Spring Boot

Accessories

Version: 1.0

Date: 03.2021

www.ivoronline.com

ABOUT THIS BOOK

CONTENT

ABOUT THIS BOOK

This is second Book in the series

Spring Boot - Quick Start

Spring Boot - Accessories

Spring Boot - Security

Content

Intention of this Book is to quickly get you started with non-core Spring Boot functionalities like: Validation, Testing, Logging, Filters, Interceptors, etc.

Standalone Tutorials

The core of this book are standalone tutorials that explain different functionalities of Spring Boot.
Each tutorial contains minimum amount of code needed to explain specific functionality. Tutorials have minimum amount of encompassing text that explains related theory and different parts of the code. This approach allows students to grasp presented concepts in a very fast and efficient manner. Full code, which can also be downloaded from GitHub, prevents any time being wasted trying to make the code work. Simple examples allow for full understanding of the functionality without any unnecessary distractions.

Theoretical Background

Where needed tutorials are preceded by chapters focusing on theoretical background. This way reader can fully understand functionalities explained in the subsequent chapters. But such chapters are in minority and of secondary importance because the main focus is on practical applications.

Demo Applications

Book contains demo Applications that show how to combine functionalities covered in previous tutorials. More complex Application "Authors & Books" is broken into multiple steps showing how to add additional functionalities at each step.

WHY TUTORIALS?

"Things are only as complicated as they are badly explained"

Proper documentation is essential to avoid struggle and frustration when working with simple things that only seem complicated by not being properly documented and explained.

WHAT KIND OF TUTORIALS?

"Working example is worth thousand words"

Just like the picture is worth thousand words the same goes for the working example. Documentation in the form of working examples is proved to be the fastest and the most effective way of transferring knowledge. Sometimes an example is all you need to get the things done. And if there are some accompanying comments that explain what is going on even better. This approach is used in this book. This results in fast learning and the ability to apply tutorials when you need them in the spirit of Just In Time Support.

I wish you rapid learning!

1 Different Functionalities

Info

- Following tutorials show how to use different Spring Boot functionalities.

1.1 Validation

Info

- Following tutorials show how to validate User Input by validating either
  - individual HTTP Request Parameters (@RequestParam @NotBlank String name)
  - DTO during Deserialization from
    - Request Parameters (@NotBlank public String name;)
    - JSON Body (@NotBlank public String name;)
- In both cases the same Annotations can be assigned either to
  - Endpoint's Input Parameter (@RequestParam @NotBlank String name)
  - DTO Property (@NotBlank public String name;)
  - additional Endpoint is used to catch validation Exceptions (and return errors to the User)
- Controller's @ExceptionHandler Methods only catch Exceptions thrown inside the same Controller.
  Use @ControllerAdvice class as a central place to catch Exceptions from all Controllers.

Annotations

ANNOTATION	PARAMETERS
@NotBlank	(message = "String Name is mandatory")
@NotNull	(message = "Integer Age is mandatory")
@Min(18)	(value = 18, message = "Minimum value for Age is 18")
@Max(100)	(value = 100, message = "Maximum value for Age is 100")
@Size(min=5, max=30)	(min=5, max=30, message = "String must be between 5 and 30 characters")

Exceptions

VALIDATION TYPE	EXCEPTION	REFERENCE
(@RequestParam @NotBlank String name)	MissingServletRequestParameterException	Request Parameters
(@Valid PersonDTO personDTO)	BindException	DTO - Parameters
(@Valid @RequestBody PersonDTO personDTO)	MethodArgumentNotValidException	DTO - JSON

Validate & Throw Exceptions

Validate Request Parameters

MyController.java

```
@Validated
@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello(@RequestParam @NotBlank String  name ) {
    return "Hello " + name;
  }
```

Validate DTO (During Deserialization from Request Parameters)

MyController.java

```
@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello(@Valid PersonDTO personDTO) {
    return "Hello " + personDTO.name;
  }
```

