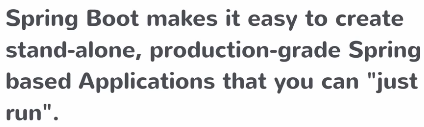
**Spring Boot Basics**

**What is Spring Boot and why to use it?**

Spring is a framework which lets you write enterprise java applications. It’s a huge framework with lots of different features. Boot is bootstrap. Spring boot is what lets you bootstrap a spring application from scratch.

**Spring Boot** is a tool which let you **create** **Spring based Applications**.

Standalone: runs on its own. Production-grade: not a basic app, something you can deploy to prod. Something that you can “just run”: If you created a spring app without spring boot, you will understand the pain of having to do a lot of stuff to get to the point where you just run it. A lot of configuration steps that needs to be done, jars to import etc. Only then you can run it. But spring boot makes it easy.

**What is Spring?**

Spring is much more than just a dependency injection framework! It is a whole **application framework**. It has its origins from DE but does a whole lot more. Let’s you write enterprise java applications.

The underlying concepts of all the 255 gazillion enterprise java applications have a lot of similarities. For example, when you write a business service, you do similar things for every business service no matter what the actual domain is. If you want to make a transaction, then you need to connect to a database etc. Common problems for every enterprise application. Spring provides a solution, a template, a framework for those common problems which lets you build those enterprise applications.

It also has a **programming and configuration model**. The cool thing with Spring is you only focus on building your business services you let spring handle a lot of these common concerns. Connection to db, running queries, handling http request with an mvc layers. This is usually done by annotation your classes, telling spring what to do with them. So, you focus on your business problem and let spring do the rest.

**Infrastructure support** for different stuff. For example different databases etc.

**Problems with Spring**

* **Huge Framework**: so many different things and spring tries to address all those different ways, all configurations and combinations of technologies.
* **Multiple setup/config steps**: spring can connect to MongoDB, RDBMS, … Since it does a whole lot, it needs a whole lot of configuration for it to do exactly what we need to do.
* **Multiple build and deploy steps**: Again, various configurations. Lot of capabilities and flexibility comes with a cost. Again, need to tell it exactly what you need. Since it can do a whole lot, you don’t have a starting point or a best practice way. No pathway figure it on your own.

This is where **Spring Boot** comes in. Abstracting away all these infrastructure/setup/configuration concerns. So, you can just focus on what you want to build. There are 100 ways of building a Spring app but we want someone to tell us “This is the best way for 80% of the cases and for the rest 20% you need to configure it yourself”. This is what Spring boot does.

**Spring Boot is**:

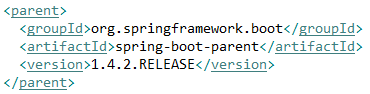
* **Opinionated**: It makes decisions for you. “Start with this, then later if you wish change it”
* **Convention over configuration**: If you belong in the %80, no configuration necessary
* **Stand alone**: You can generate a Spring application using Spring Boot. This will be a stand-alone application. Typically, when you build a Spring app, it will be a .war file, a web application that you deploy to a tomcat (or any other) servlet container. With Spring Boot what you get is a **stand-alone application.** Something you can just run and have it start a web server. No need to find a servlet container to deploy to.
* **Production ready**: as we said, you don’t have to do something extra to get it ready for production.

**Dev Env**

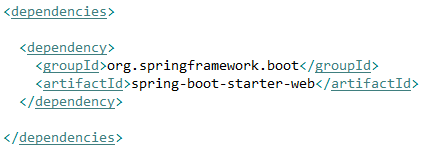
* **Spring STS (Spring Tool Suite)**: is a flavor of Eclipse, tweaked to work with Spring applications. With some Spring specific features.
* **Java 8+**

Starting a simple project with a maven archetype (template).

**We created a simple maven project. What do I have to do to make this a Spring Project?**

****

**Step 1.** Adding Spring Boot to our maven pom.xml as the parent. As we said that Spring Boot makes use of “convention over configuration”. So what the Spring team did was they created a **spring-boot-parent** project. They put all the default maven configs there, which contains all the opinionated set of maven configurations. So all we need to do declare that project as a parent and we will **inherit those configurations**.

**Step 2.** Declare dependencies. We are creating a rest web application. We need to import some jars to be able to build our app. Normally we would identify all the dependencies and add them to the list one by one. But here is where Spring Boot helps us again. Since they know that every web app needs certain jars, they created a **meta-dependency** (smth like a **parent-dependency**) which will pull all the dependencies. All we need to do is establish one dependency to that meta-dependency.



* Added version info. **Maven -> Update Project**, this updated also the JRE library version automatically.

A picture containing graphical user interface

Description automatically generated**Spring-booth-starter-web:**

Graphical user interface, text, application, email

Description automatically generated**Spring-booth-starter-parent**:

**Compile Dependencies vs Managed Dependencies**

Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.

**Text

Description automatically generatedGraphical user interface, text, application

Description automatically generated**

Text

Description automatically generated**What is happening here?**

* The **dependencies** sections tells maven what jars to download.
* Text

  Description automatically generatedThe **parent** sections configures which versions of those jars to download.

Text

Description automatically generatedText

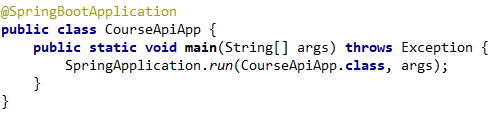
Description automatically generated

Text

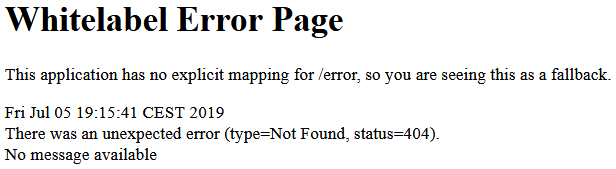
Description automatically generated

**Bill of materials**: This preset list of possible of combinations of jars that work well together without issues is what is called Bill of materials (what Spring calls it). You know that there are a certain combination of jars and their versions that work well, its approved by spring boot. So all you need to do is to pick that version of the list and you get all the preset values with it. You don’t have to worry about individual version numbers anymore, just worry about the version of the parent. The parent itself instructs maven what are the different combinations.

**Create a new class with a main method**. We already learned that our app will be stand alone. Meaning that it needs to start like a normal java program.

1. **Annotate**: **@SpringBootApplication** tells that this is our starting point for our application
2. Tell Spring Boot to start this application (CourseApiApp), create a servlet container and host this application in that container and make it available. 1- Class which you annotated, 2- args

Now when we **Run -> as a Java Application**, our first Spring app has started!



**Starting Spring Boot**

* **Sets up default configuration**
* **Starts Spring application context**: Spring is in a sense a container for all your logic/services you wrote. This container is what we call an application context. Every spring app has it, which runs when the app runs.
* **Performs class path scan**: The way to plug-in code into Spring Boot is by creating your own custom classes and annotating them with the intert. Lets say we want to create a business service. You create a class and annotate it with @Service, a controller @Contoller. Marking our classes, what they are. Spring will look at all those markers and treats your classes differently. In order to know which files you marked it has to scan the classpath to identify all the classes we annotated, on startup.
* **Starts Tomcat server**: Tomcat came with Spring Boot. This is why its a stand alone app.

