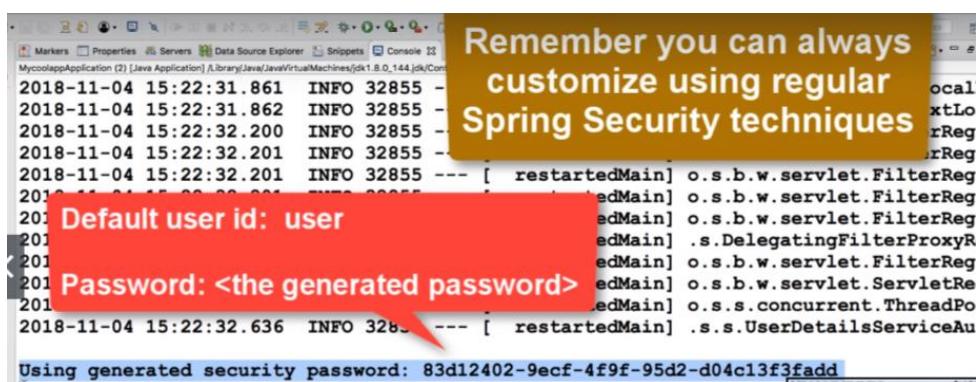
File: `src/main/resources/application.properties`

```
# Exclude individual endpoints with a comma-delimited list  
#  
management.endpoints.web.exposure.exclude=health,info
```

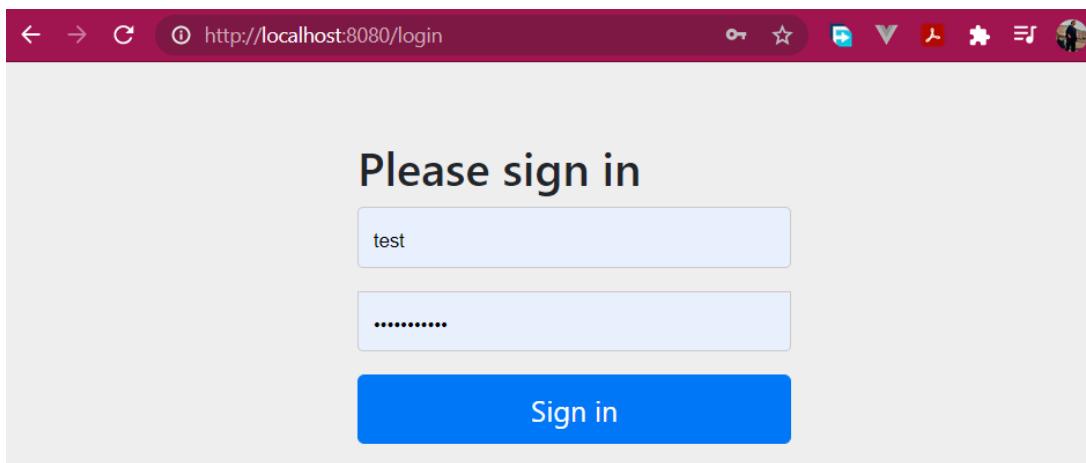
exclude

Development Process:

1. Edit `pom.xml` and add `spring-boot-starter-security`
2. Verify security on actuator endpoints for: `/beans` etc.
3. Disable endpoints for `/health` and `/info`.



On hitting the endpoints: <http://localhost:8080/actuator/mappings>, we will get to see this login page.



But by default, /health and /info will not be secure so we will disable those endpoints.

```
application.properties
1 info.app.name=My Suuper Cool App
2 info.app.description=A crazy funu app, yoohoo!
3 info.app.version=1.0.0
4
5
6 #use Wildcard "*" to expose all endpoints
7 #can also expose individual endpoints with a comma-delimited list
8
9 management.endpoints.web.exposure.include=*
10
11 #Exclude individual endpoints with comma-delimited list
12 management.endpoints.web.exposure.exclude=health,info
```

A screenshot of a code editor showing the "application.properties" file. The file contains configuration for a Spring Boot application. Lines 12 and 13, which define the exclude list for management endpoints, are highlighted with a yellow background.

Now /health and /info will not be available anywhere.



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon May 17 21:50:02 IST 2021

There was an unexpected error (type=Not Found, status=404).

No message available

Section 72: Spring Boot - Running Spring Boot Apps from the Command Line

0 / 2 | 11min

Overview:

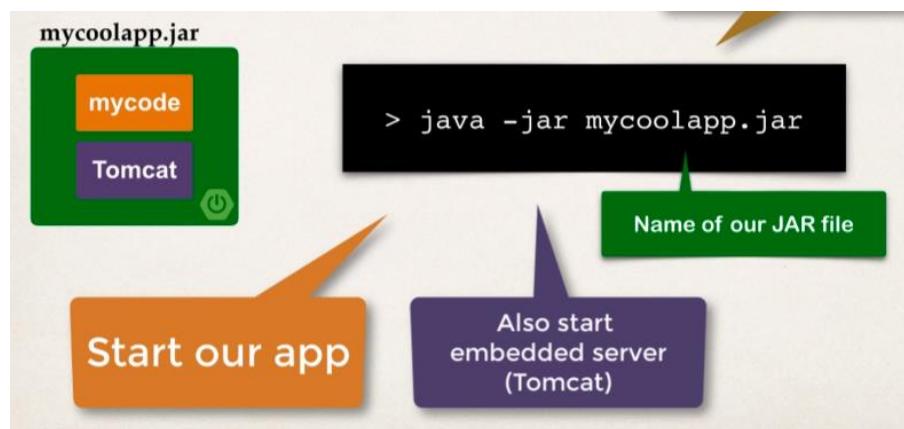
Running from the Command-Line:

- No need to have IDE open/running
- Since we are using Spring Boot, the server is embedded in our JAR File.
- So, there is no need to have separate server installed/running
- Since our spring boot apps are self-contained.



