

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 makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

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**.



```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>
```

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.

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

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

```
<properties>
  <java.version>1.8</java.version>
</properties>
```

>  JRE System Library [JavaSE-1.8]
>  Maven Dependencies

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

```
@SpringBootApplication
public class CourseApiApp {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(CourseApiApp.class, args);
    }
}
```

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

```
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
com.greydev.courseapi.CourseApiApp       : Started CourseApiApp in 4.593 seconds (JVM running for 4.966)
```

Whitelabel Error Page

localhost:8080

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

Fri Jul 05 19:15:41 CEST 2019

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

No message available

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
public class HelloController {

    import org.springframework.web.bind.annotation.RestController;
```

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

```
@RequestMapping("/hello")
public String sayHi() {
    return "hi 123";
}
```

Better to always give method name:

```
@RequestMapping(method = RequestMethod.GET, value = "/hello")
```

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'

```
✓ course-api [boot] Works
  ✓ src/main/java
    ✓ com.greydev.courseapi
      > CourseApiApp.java
    ✓ com.greydev.courseapi.hello
      > HelloController.java
```

Because **@SpringBootApplication** groups three annotations

- @EnableAutoConfiguration,
- @Configuration and
- @ComponentScan.

```
✓ course-api [boot] Won't work
  ✓ src/main/java
    ✓ com.greydev.courseapi
      > CourseApiApp.java
    ✓ com.greydev.test
      > HelloController.java
```

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

18. Using the @SpringBootApplication Annotation

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single **@SpringBootApplication** annotation can be used to enable those three features, that is:

- **@EnableAutoConfiguration**: enable [Spring Boot's auto-configuration mechanism](#)
- **@ComponentScan**: enable **@Component** scan on the package where the application is located (see [the best practices](#))
- **@Configuration**: allow to register extra beans in the context or import additional configuration classes

The **@SpringBootApplication** annotation is equivalent to using **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan** with their default attributes, as shown in the following example:

```
package com.example.myapplication;

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

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {
```

```

@RestController
public class TopicController {

    @RequestMapping("/topics")
    public List<Topic> getAllTopics() {

        List<Topic> topics = new ArrayList<>();
        Topic t1 = new Topic("1", "streams", "info about streams");
        Topic t2 = new Topic("2", "final keyword", "info about final keyword");
        Topic t3 = new Topic("3", "spring framework", "info about spring framework");
        Topic t4 = new Topic("4", "OO basics", "info about OO basics");

        topics.add(t1);
        topics.add(t2);
        topics.add(t3);
        topics.add(t4);

        return topics;
    }
}

```

```

▼ 0:
  id:      "1"
  name:    "streams"
  description: "info about streams"
▼ 1:
  id:      "2"
  name:    "final keyword"
  description: "info about final keyword"
▼ 2:
  id:      "3"
  name:    "spring framework"
  description: "info about spring framework"
▼ 3:
  id:      "4"
  name:    "OO basics"
  description: "info about OO basics"

```

- 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

What is happening here?

- The **dependencies** sections tells maven what jars to download.
- The **parent** sections configures which versions of those jars to download.

```

<artifactId>spring-boot-parent</
<version>1.1.0.RELEASE</version>

```

```

<artifactId>spring-boot-parent</
<version>1.4.2.RELEASE</version>

```

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>

```

```

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

```

```

> tomcat-embed-core-7.0.54.jar - C:\Us
> tomcat-embed-el-7.0.54.jar - C:\Users\
> tomcat-embed-logging-juli-7.0.54.jar

```

```

> tomcat-embed-core-8.5.6.jar - C:\Us
> tomcat-embed-el-8.5.6.jar - C:\User:
> tomcat-embed-websocket-8.5.6.jar -

```

```

> spring-core-4.0.5.RELEASE.jar - C:
> spring-web-4.0.5.RELEASE.jar - C:
> spring-aop-4.0.5.RELEASE.jar - C:\
> aopalliance-1.0.jar - C:\Users\Can
> spring-beans-4.0.5.RELEASE.jar - C:
> spring-context-4.0.5.RELEASE.jar -

```

```

> spring-web-4.3.4.RELEASE.jar - C:\Us
> spring-aop-4.3.4.RELEASE.jar - C:\Us
> spring-beans-4.3.4.RELEASE.jar - C:\I
> spring-context-4.3.4.RELEASE.jar - C:
> spring-webmvc-4.3.4.RELEASE.jar - C:
> spring-expression-4.3.4.RELEASE.jar -

```

Bill of materials: This preset list of possible 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.