PersonDTO.java

```
public class PersonDTO {

  //PROPERTIES
  @NotBlank public String  name;

  //SETTERS (Only needed for Deserialization from Request Parameters. JSON uses Reflection instead.)
  public void setName(String  name) { this.name = name; }

}
```

Validate DTO (During Deserialization from JSON)

MyController.java

```
@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/AddPerson")
  public String addPerson(@Valid @RequestBody PersonDTO personDTO) {
    return "Person added";
  }
```

PersonDTO.java

```
public class PersonDTO {
  //PROPERTIES
  @NotBlank public String  name;
}
```

Catch Exceptions

- Controller's `@ExceptionHandler` Method only catches Exceptions thrown inside the same Controller.
- Use `@ControllerAdvice` class as a central place to catch Exceptions from all Controllers.

Exception Methods *(differ base on the type of Exception)*

VALIDATION TYPE	EXCEPTION	REFERENCE
(@RequestParam @NotBlank String name)	MissingServletRequestParameterException	Request Parameters
(@Valid PersonDTO personDTO)	BindException	DTO - Parameters
(@Valid @RequestBody PersonDTO personDTO)	MethodArgumentNotValidException	DTO - JSON

MyController.java *(example for Request Parameters)*

```
@Controller
public class MyController {

  @ResponseBody
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  @ExceptionHandler(MissingServletRequestParameterException.class)
  public String handleExceptions(MissingServletRequestParameterException exception) {

    //GET EXCEPTION DETAILS
    String parameterType = exception.getParameterType(); //String
    String parameterName = exception.getParameterName(); //name
    String message       = exception.getMessage();       //Required String parameter 'name' is not present

    //RETURN MESSAGE
    return message;

  }
```

MyController.java

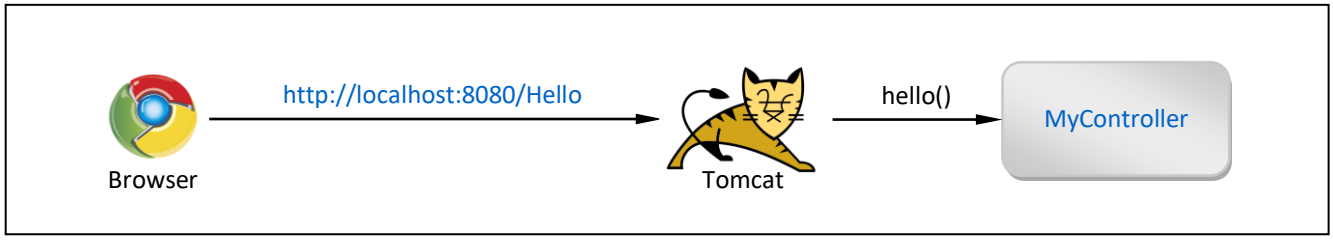
```
@ControllerAdvice
public class GlobalExceptionHandler {

  @ResponseBody
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  @ExceptionHandler(MissingServletRequestParameterException.class)
  public String handleIdentifiersNotMatchingException(MissingServletRequestParameterException exception) {

    //GET EXCEPTION DETAILS
    String parameterType = exception.getParameterType(); //String
    String parameterName = exception.getParameterName(); //name
    String message       = exception.getMessage();       //Required String parameter 'name' is not present

    //RETURN MESSAGE
    return message;

  }

}
```

1.1.1 Validate - Request Parameters

Info **[G] [R]**

- This tutorial shows how to Validate Request Parameters when using `@RequestParam`.
  If Parameter is missing or contains invalid value, `MissingServletRequestParameterException` is thrown.
  We will create Endpoint that catches Exception and returns error back to the User.
- If Endpoint uses multiple `@RequestParam`, Exception is thrown only for the **first error**.
  If you want all of the errors to be reported then you need to use DTO approach
  - DTO - From Request Parameters
  - DTO - From JSON