Two Options for running the app:

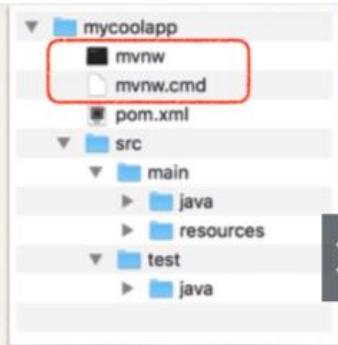
Option 1: Use `java -jar`



Option 2: Use Spring boot maven Plugin

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

- **mvnw** allows you to run a Maven project
 - No need to have Maven installed or present on your path
 - If correct version of Maven is NOT found on your computer
 - **Automatically downloads** correct version and runs Maven



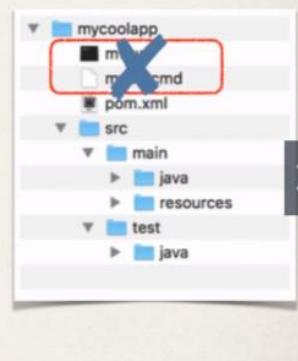
- **Automatically downloads** correct version and runs Maven
- Two files are provided
 - **mvnw.cmd** for MS Windows
 - **mvnw.sh** for Linux/Mac

> mvnw clean compile test

\$./mvnw clean compile test

- If you already have Maven installed previously
 - Then you can ignore/delete the **mvnw** files
- Just use Maven as you normally would

\$ mvn clean compile test



Option 2: Use Spring Boot Maven plugin

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

To package executable jar or war archive

Can also easily run the app

\$./mvnw package

Can also just use:

mvn package
 mvn spring-boot:run

\$./mvnw spring-boot:run

Development Process:

1. Exit the IDE
2. Package the app using **mvnw package**
3. Run app using **java -jar**
4. Run app using spring boot maven plugin, **mvnw spring-boot:run**.

Go to the directory where the app is present then;

```
D:\Java_Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot>cd 03-spring-boot-commandline-demo
D:\Java_Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot\03-spring-boot-commandline-demo>mvnw package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.example.demo:mydemoapp >-----
[INFO] Building mydemoapp 0.0.1-SNAPSHOT
[INFO]   [ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ mydemoapp ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ mydemoapp ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to D:\Java_Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot\03-spring-boot-commandline...
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:testResources (default-testResources) @ mydemoapp ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] skip non existing resourceDirectory D:\Java_Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot\03-spring-boot-commandline...
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ mydemoapp ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\Java_Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot\03-spring-boot-commandline...
[INFO]
```

Using mvnw package command to build the JAR File.

And the jar file will be created into the Target subdirectory:

```
2021-05-17 22:21:43.359 INFO 5884 --- [           main] c.e.d.m.MydemoappApplicationTests      : Starting MydemoappApplicationTests using Java 14.0.1 on LAPTOP-HL8QE68V with PID 5884
va Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot\03-spring-boot-commandline-demo)
2021-05-17 22:21:43.365 INFO 5884 --- [           main] c.e.d.m.MydemoappApplicationTests      : No active profile set, falling back to default profiles: default
2021-05-17 22:21:44.574 INFO 5884 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-05-17 22:21:44.938 INFO 5884 --- [           main] c.e.d.m.MydemoappApplicationTests      : Started MydemoappApplicationTests in 1.93 seconds (JVM running for 2.746)
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.721 s - in com.example.demo.mydemoapp.MydemoappApplicationTests
2021-05-17 22:21:45.400 INFO 5884 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicationTaskExecutor'
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ mydemoapp ---
[INFO] Building jar: D:\Java_Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot\03-spring-boot-commandline-demo\target\mydemoapp-0.0.1-SNAPSHOT.jar
[INFO]
```

And in the end, we will see BUILD SUCCESS log:

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-dependency-tree/3.0.2/maven-dependency-tree-3.0.2.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/ow2/asm/asm-util/8.0/asm-util-8.0.jar (85 kB at 85 kB)
Downloaded from central: https://repo.maven.apache.org/maven2/org/vafer/jdependency/2.4.0/jdependency-2.4.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-lang3/3.7/commons-lang3-3.7.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/jdom/jdom2/2.0.6/jdom2-2.0.6.jar (305 kB at 305 kB)
[INFO] Replacing main artifact with repackaged archive
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 24.179 s
[INFO] Finished at: 2021-05-17T22:22:02+05:30
[INFO] -----
```

Now move into the `target` subdirectory:

```
D:\Java_Learning\spring_jar_sw\Spring-security-video-tutorial-files\CommandLine-Demo-Spring-boot\03-spring-boot-commandline-demo>cd target
```

Now we need to run the created jar file:

Using the command `java -jar mydemoapp-0.0.1-SNAPSHOT.jar`

And it runs the application on cmd.

Now to run via spring boot maven plugin:

- `cd..`
 - go to application folder; use command: `mvnw spring-boot:run`

And this will run the application.

➤ Spring Boot Properties:

1. Spring Boot can be configured in the application.properties file.
 2. Server port, context path, actuator, security etc

- The properties are roughly grouped into the following categories

Core

Web

Security

Data

Actuator

Integration

DevTools

Testing

File: src/main/resources/application.properties

```
# Log levels severity mapping
logging.level.org.springframework=DEBUG
logging.level.org.hibernate=TRACE
logging.level.com.luv2code=INFO
```

Set logging levels
based on
package names

Web Properties

File: src/main/resources/application.properties

```
# HTTP server port
server.port=7070

# Context path of the application
server.servlet.context-path=/my-silly-app

# Default HTTP session time out
server.servlet.session.timeout=15m
...
```

http://localhost:7070/my-silly-app/fortune

15 minutes

Default timeout
30 minutes

Actuator Properties

Actuator

File: src/main/resources/application.properties

```
# Endpoints to include by name or wildcard  
management.endpoints.web.exposure.include=*
```

```
# Endpoints to exclude by name or wildcard  
management.endpoints.web.exposure.exclude=beans,mapping
```

```
# Base path for actuator endpoints  
management.endpoints.web.base-path=/actuator
```

```
...
```

<http://localhost:7070/actuator/health>

Security Properties

File: src/main/resources/application.properties

```
# Default user name  
spring.security.user.name=admin
```

```
# Password for default user  
spring.security.user.password=topsecret
```

```
...
```

Data Properties

Data

File: src/main/resources/application.properties

```
# JDBC URL of the database  
spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce
```

```
# Login username of the database  
spring.datasource.username=scott
```

```
# Login password of the database  
spring.datasource.password=tiger
```

```
...
```

More on this
in later videos

➤ Spring Employee CRUD Operations:

For setting up DAO service:

In traditional Spring

We normally had to do this configuration manually;

The image shows two side-by-side code snippets. The left snippet is in XML, detailing the setup of a Database DataSource, a session factory, and a transaction manager. The right snippet is in Java, showing the corresponding beans for a LocalSessionFactoryBean and a MyDataSource bean.