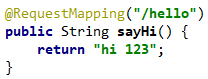
**Let’s Add a Controller**

We want to handle different requests. These are done with controllers, a class with a certain annotation. These annotations tell:

* What URL access triggers it?
* What method to run when accessed?

The web layer in Spring Boot application leverages a framework called Spring MVC. What it does it lets you build server side code which maps to URLs and provides responses. The response can be XML/JSON (for rest) or a full HTML page, a request map to JSP, FTL (FreeMarker Template) response.

We will create a package **xxx.hello** and create a new class **HelloController**. This Indicates a rest controller and its an annotation from spring MVC. A rest controller means that we can have methods which maps to requests.

**@RestController** is a convenience annotation that combines **@Controller** and **@ResponseBody** – which eliminates the need to annotate every request handling method of the controller class with the @ResponseBody annotation

The **@RequestMapping** **maps every http method to this function**. So even a **DELETE** to .../hello will trigger this and returns “hi 123”.

**Better to always give method name**:

**@GetMapping** is a composed annotation that acts as a shortcut for   
**@RequestMapping**(method = RequestMethod.GET

**@Controller vs @RestController**

Text

Description automatically generatedGraphical user interface, text, application, website

Description automatically generated

Graphical user interface, text

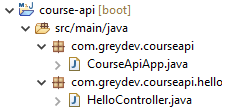
Description automatically generatedGraphical user interface, text

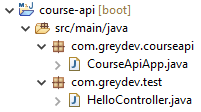
Description automatically generated

The **@ResponseBody** is activated by default. You don't need to add it above the function signature.

**@ResponseBody** means that the returned value of the method will constitute the body of the HTTP response. Of course, an HTTP response can't contain Java objects. So this list of accounts is transformed to a format suitable for REST applications, typically JSON or XML. Meaning that your response will be converted to JSON/XML automatically.

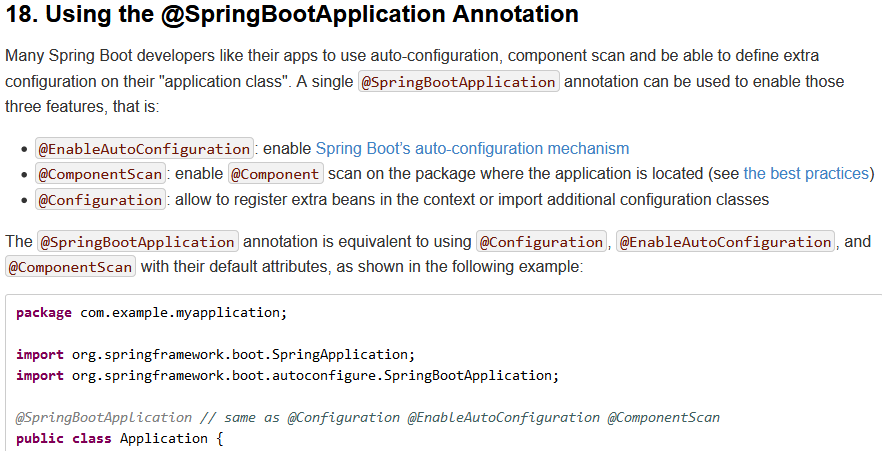
* **Don’t forget to add a default constructor (== with no args) to your model classes**. Because empty objects are first created and then populated with the values.
* **TODO**: What does @ResponseBody and @RequestBody do? <https://www.baeldung.com/spring-request-response-body>

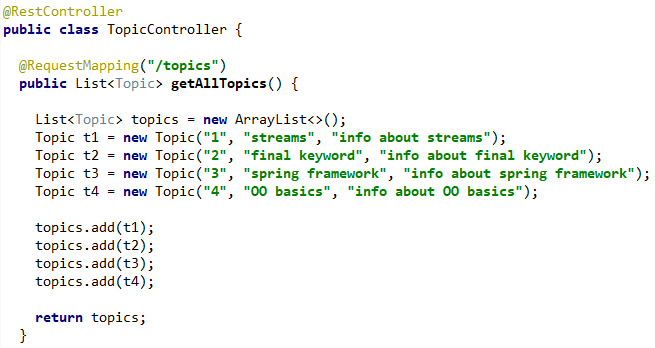
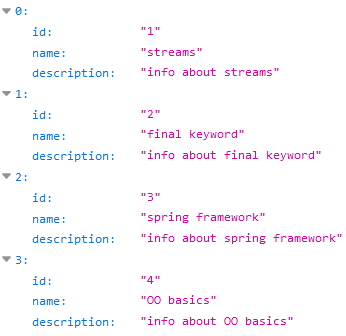
**Important**: Controller will only work if it is the part of main class. Example : if main class is in 'package1' and controller is in 'package2', then it won't work.. so controller must be in 'package1.anyname'

Because **@SpringBootApplication** groups three annotations

* @EnableAutoConfiguration,
* @Configuration and
* @ComponentScan.

**@ComponentScan** is responsible to scan the package where the application is located in order to find all Components.





* Spring MVC does the JSON conversion from List<Topic> to JSON.
* The generated JSON has key name and value corresponding to property names and values of the Topic class

**Embedded Tomcat Server**

The reason Spring Boot decided to have tomcat embedded is:

* **Convenience because stand alone**: just runs, no need to download tomcat.
* **There are some servlet container config steps that needs to happen**. When you need to deploy something, you may need to configure Tomcat. But now, the servlet container config will be a part of the **application** **config**. Just like you configure other things about your application in your source code, you also have tomcat related configs, **everything is in one package**. (meaning all can be configured in the same place, with the same style of configuration)
* **Useful for microservice architecture**. When you have a bunch of microservices, you don’t want to have additional steps in order to deploy each microservice. If you have 10 microservices, you don’t want to deploy them 10 times.

You can choose any other servlet container. Need to add it in your pom.xml as a dependency and configure it.

**How Spring MVC Works?**

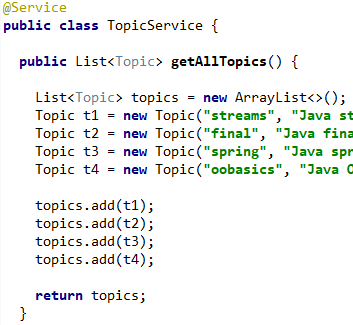
The View Tier is handled by the framework Spring MVC, which is another project under the Spring umbrella. Spring MVC lets you build controllers which maps requests to responses. We had an app, thanks to the spring-boot-starter-web dependency, we added Spring MVC to our app.

These are simple Java classes which maps a **URI** and **HTTP Method** to some **functionality**. In our project it detected a @RestController and it returned a JSON response, did the conversion for us (with Moxy?).