Application Schema *[Results]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_dto_json_object_constructor (add Spring Boot Starters from the table)
- Edit File: pom.xml (add validation dependency)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside package controllers)

pom.xml

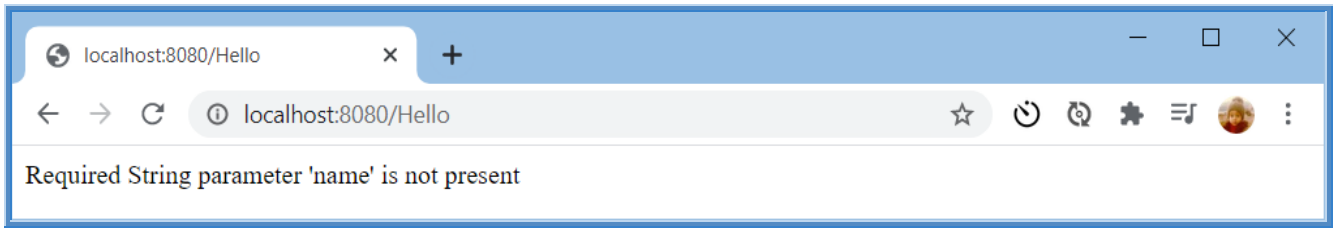
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

MyController.java

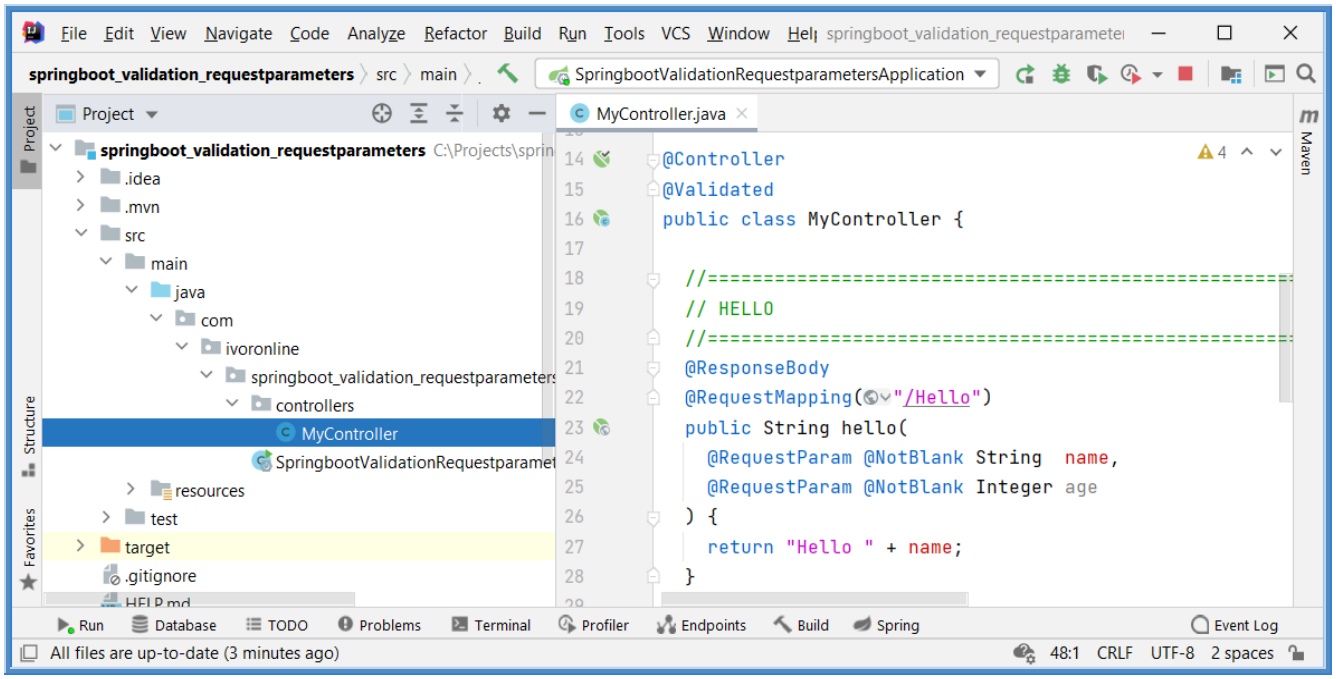
```
package com.ivoronline.springboot_validation_requestparameters.controllers;

import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.MissingServletRequestParameterException;
import org.springframework.web.bind.annotation.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

@Controller
@Validated
public class MyController {

  //=====================================================================
  // HELLO
  //=====================================================================
  @ResponseBody
  @RequestMapping("/Hello")
  public String hello(
    @RequestParam @NotBlank String  name,
    @RequestParam @NotNull  Integer age
  ) {
    return "Hello " + name;
  }

  //=====================================================================
  // HANDLE EXCEPTIONS (it only catches first exception)
  //=====================================================================
  @ResponseBody
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  @ExceptionHandler(MissingServletRequestParameterException.class)
  public String handleExceptions(MissingServletRequestParameterException exception) {

    //GET EXCEPTION DETAILS
    String parameterType = exception.getParameterType(); //String
    String parameterName = exception.getParameterName(); //name
    String message       = exception.getMessage();       //Required String parameter 'name' is not present

    //RETURN MESSAGE
    return message;
  }

}
```