```
<!-- Step 1: Define Database DataSource / connection pool -->
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <!-- destroy-method="close" />
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/web_customer_tracker" />
    <property name="user" value="springstudent" />
    <property name="password" value="springstudent" />
</bean>

<!-- these are connection pool properties for C3P0 -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />
    <property name="packagesToScan" value="com.javacode.springdemo.entity" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<!-- Step 3: Setup Hibernate transaction manager -->
<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

```
bean
public DataSource myDataSource() {
    ComboPooledDataSource myDataSource = new ComboPooledDataSource();
    try {
        myDataSource.setDriverClass("com.mysql.jdbc.Driver");
    } catch (PropertyVetoException e) {
        throw new RuntimeException(e);
    }

    myDataSource.setInitialPoolSize(new Integer("10"));
    myDataSource.setMinPoolSize(new Integer("5"));
    myDataSource.setMaxPoolSize(new Integer("20"));
    myDataSource.setAcquireIncrement(new Integer("5"));

    myDataSource.setTestConnectionOnCheckin(true);
    myDataSource.setTestConnectionOnCheckout(true);

    return myDataSource;
}

bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();

    // create session factory
    sessionFactory.setDataSource(myDataSource());
    sessionFactory.setPackagesToScan(new String[]{"hibernate.packageToScan"});
    sessionFactory.setHibernateProperties(getHibernateProperties());
    return sessionFactory;
}
```

So, there should be easier solution to do all this as this is very error prone.

Spring boot comes to the rescue:

- Spring boot will automatically configure your data source for you.
- Based on entries from Maven pom file:
 - JDBC Driver: **mysql-connector-java**
 - Spring Data (ORM): **spring-boot-starter-data-jpa**
- DB connection info from **application.properties**

So, spring boot will actually create all these info to create data source for us.

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory?
useSSL=false&serverTimezone=UTC

spring.datasource.username=springstudent
spring.datasource.password=springstudent
```

No need to give JDBC driver class name
Spring Boot will automatically detect it based on URL

There is additional data source properties that we can set:

Properties are available to configure connection pool etc.

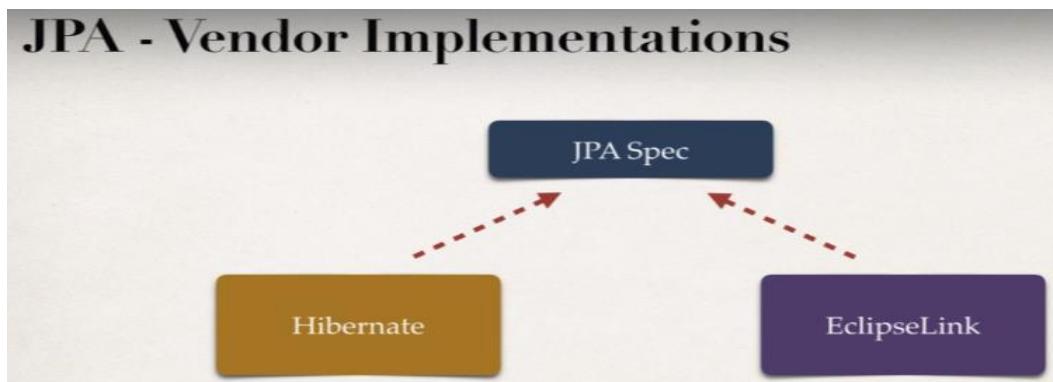
[Link](#) to follow more about spring boot properties. And go to **spring.datasource.***

Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
 - **DataSource**, **EntityManager**, ...
- You can then inject these into your app, for example your DAO
- **EntityManager** is from Java Persistence API (JPA)

What is JPA?

- **Java Persistence API (JPA)**
 - Standard API for Object-to-Relational-Mapping (ORM)
- It's only a Specification
 - Defines a set of interfaces
 - Requires an implementation to be usable



What are Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations
 - For example, if Vendor ABC stops supporting their product
 - You could switch to Vendor XYZ without vendor lock in

- In spring Boot, **hibernate** is default implementation of JPA.
- **EntityManager** is similar to Hibernate **SessionFactory**.
- **EntityManager** can serve as a wrapper for a hibernate **Session** Object.
- So, in our code we can inject **EntityManager** into our DAO

Various DAO techniques:

- Version 1: Use EntityManager but leverage native Hibernate API
- Version 2: Use EntityManager and standard JPA API
- Version 3: Spring Data JPA

Version 1: Use EntityManager but leverage native Hibernate API [Java Project link](#)

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

    private EntityManager entityManager;
    @Autowired
    public EmployeeDAOHibernateImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }
    ...
}
```

The diagram illustrates the relationship between the auto-created EntityManager by Spring Boot and its injection into the DAO implementation. A green speech bubble labeled "Automatically created by Spring Boot" points to the EntityManager field declaration. A brown speech bubble labeled "Constructor injection" points to the constructor where the EntityManager is injected via the @Autowired annotation.

```
@Override
@Transactional
public List<Employee> findAll() {
    // get the current hibernate session
    Session currentSession = entityManager.unwrap(Session.class);

    // create a query
    Query<Employee> theQuery =
        currentSession.createQuery("from Employee", Employee.class);

    // execute query and get result list
    List<Employee> employees = theQuery.getResultList();

    // return the results
    return employees;
}
```

The diagram shows the execution flow of the findAll() method. A blue speech bubble labeled "Get current Hibernate session" points to the unwrapping of the EntityManager to a Session object. A brown speech bubble labeled "Using native Hibernate API" points to the creation and execution of the native Hibernate query using the Session's createQuery method.