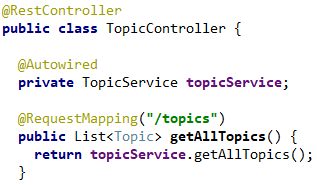
* Again **Spring MVC** will do a **class path scan** to see if there are any classes annotated for different things (controllers, services, etc).
* We will create **.../topics/id/courses/x/lessons/x**

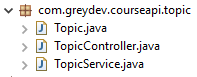
**Business Services**

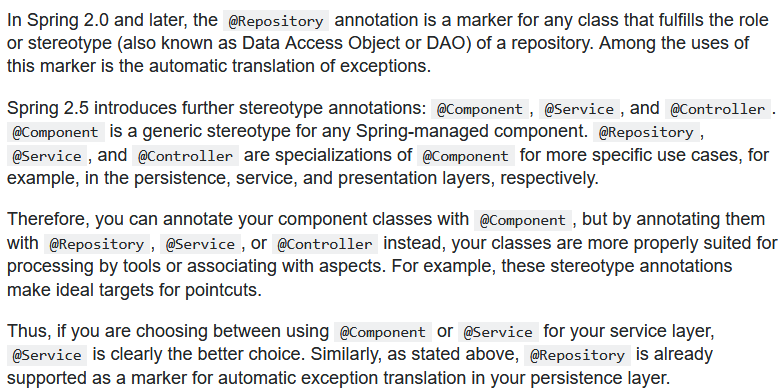
Business services are **typically singletons**. When the app starts up Spring creates an instance of this service and then it keeps that in its memory/registers it. Other controllers and classes will be dependent on that and spring knows that and will **inject** it to those different classes.

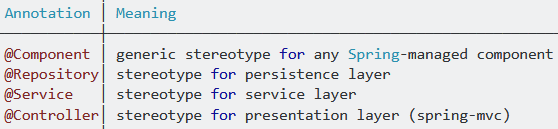
Mark a class as a Spring Business Service with **@Service** – Steriotype Annotation

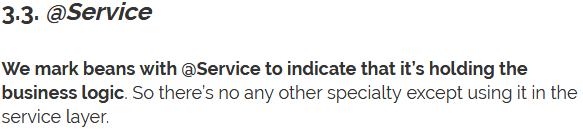
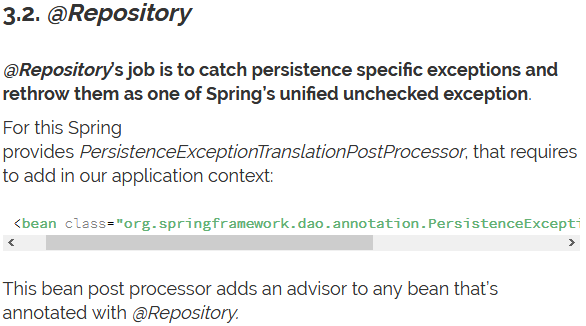
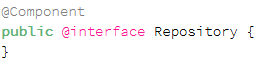
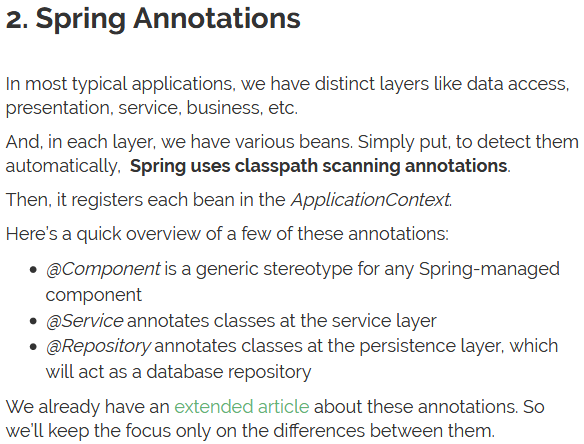
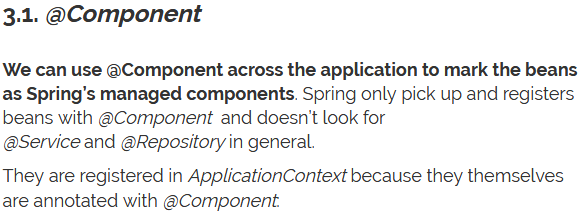
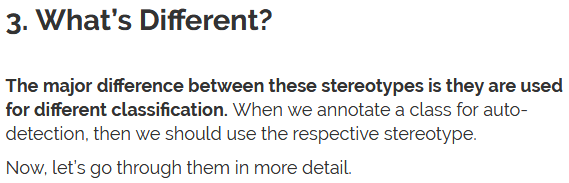
* In any other class, use **@Autowired** to get the instance. Spring will **inject** the singleton instance.

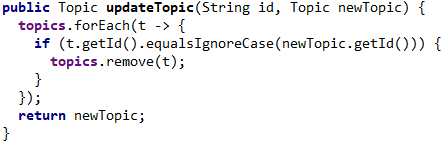
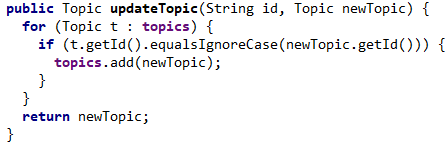


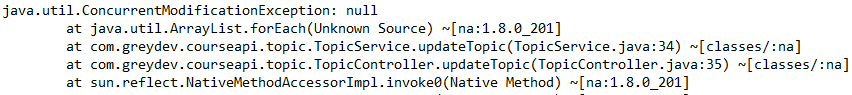


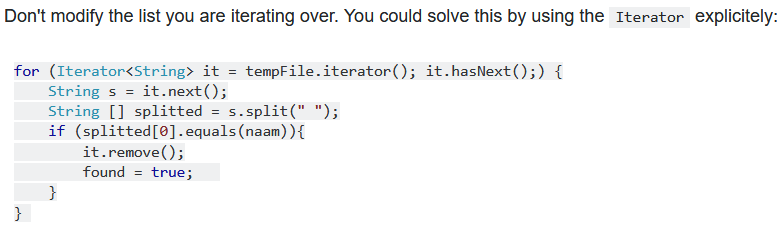
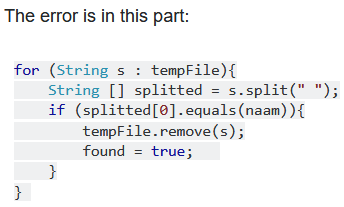


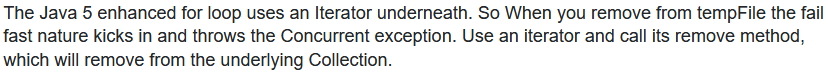


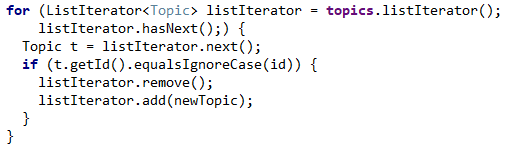
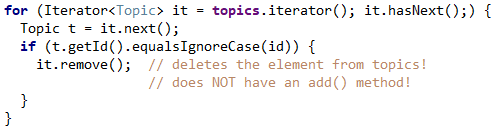