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.** **TODO Where?**
- **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.

```
@RestController
public class TopicController {

    @Autowired
    private TopicService topicService;

    @RequestMapping("/topics")
    public List<Topic> getAllTopics() {
        return topicService.getAllTopics();
    }
}
```

```
▼ com.greydev.courseapi.topic
  > Topic.java
  > TopicController.java
  > TopicService.java
```

```
import org.springframework.stereotype.Service;

@Service
public class TopicService {

    public List<Topic> getAllTopics() {

        List<Topic> topics = new ArrayList<>();
        Topic t1 = new Topic("streams", "Java st
        Topic t2 = new Topic("final", "Java fina
        Topic t3 = new Topic("spring", "Java spr
        Topic t4 = new Topic("oobasics", "Java C

        topics.add(t1);
        topics.add(t2);
        topics.add(t3);
        topics.add(t4);

        return topics;
    }
}
```


In Spring 2.0 and later, the `@Repository` annotation is a marker for any class that fulfills the role or stereotype (also known as Data Access Object or DAO) of a repository. Among the uses of this marker is the automatic translation of exceptions.

Spring 2.5 introduces further stereotype annotations: `@Component`, `@Service`, and `@Controller`. `@Component` is a generic stereotype for any Spring-managed component. `@Repository`, `@Service`, and `@Controller` are specializations of `@Component` for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.

Therefore, you can annotate your component classes with `@Component`, but by annotating them with `@Repository`, `@Service`, or `@Controller` instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts.

Thus, if you are choosing between using `@Component` or `@Service` for your service layer, `@Service` is clearly the better choice. Similarly, as stated above, `@Repository` is already supported as a marker for automatic exception translation in your persistence layer.

Annotation	Meaning
<code>@Component</code>	generic stereotype for any Spring-managed component
<code>@Repository</code>	stereotype for persistence layer
<code>@Service</code>	stereotype for service layer
<code>@Controller</code>	stereotype for presentation layer (spring-mvc)

2. Spring Annotations

In most typical applications, we have distinct layers like data access, presentation, service, business, etc.

And, in each layer, we have various beans. Simply put, to detect them automatically, **Spring uses classpath scanning annotations**.

Then, it registers each bean in the *ApplicationContext*.

Here's a quick overview of a few of these annotations:

- `@Component` is a generic stereotype for any Spring-managed component
- `@Service` annotates classes at the service layer
- `@Repository` annotates classes at the persistence layer, which will act as a database repository

We already have an [extended article](#) about these annotations. So we'll keep the focus only on the differences between them.

3.2. @Repository

@Repository's job is to catch persistence specific exceptions and rethrow them as one of Spring's unified unchecked exception.

For this Spring provides *PersistenceExceptionTranslationPostProcessor*, that requires to add in our application context:

```
<bean class="org.springframework.dao.annotation.PersistenceExcepti
< >
```

This bean post processor adds an advisor to any bean that's annotated with `@Repository`.

3. What's Different?

The major difference between these stereotypes is they are used for different classification. When we annotate a class for auto-detection, then we should use the respective stereotype.

Now, let's go through them in more detail.

3.1. @Component

We can use @Component across the application to mark the beans as Spring's managed components. Spring only pick up and registers beans with `@Component` and doesn't look for `@Service` and `@Repository` in general.

They are registered in *ApplicationContext* because they themselves are annotated with `@Component`.

```
@Component      @Component
public @interface Service {    public @interface Repository {
}                      }
```

`@Service` and `@Repository` are special cases of `@Component`. They are technically the same but we use them for the different purposes.

3.3. @Service

We mark beans with @Service to indicate that it's holding the business logic. So there's no any other specialty except using it in the service layer.

```

public Topic updateTopic(String id, Topic newTopic) {
    topics.forEach(t -> {
        if (t.getId().equalsIgnoreCase(newTopic.getId())) {
            topics.remove(t);
        }
    });
    return newTopic;
}

```

```

public Topic updateTopic(String id, Topic newTopic) {
    for (Topic t : topics) {
        if (t.getId().equalsIgnoreCase(newTopic.getId())) {
            topics.add(newTopic);
        }
    }
    return newTopic;
}

```

```

java.util.ConcurrentModificationException: null
    at java.util.ArrayList.forEach(Unknown Source) ~[na:1.8.0_201]
    at com.greydev.courseapi.topic.TopicService.updateTopic(TopicService.java:34) ~[classes/:na]
    at com.greydev.courseapi.topic.TopicController.updateTopic(TopicController.java:35) ~[classes/:na]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_201]