Development Process:

- Update DB configs in `application.properties`

```
application.properties
1 #
2 # JDBC properties
3 #
4 spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory?useSSL=false
5 spring.datasource.username=springstudent
6 spring.datasource.password=springstudent
```

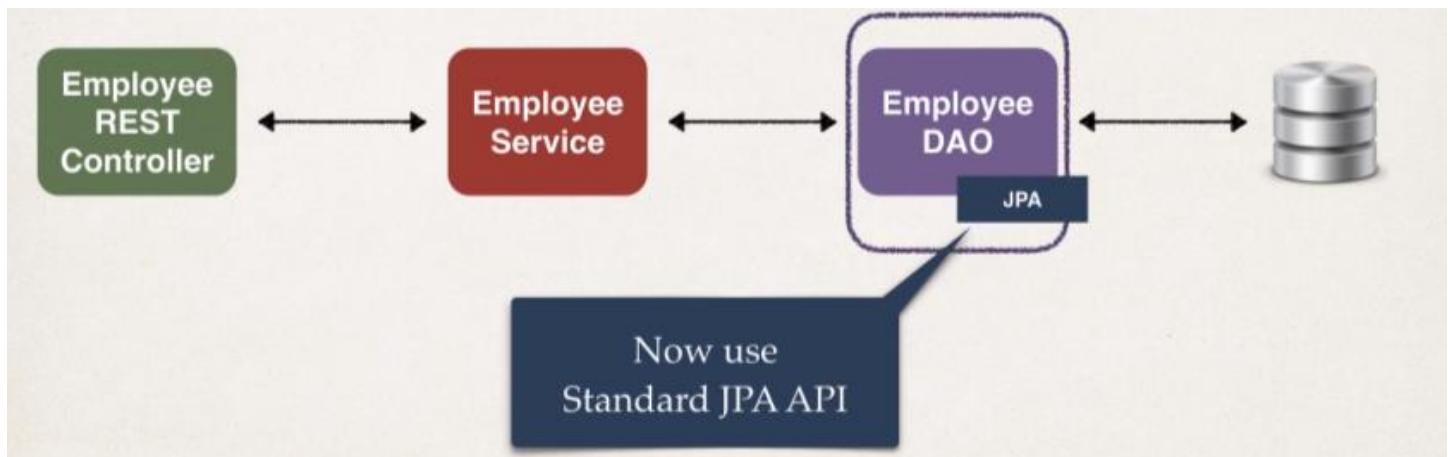
Spring Boot will automatically create beans for
DataSource, EntityManager, ...

- Create `Employee` entity
- Create DAO interface.
- Create DAO implementation
- Create REST Controller to use DAO.

➤ Create a JPA DAO in Spring boot: [Java Project Link](#)

We have already covered version 1: Use EntityManager but leverage native Hibernate API

Here we will focus on the **usage of EntityManager and standard Data JPA API**



Benefits of JPA:

- By having a standard API, you are not locked to vendor's implementation.
- Maintain portable, flexible code
- Can theoretically switch vendor implementation
- If vendor ABC stops supporting their product, then we can easily switch to vendor XYZ without vendor lock in.

Standard JPA API

- The JPA API methods are similar to Native Hibernate API
- JPA also supports a query language: JPQL (JPA Query Language)
 - For more details on JPQL, see this link

www.luv2code.com/jpql

Comparing JPA to Native Hibernate Methods

Action	Native Hibernate method	JPA method
Create/save new entity	<code>session.save(...)</code>	<code>entityManager.persist(...)</code>
Retrieve entity by id	<code>session.get(...) / load(...)</code>	<code>entityManager.find(...)</code>
Retrieve list of entities	<code>session.createQuery(...)</code>	<code>entityManager.createQuery(...)</code>
Save or update entity	<code>session.saveOrUpdate(...)</code>	<code>entityManager.merge(...)</code>
Delete entity	<code>session.delete(...)</code>	<code>entityManager.remove(...)</code>

Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Step-By-Step

Let's build a
DAO layer for this

Using
Standard
JPA API

DAO Impl

Same interface for
consistent API

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {
    private EntityManager entityManager;

    @Autowired
    public EmployeeDAOJpaImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    ...
}
```

Automatically created
by Spring Boot

Constructor
injection

Get a list of employees

JPA API

```
@Override
public List<Employee> findAll() {
    // create a query
    TypedQuery<Employee> theQuery =
        entityManager.createQuery("from Employee", Employee.class);

    // execute query and get result list
    List<Employee> employees = theQuery.getResultList();

    // return the results
    return employees;
}
```

Remember: No need to manage transactions ...

Handled at Service layer with @Transactional

Using
Standard
JPA API

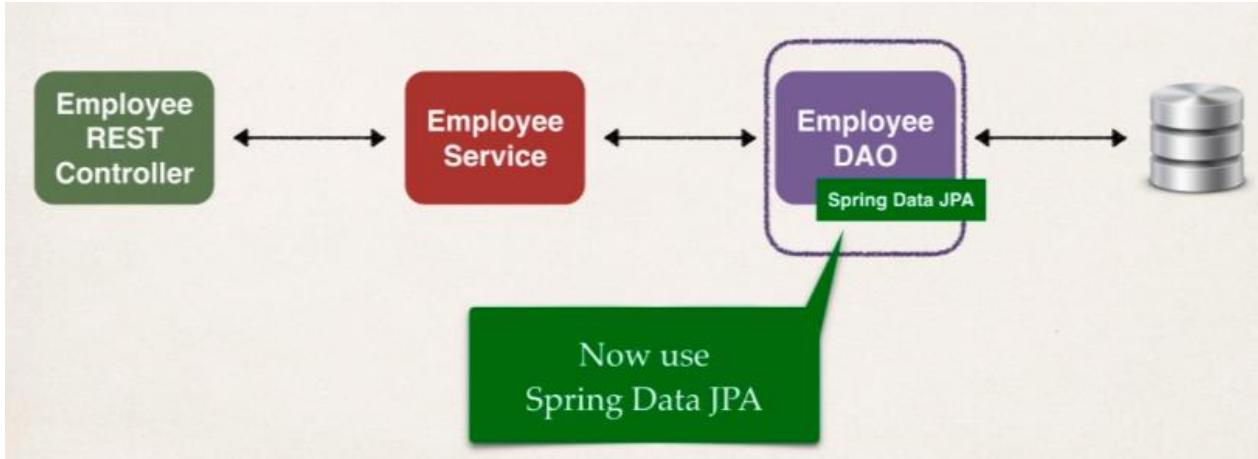
Delete an existing employee

JPA API