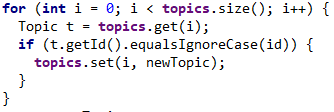


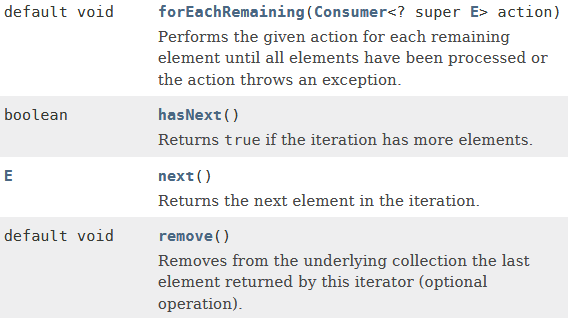
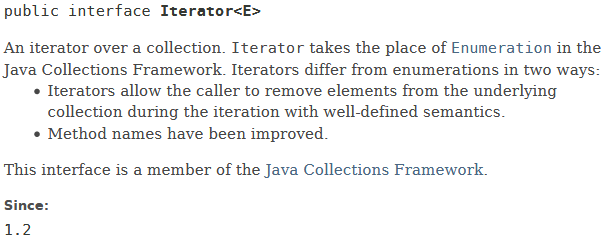


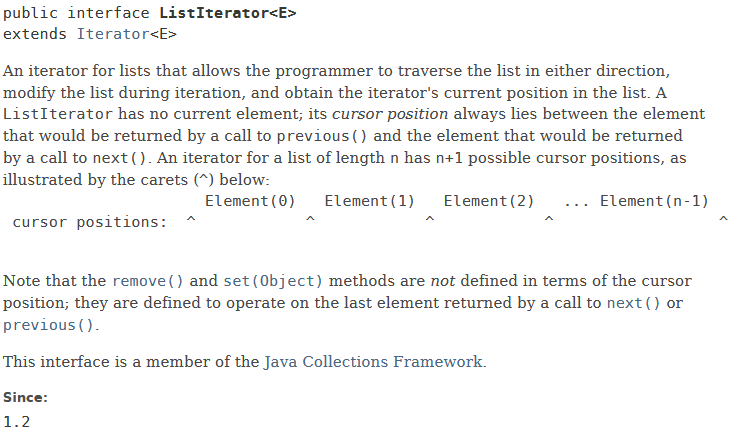


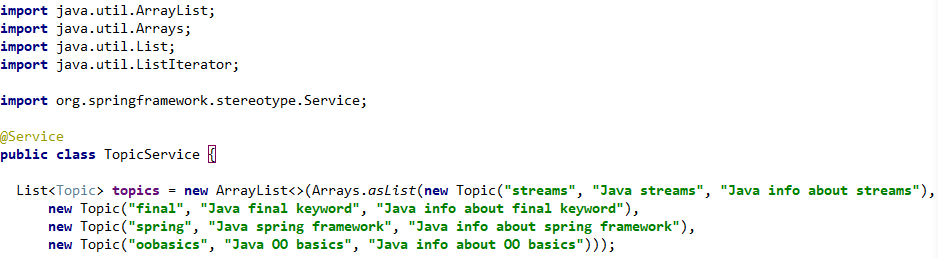


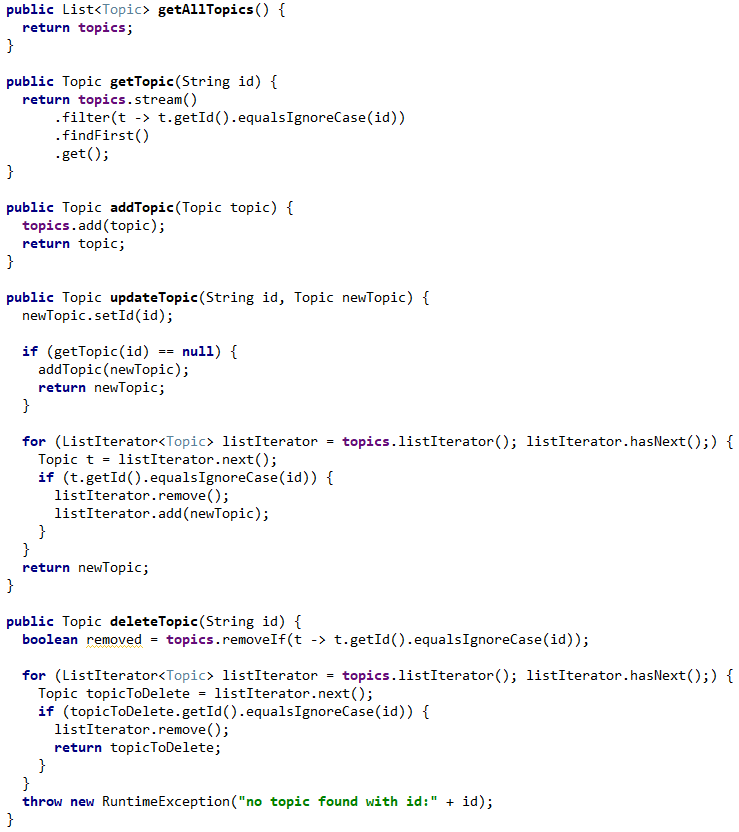


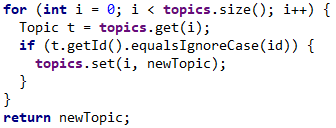












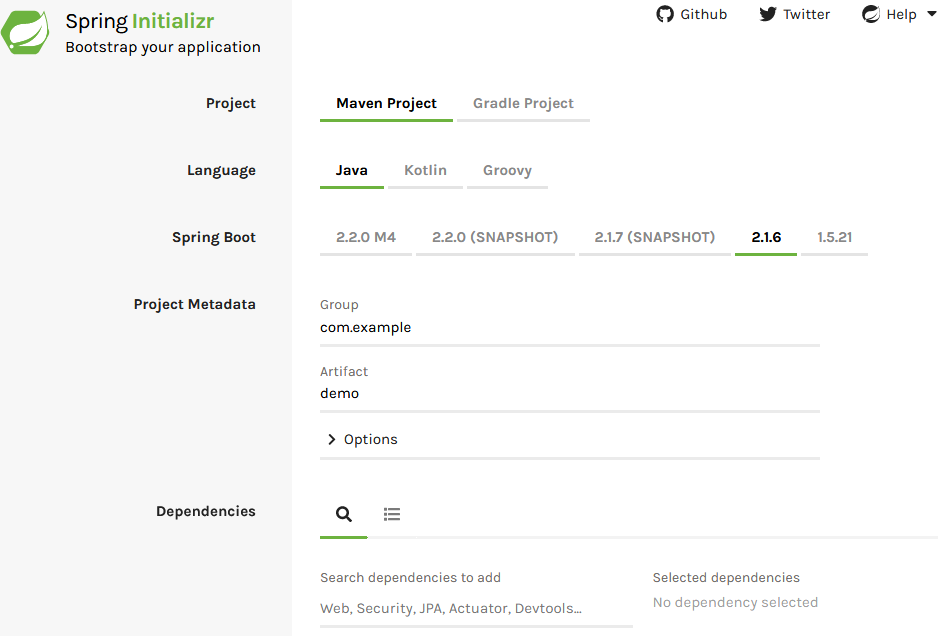
**Booting Spring Boot**

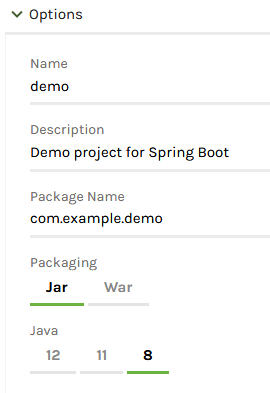
We already saw how we can create a Spring Boot application with maven. We will see some other ways as well.