Results

http://localhost:8080/Hello *(reports only first error)*



Application Structure



pom.xml

```
<dependencies>

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

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

</dependencies>
```

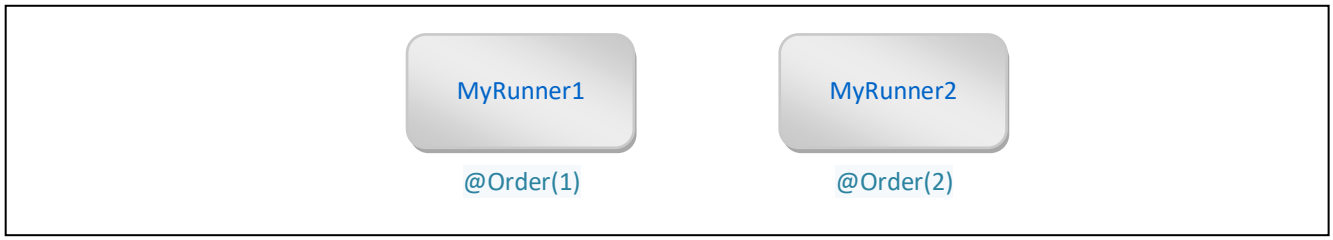
1.2 Tasks

1.2.1 Startup

Info [G]

- Following tutorials show how to create Class that implements CommandLineRunner to run Task at App startup.
- You simply need to @Override its run() Method with your custom code.
- You can use @Order(1) to specify order in which such Classes (and their run() Methods) should be executed.

Application Schema *[Results]*



Procedure

- Create Project: springboot_commandlinerunner (Spring Boot Starters are not used)
- Create Package: runners (inside main package)
  - Create Class: MyRunner1.java (inside runners package)
  - Create Class: MyRunner2.java (inside runners package)

MyRunner1.java

```
package com.ivoronline.springboot_commandlinerunner.runners;

import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Order(1)
@Component
public class MyRunner1 implements CommandLineRunner {

  @Override
  public void run(String... args) throws Exception {
    System.out.println("Hello from MyRunner1");
  }

}
```