```
@Override
public void deleteById(int theId) {
    // delete object with primary key
    Query theQuery = entityManager.createQuery(
        "delete from Employee where id=:employeeId");

    theQuery.setParameter("employeeId", theId);
    theQuery.executeUpdate();
}
```

We will **Spring Data JPA**: [Java Project Link](#)



The Problem

- We saw how to create a DAO for **Employee**
- What if we need to create a DAO for another entity?
 - **Customer, Student, Product, Book ...**
- Do we have to repeat all of the same code again???

```

public interface EmployeeDAO {
    public List<Employee> findAll();
    public Employee findById(int theId);
    public void save(Employee theEmployee);
    public void deleteById(int theId);
}

@Repository
public class EmployeeJpaImpl implements EmployeeDAO {
    private EntityManager entityManager;
    @Autowired
    public EmployeeJpaImpl(EntityManager entityManager) {
        entityManager = entityManager;
    }

    @Override
    public List<Employee> findAll() {
        // create a query
        TypedQuery<Employee> query = entityManager.createQuery("from Employee", Employee.class);
        // execute query and get results
        List<Employee> employees = query.getResultList();
        // return the results
        return employees;
    }

    @Override
    public Employee findById(int theId) {
        // get employee
        Employee employee = entityManager.find(Employee.class, theId);
        // return employee
        return employee;
    }
}
  
```

Creating DAO

- You may have noticed a pattern with creating DAOs

```

@Override
public Employee findById(int theId) {
    // get data
    Employee theData = entityManager.find(Employee.class, theId);

    // return data
    return theData;
}
  
```

Most of the code is the same

Only difference is the entity type and primary key

Entity type

Primary key

My Wish Diagram



Spring Data JPA - Solution

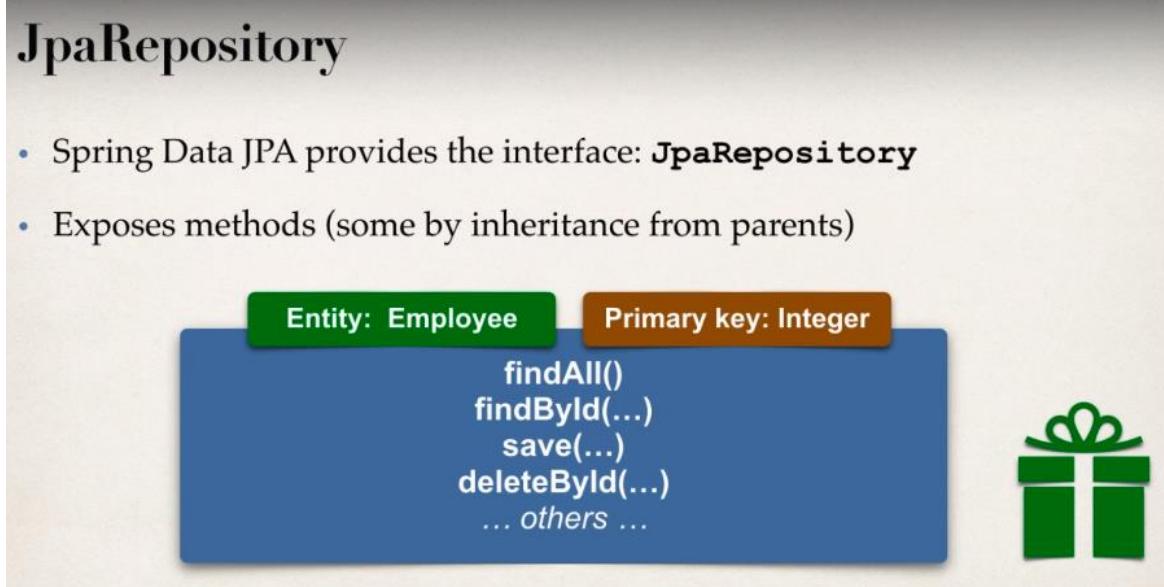
- Spring Data JPA is the solution!!!! <https://spring.io/projects/spring-data-jpa>
- Create a DAO and just plug in your entity type and primary key
- Spring will give you a CRUD implementation for FREE like MAGIC!!
 - Helps to minimize boiler-plate DAO code ... yaaay!!!

More than 70% reduction in code ... depending on use case



JpaRepository

- Spring Data JPA provides the interface: **JpaRepository**
- Exposes methods (some by inheritance from parents)



Development Process

Step-By-Step

1. Extend **JpaRepository** interface

2. Use your Repository in your app

No need for implementation class

Step 1: Extend JpaRepository interface

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {  
    // that's it ... no need to write any code LOL!  
}
```

No need for implementation class

Get these methods for free



Entity: Employee
Primary key: Integer
findAll()
findById(...)
save(...)
deleteById(...)
... others ...