```

The error is in this part:

Don't modify the list you are iterating over. You could solve this by using the `Iterator` explicitly:

```

for (String s : tempFile){
    String [] splitted = s.split(" ");
    if (splitted[0].equals(naam)){
        tempFile.remove(s);
        found = true;
    }
}

```

```

for (Iterator<String> it = tempFile.iterator(); it.hasNext();) {
    String s = it.next();
    String [] splitted = s.split(" ");
    if (splitted[0].equals(naam)){
        it.remove();
        found = true;
    }
}

```

The Java 5 enhanced for loop uses an Iterator underneath. So When you remove from tempFile the fail fast nature kicks in and throws the Concurrent exception. Use an iterator and call its remove method, which will remove from the underlying Collection.

```

for (Iterator<Topic> it = topics.iterator(); it.hasNext();) {
    Topic t = it.next();
    if (t.getId().equalsIgnoreCase(id)) {
        it.remove(); // deletes the element from topics!
                    // does NOT have an add() method!
    }
}

```

```

for (ListIterator<Topic> listIterator = topics.listIterator();
     listIterator.hasNext();) {
    Topic t = listIterator.next();
    if (t.getId().equalsIgnoreCase(id)) {
        listIterator.remove();
        listIterator.add(newTopic);
    }
}

```

```

for (int i = 0; i < topics.size(); i++) {
    Topic t = topics.get(i);
    if (t.getId().equalsIgnoreCase(id)) {
        topics.set(i, newTopic);
    }
}

```

```

public Topic deleteTopic(String id) {
    boolean removed = topics.removeIf(t -> t.getId().equalsIgnoreCase(id));
}

```



```
public interface Iterator<E>
```

An iterator over a collection. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:

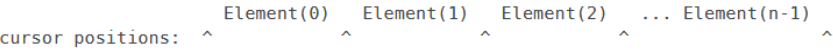
- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:
1.2

```
public interface ListIterator<E>  
extends Iterator<E>
```

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A ListIterator has no current element; its *cursor position* always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next(). An iterator for a list of length n has n+1 possible cursor positions, as illustrated by the carets (^) below:



Note that the remove() and set(Object) methods are *not* defined in terms of the cursor position; they are defined to operate on the last element returned by a call to next() or previous().

This interface is a member of the Java Collections Framework.

Since:
1.2

default void **forEachRemaining**(Consumer<? super E> action)
Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

boolean **hasNext**()
Returns true if the iteration has more elements.

E **next**()
Returns the next element in the iteration.

default void **remove**()
Removes from the underlying collection the last element returned by this iterator (optional operation).

Modifier and Type	Method and Description
void	add (E e) Inserts the specified element into the list (optional operation).
boolean	hasNext () Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious () Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next () Returns the next element in the list and advances the cursor position.
int	nextIndex () Returns the index of the element that would be returned by a subsequent call to next () .
E	previous () Returns the previous element in the list and moves the cursor position backwards.
int	previousIndex () Returns the index of the element that would be returned by a subsequent call to previous () .
void	remove () Removes from the list the last element that was returned by next () or previous () (optional operation).
void	set (E e) Replaces the last element returned by next () or previous () with the specified element (optional operation).

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.ListIterator;

import org.springframework.stereotype.Service;

@Service
public class TopicService {

    List<Topic> topics = new ArrayList<>(Arrays.asList(new Topic("streams", "Java streams", "Java info about streams"),
        new Topic("final", "Java final keyword", "Java info about final keyword"),
        new Topic("spring", "Java spring framework", "Java info about spring framework"),
        new Topic("oobasics", "Java OO basics", "Java info about OO basics")));

    public List<Topic> getAllTopics() {
        return topics;
    }

    public Topic getTopic(String id) {
        return topics.stream()
            .filter(t -> t.getId().equalsIgnoreCase(id))
            .findFirst()
            .get();
    }

    public Topic addTopic(Topic topic) {
        topics.add(topic);
        return topic;
    }

    public Topic updateTopic(String id, Topic newTopic) {
        newTopic.setId(id);

        if (getTopic(id) == null) {
            addTopic(newTopic);
            return newTopic;
        }

        for (ListIterator<Topic> listIterator = topics.listIterator(); listIterator.hasNext();) {
            Topic t = listIterator.next();
            if (t.getId().equalsIgnoreCase(id)) {
                listIterator.remove();
                listIterator.add(newTopic);
            }
        }
        return newTopic;
    }