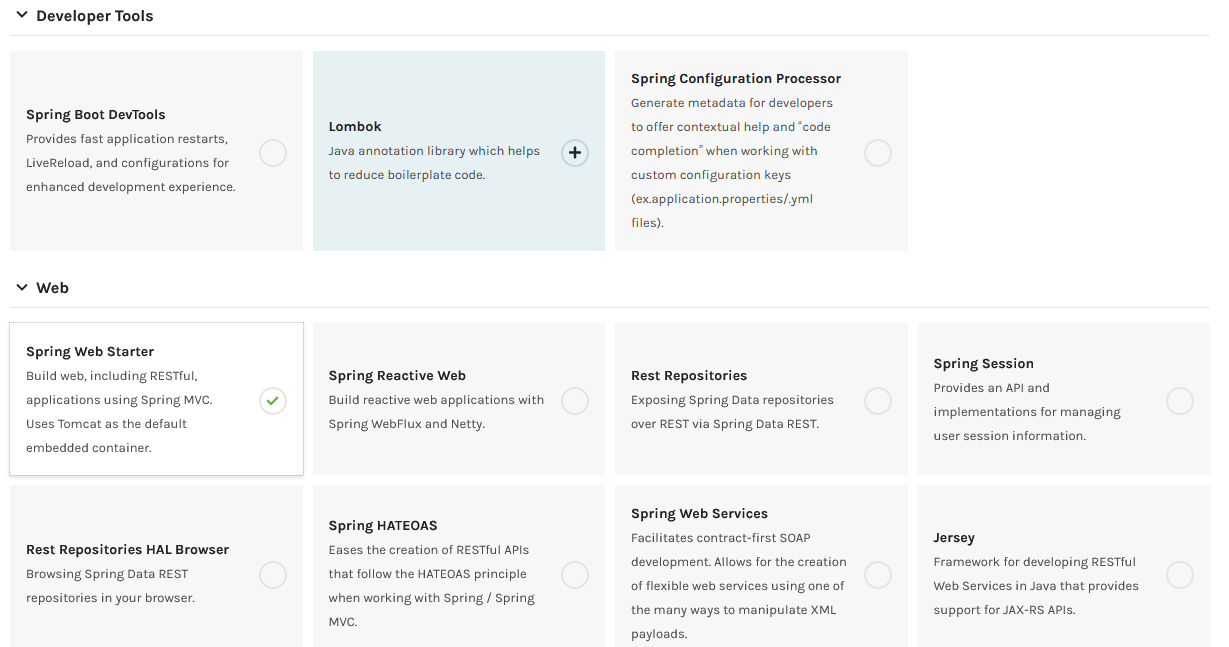
* **Plain Maven project**: add all the necessary things to your project to make it a spring boot app.
* **Spring Initializr**
* **Spring Boot CLI**
* **STS IDE / Eclipse + Spring Devtools Plugin**

**Spring Initialzr**

Web UI under **start.spring.io**. You choose your configurations and then download the project.













**...**

**Spring Boot CLI**

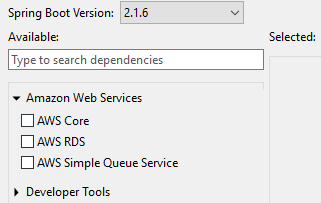
The Spring Boot CLI (Command Line Interface) is a command line tool that you can use to **quickly prototype** with Spring. It lets you **run Groovy scripts**, which means that you have a familiar Java-like syntax without so much boilerplate code.

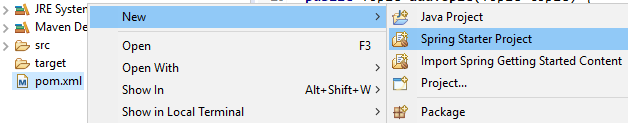
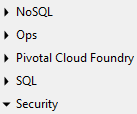
You do not need to use the CLI to work with Spring Boot, but it is definitely the quickest way to get a Spring application off the ground.

This is not something you will typically use for developing production things. Just for quick prototyping.

You can write a rest controller class with groovy and then start your app. The CLI tool will do everything necessary for your program to run and return a “hello world” from your controller your wrote. Without you needing to configure something.

**STS IDE / Eclipse + Spring Devtools Plugin**

Using the IDE itself is typically what you will do. The easiest way.

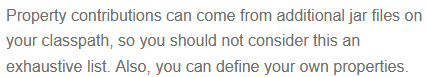
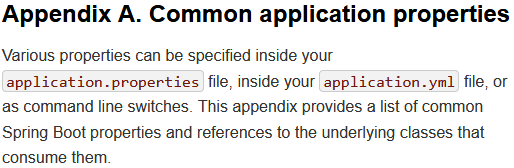


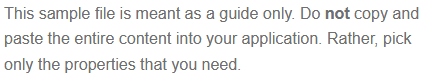
**Configure Spring Boot**

80% use case might not always apply to our cases. We might want to tweak some properties.

There are multiple ways to do it. One way (typical, and easiest) way is to use a **.properties** file to overwrite the default properties.

The server starts at port 8080 by default. We can change that. Add a new properties files under **src/main/resources** and it **must** have the name **application.properties**







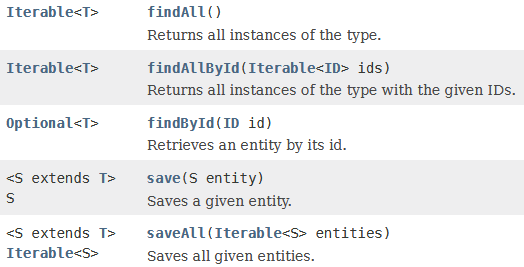
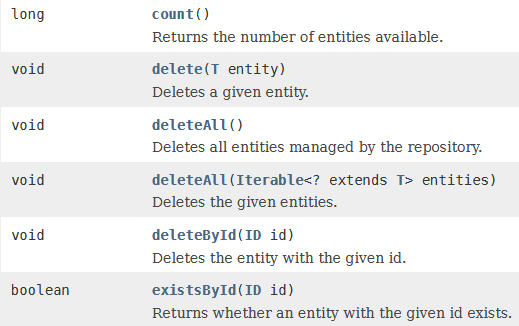
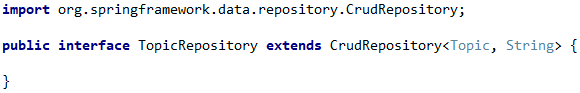
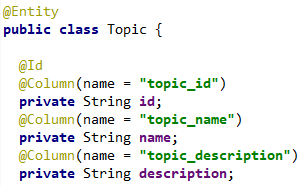
**Spring Data JPA: The Data Tier**

**Spring Data JPA** provides implementations for commonly used **CRUD** operations and makes working with these really easy. You don’t need to implement each of these yourself like you would if you would work with Hibernate. Spring data JPA is one of many possible choices to access a DB. You could use plain **Spring JDBC** if you wanted to.