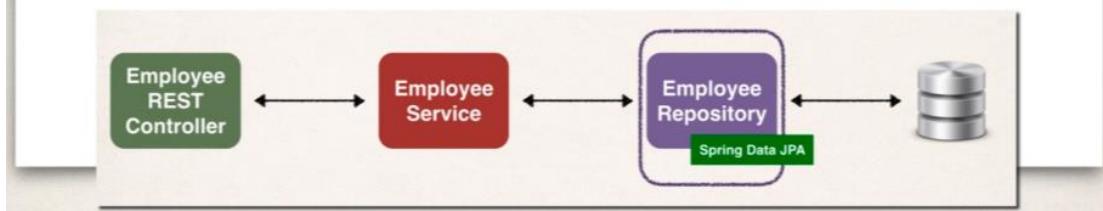
Entity type

Primary key

Step 2: Use Repository in your app

```
@Service  
public class EmployeeServiceImpl implements EmployeeService {  
    private EmployeeRepository employeeRepository;  
  
    @Autowired  
    public EmployeeServiceImpl(EmployeeRepository theEmployeeRepository) {  
        employeeRepository = theEmployeeRepository;  
    }
```

Our repository



```

@Service
public class EmployeeServiceImpl implements EmployeeService {
    private EmployeeRepository employeeRepository;

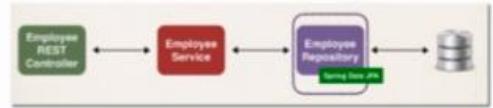
    @Autowired
    public EmployeeServiceImpl(EmployeeRepository theEmployeeRepository) {
        employeeRepository = theEmployeeRepository;
    }

    @Override
    public List<Employee> findAll() {
        return employeeRepository.findAll();
    }

    ...
}

```

Our repository



Magic method that is available via repository



Minimized Boilerplate Code

Before Spring Data JPA

```

public interface EmployeeDAO {
    public List<Employee> findAll();
    public Employee findById(int theId);
    public void save(Employee theEmployee);
    public void deleteById(int theId);
}

@Repository
public class EmployeeJdbcImpl implements EmployeeDAO {
    private EntityManager entityManager;
    @Autowired
    public EmployeeJdbcImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }
    @Override
    public List<Employee> findAll() {
        // create a query
        TypedQuery<Employee> query =
            entityManager.createQuery("from Employee", Employee.class);
        // execute query and get result list
        List<Employee> employees = query.getResultList();
        // return the results
        return employees;
    }
    @Override
    public Employee findById(int theId) {
        // get employee
        Employee theEmployee =
            entityManager.find(Employee.class, theId);
        // return employee
        return theEmployee;
    }
}

```

2 Files
30+ lines of code

After Spring Data JPA

```

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
    // that's it ... no need to write any code LOL!
}

```

1 File
3 lines of code!

No need for implementation class



Advanced Features

- Advanced features available for
 - Extending and adding custom queries with JPQL
 - Query Domain Specific Language (Query DSL)
 - Defining custom methods (low-level coding)

Visit for more info: [Link](#)

JPARepository make use of **Optional**; so instead of checking for nulls we can make use of **Optional** to see if value is present or not.

```
@Override  
public Employee findById(int theId) {  
    Optional<Employee> result = employeeRepository.findById(theId);  
  
    Employee theEmployee = null;  
    if(result.isPresent()) {  
        theEmployee= result.get();  
    }  
    else {  
        throw new RuntimeException("Did not find Employee id - "+theId);  
    }  
    return theEmployee;  
}
```

Section 77: Spring Boot - Spring Data REST - Real-Time Project

0 / 3 | 30min

- Earlier we saw the magic of **Spring-data-JPA**
- This helped to eliminate Boilerplate code.

The Problem

- We saw how to create a REST API for **Employee**
- Need to create REST API for another entity?
 - **Customer, Student, Product, Book ...**
- Do we have to repeat all of the same code again???



My Wish

- I wish we could tell Spring:

Create a REST API for me

Use my existing JpaRepository (entity, primary key)

Give me all of the basic REST API CRUD features for free

Spring Data REST - Solution

- Spring Data REST is the solution!!!! <https://spring.io/projects/spring-data-rest>
- Leverages your existing **JpaRepository**
- Spring will give you a REST CRUD implementation for FREE like MAGIC!!
 - Helps to minimize boiler-plate REST code!!!
 - No new coding required!!!



REST API

- Spring Data REST will expose these endpoints for free!

HTTP Method		CRUD Action
POST	/employees	Create a new employee
GET	/employees	Read a list of employees
GET	/employees/{employeeId}	Read a single employee
PUT	/employees/{employeeId}	Update an existing employee
DELETE	/employees/{employeeId}	Delete an existing employee

Get these REST endpoints for free



Spring Data REST – How Does it work?

- Spring Data REST will scan your project for JpaRepository.
- Expose REST APIs for each entity for your JpaRepository.

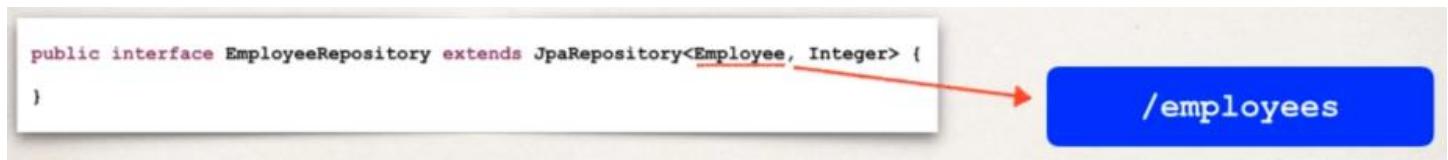
```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {  
}
```

REST Endpoints:

By Default, Spring Data REST will create the endpoints based on entity type.

Simple plurized form

- First Character of Entity type is lowercase
- Then just add an “s” to the entity.



Development Process:

1. Add Spring Data REST to your Maven POM file.

Step 1: Add Spring Data REST to POM file

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

That's it!!!