    public Topic deleteTopic(String id) {
        boolean removed = topics.removeIf(t -> t.getId().equalsIgnoreCase(id));

        for (ListIterator<Topic> listIterator = topics.listIterator(); listIterator.hasNext();) {
            Topic topicToDelete = listIterator.next();
            if (topicToDelete.getId().equalsIgnoreCase(id)) {
                listIterator.remove();
                return topicToDelete;
            }
        }
        throw new RuntimeException("no topic found with id:" + id);
    }
}

```


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**

Spring Initializr

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

**Spring Initializr**
Bootstrap your application

Project

Language

Spring Boot

Project Metadata

Dependencies

Maven Project

Gradle Project

Java

Kotlin

Groovy

2.2.0 M4

2.2.0 (SNAPSHOT)

2.1.7 (SNAPSHOT)

2.1.6

1.5.21

Group
com.example

Artifact
demo

> Options

Q

≡

Options

Name
demo

Description
Demo project for Spring Boot

Package Name
com.example.demo

Packaging
JarWar

Developer Tools

Spring Boot DevTools
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Lombok
Java annotation library which helps to reduce boilerplate code.

Spring Configuration Processor
Generate metadata for developers to offer contextual help and "code completion" when working with custom configuration keys (ex.application.properties/.yaml files).

Template Engines

Security

SQL

NoSQL

Messaging

I/O

Spring Cloud

Spring Cloud Security

Amazon Web Services

...

Web

Spring Web Starter
Build web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container.

Spring Reactive Web
Build reactive web applications with Spring WebFlux and Netty.

Rest Repositories
Exposing Spring Data repositories over REST via Spring Data REST.

Spring Session
Provides an API and implementations for managing user session information.

Rest Repositories HAL Browser
Browsing Spring Data REST repositories in your browser.

Spring HATEOAS
Eases the creation of RESTful APIs that follow the HATEOAS principle when working with Spring / Spring MVC.

Spring Web Services
Facilitates contract-first SOAP development. Allows for the creation of flexible web services using one of the many ways to manipulate XML payloads.

Jersey
Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs.

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.

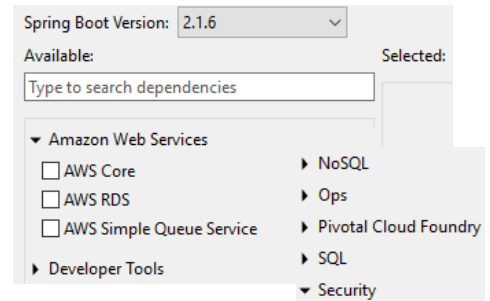
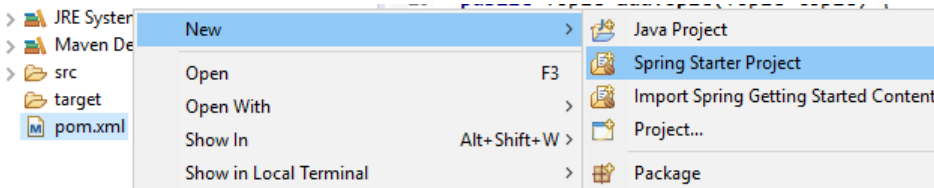
You do not need to use the CLI to work with Spring Boot, but it is definitely the quickest way to get a Spring application off the ground.

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

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

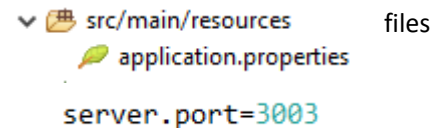
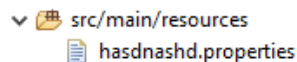


Configure Spring Boot

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 under **src/main/resources** and it **must** have the name **application.properties**



Appendix A. Common application properties

Various properties can be specified inside your **application.properties** file, inside your **application.yml** file, or as command line switches. This appendix provides a list of common Spring Boot properties and references to the underlying classes that consume them.