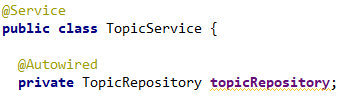


We added 2 dependencies.

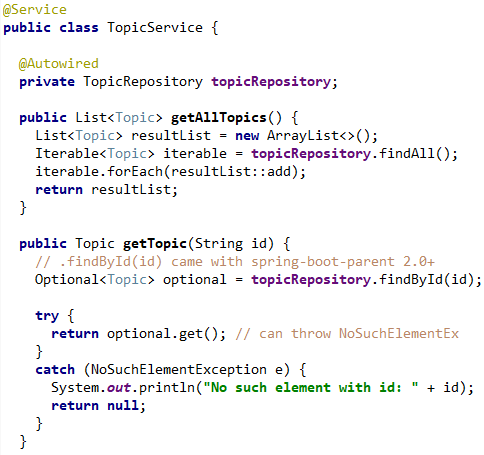
* SQL -> JPA (Spring Data JPA)
* An **in-memory, embedded database,** Apache Derby



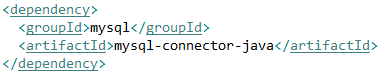




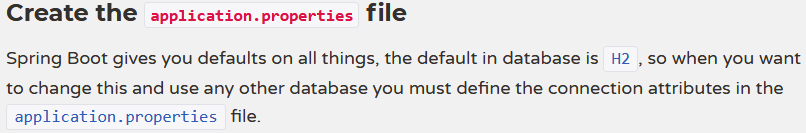
* The frameworks sees the embedded Derby database in the classpath and assumes it needs to connect to that. No connection information is necessary.

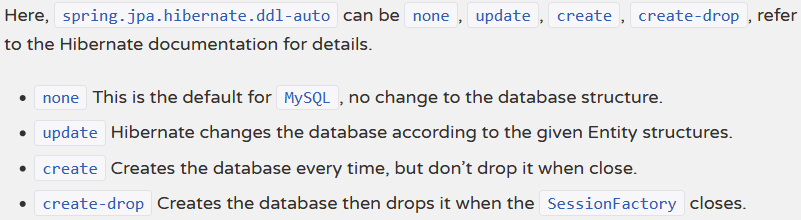


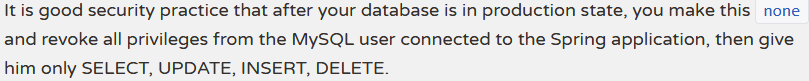
**Connecting to an External Database – MySQL**

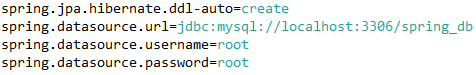


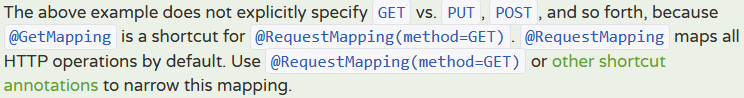










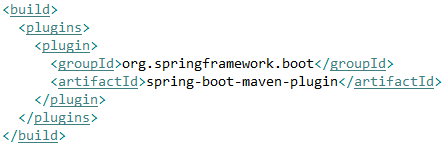


org.springframework.jdbc.support.**MetaDataAccessException**: Could not get Connection for extracting meta-data; nested exception is org.springframework.jdbc.CannotGetJdbcConnectionException: Failed to obtain JDBC Connection; nested exception is java.sql.SQLException: **The server time zone value 'Mitteleurop?ische Sommerzeit' is unrecognized or represents more than one time zone. You must configure either the server or JDBC driver (via the serverTimezone configuration property) to use a more specifc time zone value if you want to utilize time zone support**

* **spring.datasource.url**=jdbc:mysql://localhost:3306/spring\_db?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC

Now it works as expected.

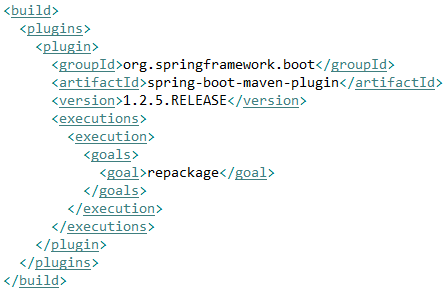
**Build and Run with Maven**



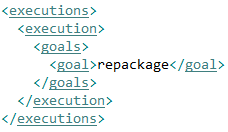
* **Adding the following build config as suggested from the official spring website, always gave me this error**:







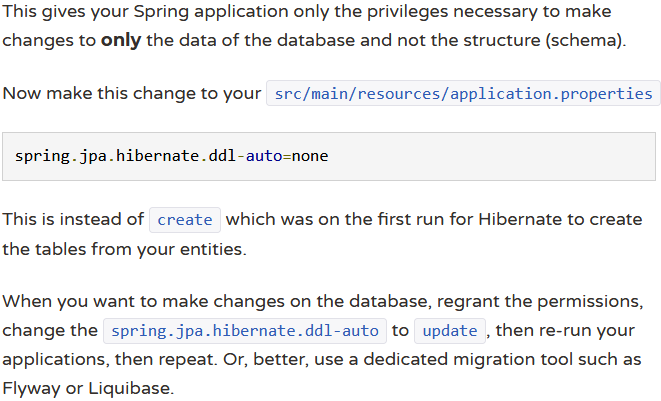
* **ADDING THIS MAKES IT WORK, NOW IT CAN FIND THE MAIN CLASS AND START THE APP NORMALLY**!!!

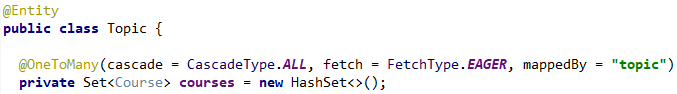


* Use to directly build and run you app:









<http://localhost:8080/topics/j2e/courses/hibernate> DELETE, doesn’t work.

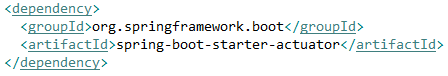
 Works like this, why?

* Update topic with a course id existing in different topic?
* Why returning ‘topicId’ instead of ‘parentTopicID’ like I wrote?