Absolutely NO CODING required

Spring Data REST will
scan for JpaRepository

HTTP Method	CRUD Action
POST /employees	Create a new employee
GET /employees	Read a list of employees
GET /employees/{employeeId}	Read a single employee
PUT /employees/{employeeId}	Update an existing employee
DELETE /employees/{employeeId}	Delete an existing employee

Get these REST
endpoints for free

Gift

In A Nutshell

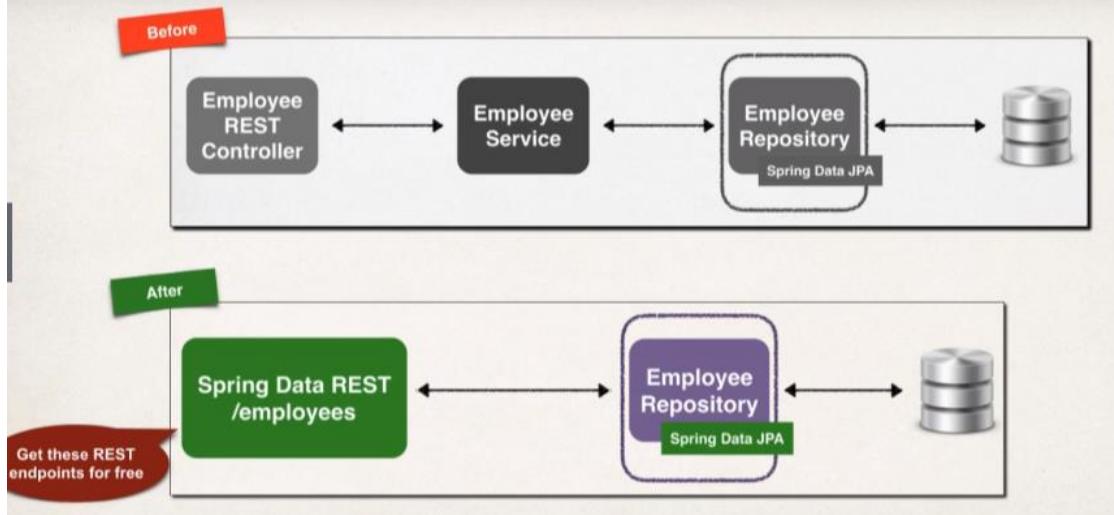
For Spring Data REST, you only need 3 items

We already have these two

1. Your entity: `Employee`
2. JpaRepository: `EmployeeRepository extends JpaRepository`
3. Maven POM dependency for: `spring-boot-starter-data-rest`

Only item that is new

Application Architecture



HATEOAS

- Spring Data REST endpoints are HATEOAS compliant
 - **HATEOAS:** Hypermedia as the Engine of Application State
- Hypermedia-driven sites provide information to access REST interfaces
 - Think of it as meta-data for REST data

<https://spring.io/understanding/HATEOAS>

HATEOAS

Get single employee

- Spring Data REST response using HATEOAS
- For example REST response from: **GET /employees/3**

Response

```
{  
    "firstName": "Avani",  
    "lastName": "Gupta",  
    "email": "avani@luv2code.com",  
    "_links": {  
        "self": {  
            "href": "http://localhost:8080/employees/3"  
        },  
        "employee": {  
            "href": "http://localhost:8080/employees/3"  
        }  
    }  
}
```

Employee data

Response meta-data
Links to data

HATEOAS

Get list of employees

- For a collection, meta-data includes page size, total elements, pages etc
- For example REST response from: **GET /employees**

Response

```
{  
    "_embedded": {  
        "employees": [  
            {  
                "firstName": "Leslie",  
                ...  
            },  
            ...  
        ]  
    },  
    "page": {  
        "size": 20,  
        "totalElements": 5,  
        "totalPages": 1,  
        "number": 0  
    }  
}
```

JSON Array of employees

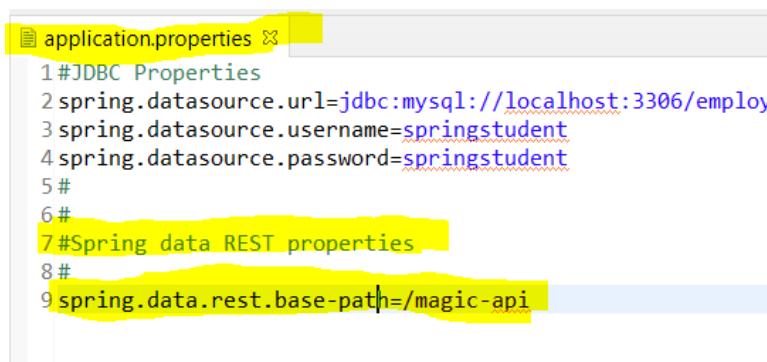
Response meta-data
Information about the page

Advanced Features:

Spring Data REST advanced features

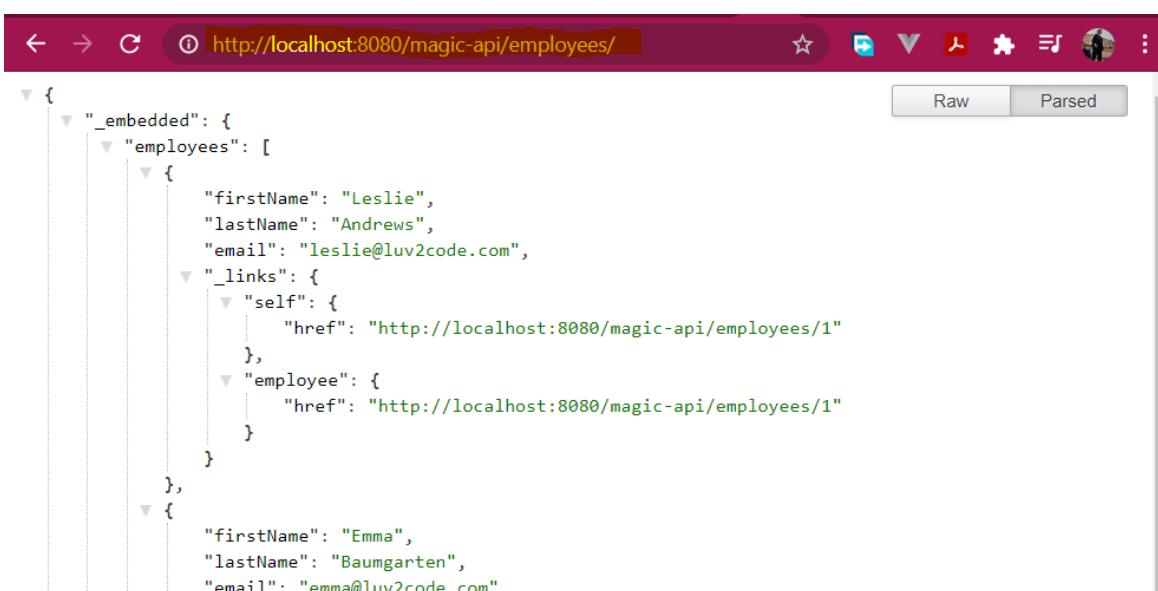
- Pagination, sorting and searching.
- Extending and adding custom queries with JPQL.
- We can also customize the REST APIs by making use of Query Domain Specific Language (Query DSL).