```
server.port=8080 # Server HTTP port.
```

Property contributions can come from additional jar files on your classpath, so you should not consider this an exhaustive list. Also, you can define your own properties.

This sample file is meant as a guide only. Do **not** copy and paste the entire content into your application. Rather, pick only the properties that you need.

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

```
import org.springframework.data.repository.CrudRepository;

public interface TopicRepository extends CrudRepository<Topic, String> {

}
```

org.springframework.data.repository	
Interface CrudRepository<T,ID>	
long	Returns the number of entities available.
void	delete(T entity) Deletes a given entity.
void	deleteAll() Deletes all entities managed by the repository.
void	deleteAll(Iterable<? extends T> entities) Deletes the given entities.
void	deleteById(ID id) Deletes the entity with the given id.
boolean	existsById(ID id) Returns whether an entity with the given id exists.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
<version>1.4.2.RELEASE</version>
</dependency>

<dependency>
<groupId>org.apache.derby</groupId>
<artifactId>derby</artifactId>
<scope>runtime</scope>
</dependency>

@Entity
public class Topic {

    @Id
    @Column(name = "topic_id")
    private String id;
    @Column(name = "topic_name")
    private String name;
    @Column(name = "topic_description")
    private String description;

    import javax.persistence.Column;
    import javax.persistence.Entity;
    import javax.persistence.Id;
```

Iterable<T>	findAll() Returns all instances of the type.
Iterable<T>	findAllById(Iterable<ID> ids) Returns all instances of the type with the given IDs.
Optional<T>	findById(ID id) Retrieves an entity by its id.
<S extends T> S	save(S entity) Saves a given entity.
<S extends T> Iterable<S>	saveAll(Iterable<S> entities) Saves all given entities.

```
@Service
public class TopicService {

    @Autowired
    private TopicRepository topicRepository;
```

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

```

@Service
public class TopicService {

    @Autowired
    private TopicRepository topicRepository;

    public List<Topic> getAllTopics() {
        List<Topic> resultList = new ArrayList<>();
        Iterable<Topic> iterable = topicRepository.findAll();
        iterable.forEach(resultList::add);
        return resultList;
    }

    public Topic getTopic(String id) {
        // .findById(id) came with spring-boot-parent 2.0+
        Optional<Topic> optional = topicRepository.findById(id);

        try {
            return optional.get(); // can throw NoSuchElementException
        }
        catch (NoSuchElementException e) {
            System.out.println("No such element with id: " + id);
            return null;
        }
    }
}

```

```

public Topic addTopic(Topic topic) {
    if (topicRepository.existsById(topic.getId())) {
        throw new RuntimeException("topic id already exists");
    }
    Topic savedTopic = topicRepository.save(topic);
    if (savedTopic == null) {
        throw new RuntimeException("topic couldn't be saved");
    }
    return savedTopic;
}

public Topic updateTopic(String id, Topic newTopic) {
    newTopic.setId(id);

    Topic savedTopic = topicRepository.save(newTopic);
    if (savedTopic == null) {
        throw new RuntimeException("topic couldn't be saved");
    }
    return savedTopic;
}

public void deleteTopic(String id) {
    Objects.requireNonNull(id, "id can't be null");

    try {
        // .deleteById(id) came with spring-boot-parent 2.0+
        topicRepository.deleteById(id);
    }
    catch (EmptyResultDataAccessException e) {
        System.out.println("failed delete by id: " + id);
    }
}

```


Connecting to an External Database – MySQL

MySQL is licensed with the GPL, so any program binary that you distribute using it must use the GPL too. Refer to the [GNU General Public Licence](#).

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
```

```
create database spring_db;
```

Create the `application.properties` file

Spring Boot gives you defaults on all things, the default in database is `H2`, so when you want to change this and use any other database you must define the connection attributes in the `application.properties` file.

Here, `spring.jpa.hibernate.ddl-auto` can be `none`, `update`, `create`, `create-drop`, refer to the Hibernate documentation for details.

- `none` This is the default for `MySQL`, no change to the database structure.
- `update` Hibernate changes the database according to the given Entity structures.
- `create` Creates the database every time, but don't drop it when close.
- `create-drop` Creates the database then drops it when the `SessionFactory` closes.

The default is `none`. It is good security practice that after your database is in production state, you make this `none` and revoke all privileges from the MySQL user connected to the Spring application, then give him only SELECT, UPDATE, INSERT, DELETE.

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/spring_db
spring.datasource.username=root
spring.datasource.password=root
```

The above example does not explicitly specify `GET` vs. `PUT`, `POST`, and so forth, because `@GetMapping` is a shortcut for `@RequestMapping(method=GET)`. `@RequestMapping` maps all HTTP operations by default. Use `@RequestMapping(method=GET)` or other shortcut annotations to narrow this mapping.