**Spring Boot Actuator**

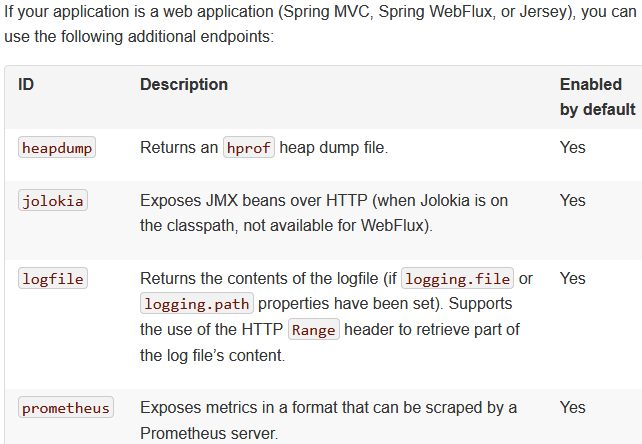
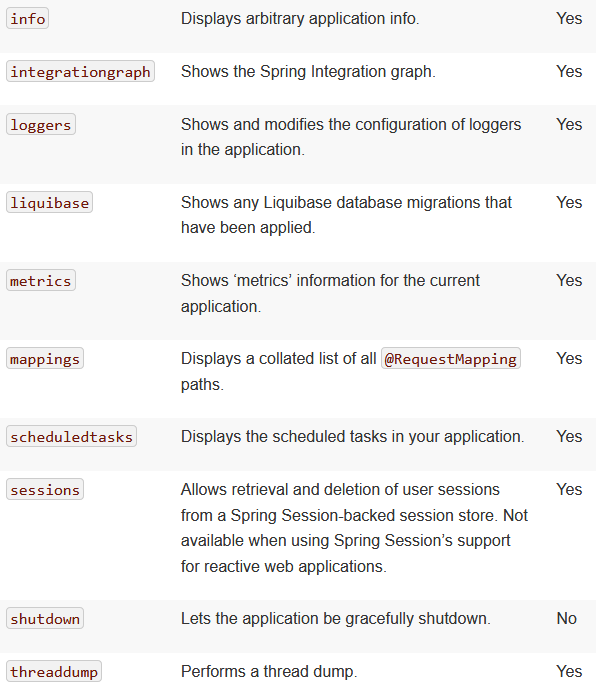
Spring Boot includes a number of additional features to help you monitor and manage your application when you push it to production. You can choose to manage and monitor your application by using HTTP endpoints or with JMX. Auditing, health, and metrics gathering can also be automatically applied to your application.

The spring-boot-actuator module provides all of Spring Boot’s production-ready features. The simplest way to enable the features is to add a dependency to the spring-boot-starter-actuator ‘Starter’.

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the health endpoint provides basic application health information.

Each individual endpoint can be enabled or disabled. This controls whether or not the endpoint is created and its bean exists in the application context. To be remotely accessible an endpoint also has to be exposed via JMX or HTTP. Most applications choose HTTP, where the ID of the endpoint along with a prefix of /actuator is mapped to a URL. For example, by default, the health endpoint is mapped to /actuator/health.

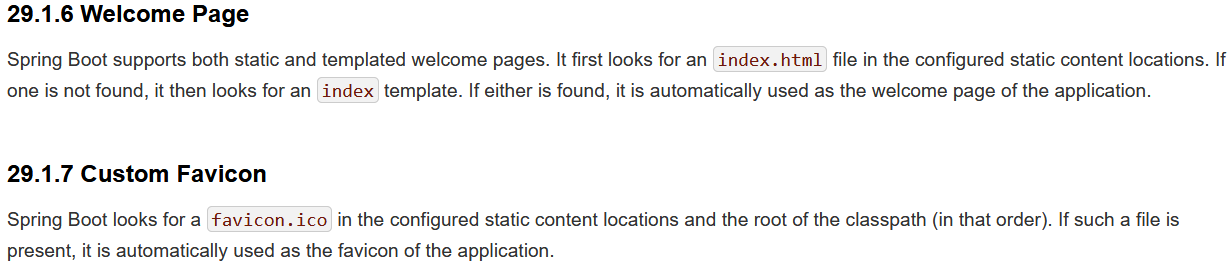




**Static Content**

By default, Spring Boot serves static content (images, fonts, style sheets, js, etc.) from a directory called **/static** (or **/public** or **/resources** or **/META-INF/resources**) in the classpath or from the root of the ServletContext. It uses the ResourceHttpRequestHandler from Spring MVC so that you can modify that behavior by adding your own WebMvcConfigurer and overriding the addResourceHandlers method…

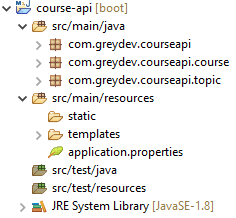
* **Do not use** the **src/main/webapp** directory **if your application is packaged as a jar**. Although this directory is a common standard, it works only with **war packaging**, and it is silently ignored by most build tools if you generate a jar.



**Template Engines**

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

If possible, **JSPs should be avoided**. There are several known limitations when using them with embedded servlet containers.

* When you use one of these templating engines with the default configuration, **your templates are picked up automatically from src/main/resources/templates**.

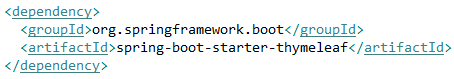
**Thymeleaf**

Thymeleaf is a template engine that can serve views of your web application in **online and offline** scenarios.

Offline meaning, you don’t need any back-end component to be active or involved in order to show the template correctly. You can just open the file with a browser and it will look the same if it were in production, sent by the server. This is not the case i.e. with jsp files. If you try to open jsp files directly it will look horrible.

This makes it convenient when working with designers or frontend devs. They can create their style sheets and html without depending on server side stuff, no need to install java, database, tomcat etc.

* Supports view composition: creating a part of the view and then reusing it again in different parts.

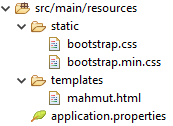
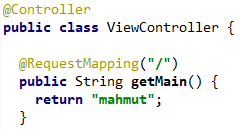




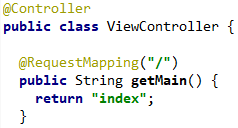
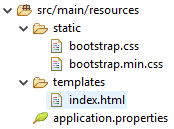
Created a simple **index.html** file under **templates**.

Created a class called **ViewController** which will return this template

**You need to return a string with the name of the template**:









**Important**:

We wrote href="../static/bootstrap.css", meaning that we defined a path to our css files. But when we open our site, we can see that the file couldn’t be fetched from the server.

Because this href basically uses a relative path to find the file on the server. But this does not work inside a server context, in online mode.

* However if we open this file with a browser (offline mode), it will work as expected.

To make it work in the online mode we need a different attribute name.

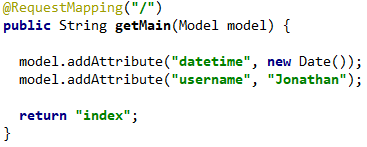
* **@** means the **static folder** in online mode.

In **offline mode** the **th:href** attribute is ignored by the browser, meaning the normal **href** will be used.