Customize the endpoint base path:



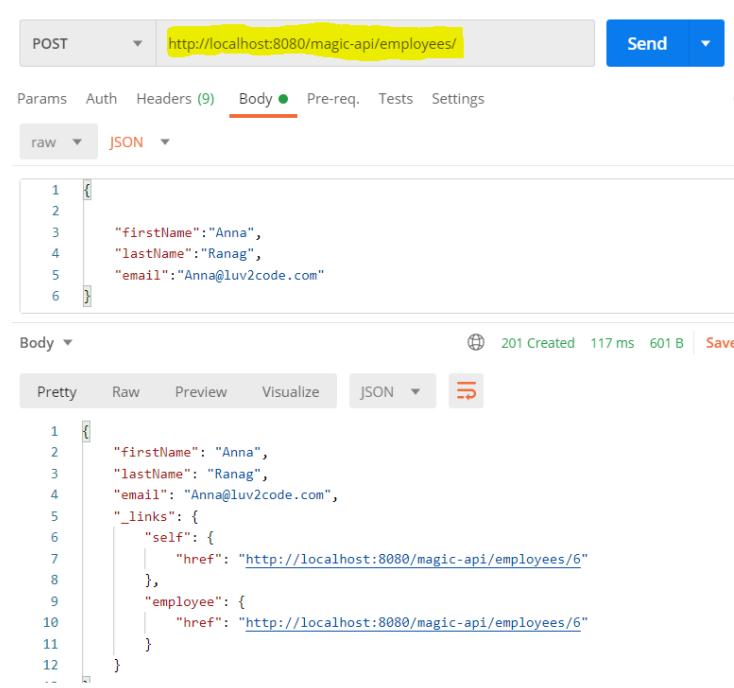
application.properties

```
1 # JDBC Properties
2 spring.datasource.url=jdbc:mysql://localhost:3306/employees
3 spring.datasource.username=springstudent
4 spring.datasource.password=springstudent
5 #
6 #
7 # Spring data REST properties
8 #
9 spring.data.rest.base-path=/magic-api
```



http://localhost:8080/magic-api/employees/1

```
{
  "_embedded": {
    "employees": [
      {
        "firstName": "Leslie",
        "lastName": "Andrews",
        "email": "leslie@luv2code.com",
        "_links": {
          "self": {
            "href": "http://localhost:8080/magic-api/employees/1"
          },
          "employee": {
            "href": "http://localhost:8080/magic-api/employees/1"
          }
        }
      },
      {
        "firstName": "Emma",
        "lastName": "Baumgarten",
        "email": "emma@luv2code.com",
        "_links": {
          "self": {
            "href": "http://localhost:8080/magic-api/employees/2"
          },
          "employee": {
            "href": "http://localhost:8080/magic-api/employees/2"
          }
        }
      }
    ]
  }
}
```



POST http://localhost:8080/magic-api/employees/1 Send

Params Auth Headers (9) Body Pre-request Tests Settings

raw JSON

```
{
  "firstName": "Anna",
  "lastName": "Ranag",
  "email": "Anna@luv2code.com"
}
```

Body 201 Created 117 ms 601 B Save

Pretty Raw Preview Visualize JSON

```

1 {
  "firstName": "Anna",
  "lastName": "Ranag",
  "email": "Anna@luv2code.com",
  "_links": {
    "self": {
      "href": "http://localhost:8080/magic-api/employees/6"
    },
    "employee": {
      "href": "http://localhost:8080/magic-api/employees/6"
    }
  }
}
```

Spring Data REST Configuration, Pagination and Sorting:

Pluralized Form

- Spring Data REST pluralized form is VERY simple
 - Just adds an "s" to the entity
- The English language is VERY complex!
 - Spring Data REST does NOT handle

Singular	Plural
Goose	Geese
Person	People
Syllabus	Syllabi
...	...



- Spring Data REST does not handle complex pluralized forms.
 - In this case we need to specify plural name.
- What if we want to expose a different resource name?
 - Instead of /employees, we want to expose /members

Solution:

- Specify plural name / path with an annotation

```
@RepositoryRestResource(path="members")
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
}
```

<http://localhost:8080/members>

Pagination

- By default, Spring Data REST will return the first 20 elements
 - Page size = 20
- You can navigate to the different pages of data using query param

<http://localhost:8080/employees?page=0>

<http://localhost:8080/employees?page=1>

...

Pages are zero-based

Spring Data REST Configuration

- Following properties available: application.properties

Name	Description
spring.data.rest.base-path	Base path used to expose repository resources
spring.data.rest.default-page-size	Default size of pages
spring.data.rest.max-page-size	Maximum size of pages
...	...

More properties available

www.luv2code.com/spring-boot-props

spring.data.rest.*

Sample Configuration

File: application.properties

spring.data.rest.base-path=/magic-api

spring.data.rest.default-page-size=50

http://localhost:8080/magic-api/employees

Returns 50
elements per page

Sorting

- You can sort by the property names of your entity
 - In our Employee example, we have: **firstName**, **lastName** and **email**
- Sort by last name (ascending is default)
<http://localhost:8080/employees?sort=lastName>
- Sort by first name, descending
<http://localhost:8080/employees?sort=firstName,desc>
- Sort by last name, then first name, ascending
<http://localhost:8080/employees?sort=lastName,firstName,asc>