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 'Mitteleuropäische 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 specific 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:

```
mvn clean install
$ java -jar ./target/course-api-0.0.1-SNAPSHOT.jar
no main manifest attribute, in ./target/course-api-0.0.1-SNAPSHOT.jar
Picked up _JAVA_OPTIONS: -Xmx512M
```

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

```
<executions>
  <execution>
    <goals>
      <goal>repackage</goal>
    </goals>
  </execution>
</executions>
```

- Use to directly build and run you app:

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

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

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>1.2.5.RELEASE</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Make some security changes

Now when you are on production environment, you may be exposed to SQL injection attacks. A hacker may inject `DROP TABLE` or any other destructive SQL commands. So as a security practice, make those changes to your database before you expose the application to users.

```
mysql> revoke all on db_example.* from 'springuser'@'localhost';
```

This revokes ALL the privileges from the user associated with the Spring application. Now the Spring application **cannot do** anything in the database. We don't want that, so

```
sql> grant select, insert, delete, update on db_example.* to 'springuse
```

This gives your Spring application only the privileges necessary to make changes to **only** the data of the database and not the structure (schema).

Now make this change to your `src/main/resources/application.properties`

```
spring.jpa.hibernate.ddl-auto=none
```

This is instead of `create` which was on the first run for Hibernate to create the tables from your entities.

When you want to make changes on the database, regrant the permissions, change the `spring.jpa.hibernate.ddl-auto` to `update`, then re-run your applications, then repeat. Or, better, use a dedicated migration tool such as Flyway or Liquibase.

```
@Entity
public class Topic {

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "topic")
    private Set<Course> courses = new HashSet<>();
```

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

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "topic")
private Set<Course> courses = new HashSet<>();
```

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.

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

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.

ID	Description	Enabled by default
<code>auditevents</code>	Exposes audit events information for the current application.	Yes
<code>beans</code>	Displays a complete list of all the Spring beans in your application.	Yes
<code>caches</code>	Exposes available caches.	Yes
<code>conditions</code>	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.	Yes
<code>configprops</code>	Displays a collated list of all <code>@ConfigurationProperties</code> .	Yes
<code>env</code>	Exposes properties from Spring's <code>ConfigurableEnvironment</code> .	Yes
<code>flyway</code>	Shows any Flyway database migrations that have been applied.	Yes
<code>health</code>	Shows application health information.	Yes
<code>httptrace</code>	Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges).	Yes

<code>info</code>	Displays arbitrary application info.	Yes
<code>integrationgraph</code>	Shows the Spring Integration graph.	Yes
<code>loggers</code>	Shows and modifies the configuration of loggers in the application.	Yes
<code>liquibase</code>	Shows any Liquibase database migrations that have been applied.	Yes
<code>metrics</code>	Shows 'metrics' information for the current application.	Yes
<code>mappings</code>	Displays a collated list of all <code>@RequestMapping</code> paths.	Yes
<code>scheduledtasks</code>	Displays the scheduled tasks in your application.	Yes
<code>sessions</code>	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Not available when using Spring Session's support for reactive web applications.	Yes
<code>shutdown</code>	Lets the application be gracefully shutdown.	No
<code>threaddump</code>	Performs a thread dump.	Yes

If your application is a web application (Spring MVC, Spring WebFlux, or Jersey), you can use the following additional endpoints:

ID	Description	Enabled by default
<code>heapdump</code>	Returns an <code>hprof</code> heap dump file.	Yes
<code>jolokia</code>	Exposes JMX beans over HTTP (when Jolokia is on the classpath, not available for WebFlux).	Yes
<code>logfile</code>	Returns the contents of the logfile (if <code>logging.file</code> or <code>logging.path</code> properties have been set). Supports the use of the HTTP <code>Range</code> header to retrieve part of the log file's content.	Yes
<code>prometheus</code>	Exposes metrics in a format that can be scraped by a Prometheus server.	Yes

@Controller vs @RestController

- `@Controller` is used to mark classes as Spring MVC Controller.
- `@RestController` is a convenience annotation that does nothing more than adding the `@Controller` and `@ResponseBody` annotations (see: [Javadoc](#))

So the following two controller definitions should do the same

```
@Controller
@ResponseBody
public class MyController { }

@RestController
public class MyRestController { }
```

```
@Controller
public class restClassName{

    @RequestMapping(value={"/uri"})
    @ResponseBody
    public ObjectResponse functionRestName(){
        //...
        return instance
    }
}
```