In **online mode**, the **th:href** will be used to replace the **href** value

* This dual way of doing things is typical for Thymeleaf. This will be used in a lot of different things.

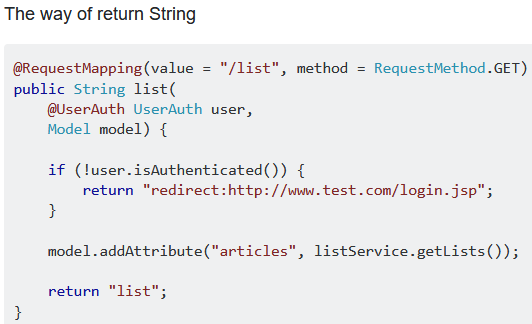
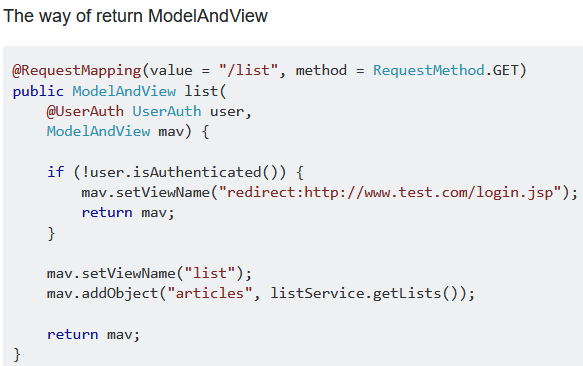


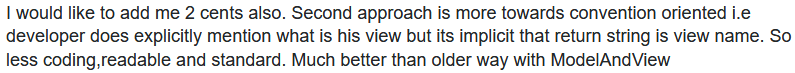


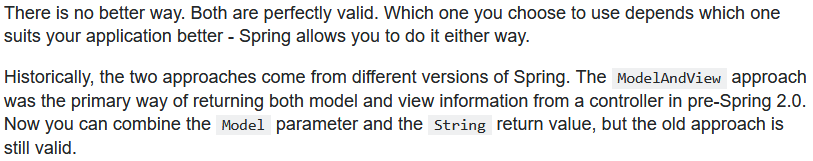
Right now my index template is binded with the model that I took as the argument (injected). So I can add an attribute to my ‘index’ model. I think the model object that we get is representing only the index template. If I try to access ‘username’ from another template, it is not working.

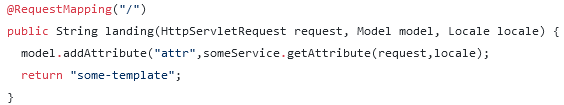


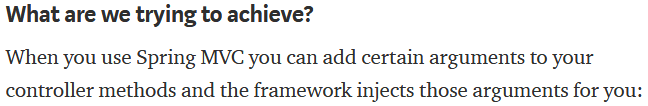
There are different ways a controller class can return a view (both methods get parameters injected):











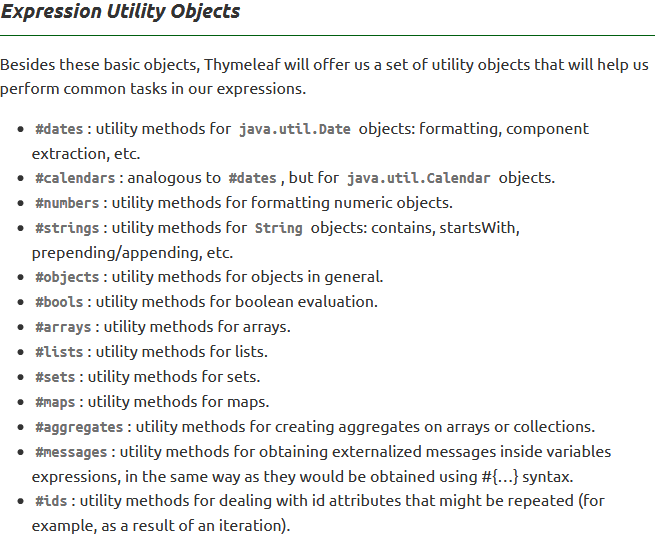
**Variables**

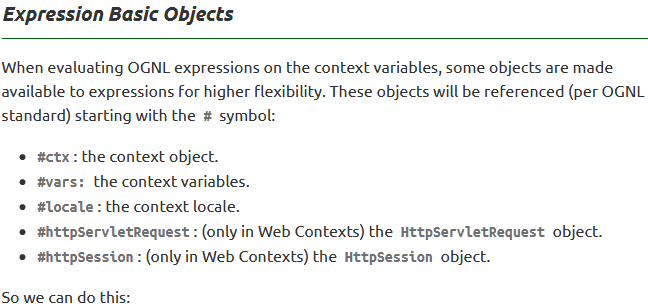
We already mentioned that ${...} expressions are in fact **OGNL (Object-Graph Navigation Language)** expressions executed on the map of variables contained in the context.

From OGNL’s syntax, we know that this:

**<p>Today is: <span th:text="${today}">13 february 2011</span>.</p>**

…is in fact equivalent to this: **ctx.getVariables().get("today");**





Established locale country:

<span th:text="${#locale.country}">US</span>.



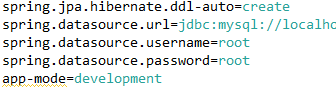
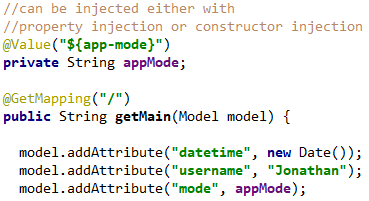
You can use the utility methods to format the date.

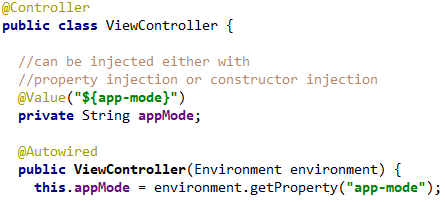
**Conditional logic**



**Configuration**

Our main goal is to change our application properties without needing to restart our server.





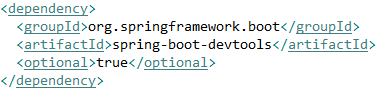
* In a real world application, constructor injection would be more common and maybe better

**Another way to do it is from the command line**:

java -jar demo.jar -Dserver.port:8080 -Dapp-mode:production

* You can also define multiple ‘profiles’, application.properties files named differently and at runtime decide which properties file to read.

**Hot Reload**

Spring Dev Tools, modifications to view and java classes will be live without (usually) a hard refresh of the server.

You don’t need to worry how this dependency will behave in production. This will work as long as you start your application in a debugging environment. If you start it in the IDE, the devtools will be enabled. However with java -jar, then spring will consider it production mode and devtools won’t start.

* In Eclipse: **Build -> Build Automatically** must be selected.
* Now after a change in a **view**, a page refresh will show the changes immediately. Changing a **java class** (and saving it) automatically trigger a server refresh but its not a cold refresh, takes not much time.