```
@RestController
public class restClassName{

    @RequestMapping(value={"/uri"})
    public ObjectResponse functionRestName(){
        //...
        return instance
    }
}
```

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.

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.

29.1.6 Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

29.1.7 Custom Favicon

Spring Boot looks for a `favicon.ico` in the configured static content locations and the root of the classpath (in that order). If such a file is present, it is automatically used as the favicon of the application.

Template Engines

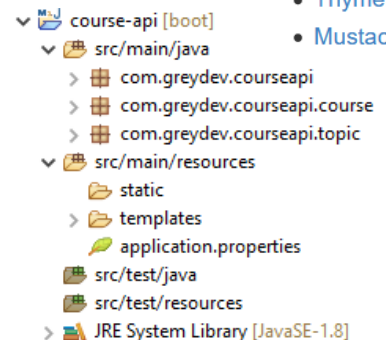
As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

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

- [FreeMarker](#)
- [Groovy](#)
- [Thymeleaf](#)
- [Mustache](#)

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.

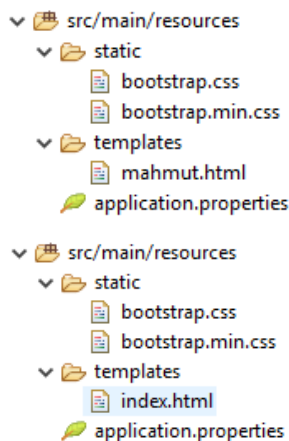
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8" />
<title>Hotel Booking App</title>
<link rel="stylesheet" href=" ../static/bootstrap.css" />
</head>
<body>
  Hello from index!
</body>
</html>
```

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:



```
@Controller
public class ViewController {

    @RequestMapping("/")
    public String getMain() {
        return "mahmut";
    }
}
```

```
@Controller
public class ViewController {

    @RequestMapping("/")
    public String getMain() {
        return "index";
    }
}
```

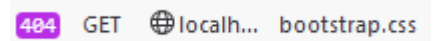
localhost:8080
Hello from index!

localhost:8080
Hello from index!

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.

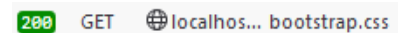


- However if we open this file with a browser (offline mode), it will work as expected.

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

th:href="@{/bootstrap.css}"

- **@** 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.

```
<p class="navbar-text navbar-right" th:text="'Signed in as ' + ${username}"
style="margin-right: 10px;">Signed in as Anonymus</p>
```

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.

```
@RequestMapping("/")
public String getMain(Model model) {

    model.addAttribute("datetime", new Date());
    model.addAttribute("username", "Jonathan");

    return "index";
}
```

localhost:8080 Signed in as Jonathan

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

The way of return ModelAndView

```
@RequestMapping(value = "/list", method = RequestMethod.GET)
public ModelAndView list(
    @UserAuth UserAuth user,
    ModelAndView mav) {

    if (!user.isAuthenticated()) {
        mav.setViewName("redirect:http://www.test.com/login.jsp");
        return mav;
    }

    mav.setViewName("list");
    mav.addObject("articles", listService.getLists());

    return mav;
}
```

The way of return String

```
@RequestMapping(value = "/list", method = RequestMethod.GET)
public String list(
    @UserAuth UserAuth user,
    Model model) {

    if (!user.isAuthenticated()) {
        return "redirect:http://www.test.com/login.jsp";
    }

    model.addAttribute("articles", listService.getLists());

    return "list";
}
```

There is no better way. Both are perfectly valid. Which one you choose to use depends which one suits your application better - Spring allows you to do it either way.

I would like to add me 2 cents also. Second approach is more towards convention oriented i.e developer does explicitly mention what is his view but its implicit that return string is view name. So less coding, readable and standard. Much better than older way with ModelAndView

Historically, the two approaches come from different versions of Spring. The `ModelAndView` approach was the primary way of returning both model and view information from a controller in pre-Spring 2.0. Now you can combine the `Model` parameter and the `String` return value, but the old approach is still valid.

What are we trying to achieve?

When you use Spring MVC you can add certain arguments to your controller methods and the framework injects those arguments for you:

In the snippet above `request`, `model` and `locale` are effortlessly provided to us by the framework. We don't need to expend time and lines of code retrieving them and we can just focus on our business logic.

The good news is that Spring MVC provides us with the tools we need to use *this magic* to support our own custom arguments.

```
@RequestMapping("/")
public String landing(HttpServletRequest request, Model model, Locale locale) {
    model.addAttribute("attr", someService.getAttribute(request, locale);
    return "some-template";
}
```

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");`

Expression Basic Objects

When evaluating OGNL expressions on the context variables, some objects are made available to expressions for higher flexibility. These objects will be referenced (per OGNL standard) starting with the `#` symbol:

- `#ctx` : the context object.
- `#vars` : the context variables.
- `#locale` : the context locale.
- `#HttpServletRequest` : (only in Web Contexts) the `HttpServletRequest` object.
- `#HttpSession` : (only in Web Contexts) the `HttpSession` object.

So we can do this:

Established locale country:

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

Expression Utility Objects

Besides these basic objects, Thymeleaf will offer us a set of utility objects that will help us perform common tasks in our expressions.

- `#dates` : utility methods for `java.util.Date` objects: formatting, component extraction, etc.
- `#calendars` : analogous to `#dates`, but for `java.util.Calendar` objects.
- `#numbers` : utility methods for formatting numeric objects.
- `#strings` : utility methods for `String` objects: contains, startsWith, prepending/appending, etc.
- `#objects` : utility methods for objects in general.
- `#booleans` : utility methods for boolean evaluation.
- `#arrays` : utility methods for arrays.
- `#lists` : utility methods for lists.
- `#sets` : utility methods for sets.
- `#maps` : utility methods for maps.
- `#aggregates` : utility methods for creating aggregates on arrays or collections.
- `#messages` : utility methods for obtaining externalized messages inside variables expressions, in the same way as they would be obtained using `#{...}` syntax.
- `#ids` : utility methods for dealing with id attributes that might be repeated (for example, as a result of an iteration).

```
<p th:text="${datetime}" class="text-muted">Page was rendered today.</p>
```

Sat Jul 13 22:40:10 CEST 2019

```
<p th:text="${#dates.format(datetime, 'dd MMM yyyy HH:mm')}" class="text-muted">Page
```

13 Jul 2019 22:41

You can use the utility methods to format the date.

Conditional logic

```
model.addAttribute("mode", "production");
```

```
<link th:if="${mode=='development'}" rel="stylesheet" href="../static/bootstrap.css" th:href="@{/bootstrap.css}" />
<link th:if="${mode=='production'}" rel="stylesheet" href="../static/bootstrap.css" th:href="@{/bootstrap.min.css}" />
```

Configuration

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

```
//can be injected either with
//property injection or constructor injection
@Value("${app-mode}")
private String appMode;

@GetMapping("/")
public String getMain(Model model) {

    model.addAttribute("datetime", new Date());
    model.addAttribute("username", "Jonathan");
    model.addAttribute("mode", appMode);
}
```

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

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/
spring.datasource.username=root
spring.datasource.password=root
app-mode=development

@Controller
public class ViewController {

    //can be injected either with
    //property injection or constructor injection
    @Value("${app-mode}")
    private String appMode;

    @Autowired
    public ViewController(Environment environment) {
        this.appMode = environment.getProperty("app-mode");
    }
}
```

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.

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



Developer tools are automatically disabled when running a fully packaged application. If your application is launched from `java -jar` or if it is started from a special classloader, then it is considered a "production application". If that does not apply to you (i.e. if you run your application from a container), consider excluding devtools or set the `-Dspring.devtools.restart.enabled=false` system property.



Flagging the dependency as optional in Maven or using a custom 'developmentOnly' configuration in Gradle (as shown above) is a best practice that prevents devtools from being transitively applied to other modules that use your project.

20.2 Automatic Restart

Applications that use `spring-boot-devtools` automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE, as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder is monitored for changes. Note that certain resources, such as static assets and view templates, [do not need to restart the application](#).



Automatic restart works very well when used with LiveReload. [See the LiveReload section](#) for details. If you use JRebel, automatic restarts are disabled in favor of dynamic class reloading. Other devtools features (such as LiveReload and property overrides) can still be used.

Triggering a restart

As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. The way in which you cause the classpath to be updated depends on the IDE that you are using. In Eclipse, saving a modified file causes the classpath to be updated and triggers a restart. In IntelliJ IDEA, building the project (`Build -> Build Project`) has the same effect.

20.3 LiveReload

The `spring-boot-devtools` module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. LiveReload browser extensions are freely available for Chrome, Firefox and Safari from livereload.com.

If you do not want to start the LiveReload server when your application runs, you can set the `spring.devtools.livereload.enabled` property to `false`



You can only run one LiveReload server at a time. Before starting your application, ensure that no other LiveReload servers are running. If you start multiple applications from your IDE, only the first has LiveReload support.