MyRunner2.java

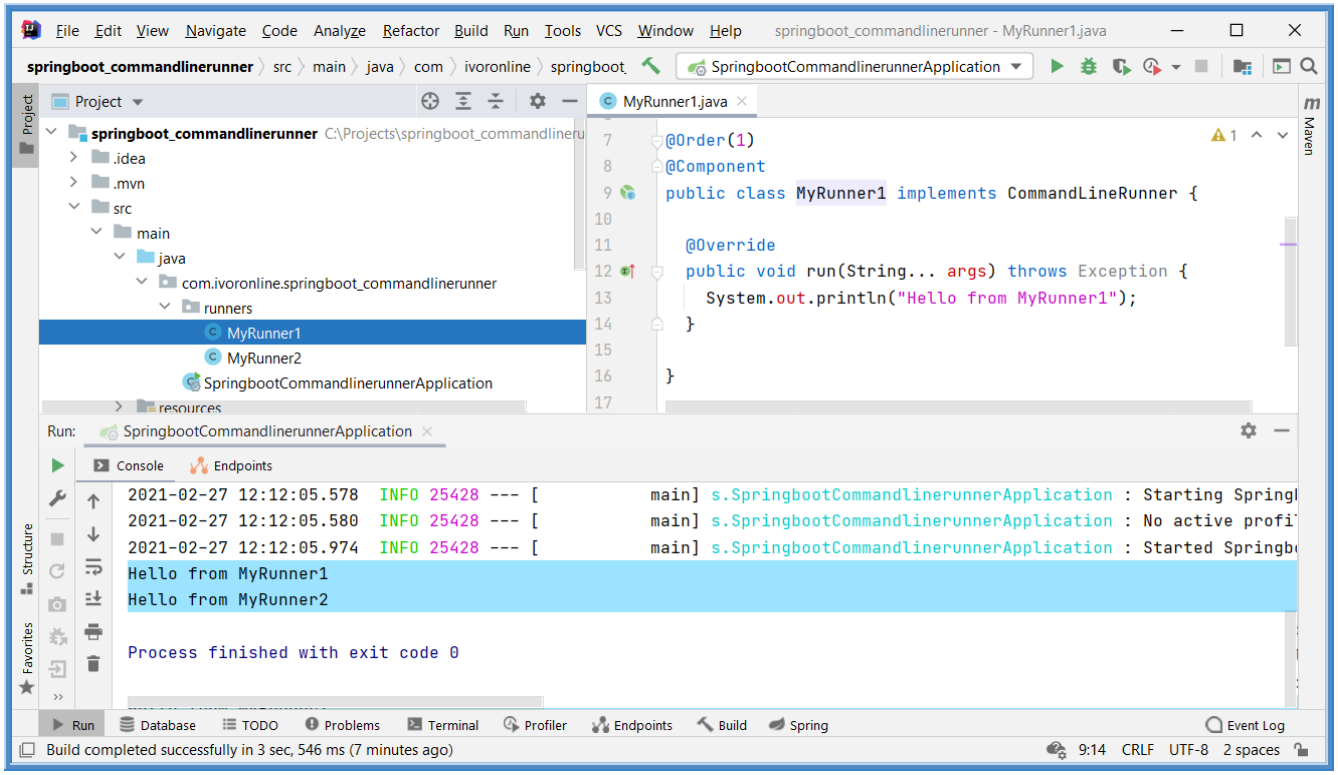
```
package com.ivoronline.springboot_commandlinerunner.runners;

import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Order(2)
@Component
public class MyRunner2 implements CommandLineRunner {

  @Override
  public void run(String... args) throws Exception {
    System.out.println("Hello from MyRunner2");
  }

}
```

Results

Console

```
Hello from MyRunner1
Hello from MyRunner2
```

Application Structure



1.2.2 Scheduled

Info [G] [R]

- This tutorial shows how to use **@Scheduled** Annotation to **periodically run methods** (in the future).
- To schedule a Method you just need to use @Scheduled(fixedDelay=5000) Annotation.
  To enable @Scheduled Annotation you need to add @EnableScheduling to some @Configuration Class.
- Method's **Class** must be declared as Spring Component for Spring to detect it.
  In this example this is achieved also by @Configuration (so @Configuration has dual role in this example).

@Scheduled Parameters *(milliseconds)*

```
@Scheduled(fixedRate  = 5000)                    //Measured from the START of previous task
@Scheduled(fixedDelay = 5000)                    //Measured from the END   of previous task
@Scheduled(fixedDelay = 5000, initialDelay = 1000)   //Delay before first execution
@Scheduled(cron="*/5 * * * * MON-FRI")           //Execute on weekdays
```

Application Schema *[Results]*

Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	To keep the Application running

Procedure

- Create Project: springboot_taskscheduler.tasks (add Spring Boot Starters from the table)
- Create Package: tasks (inside main package)
  - Create Class: MyTasks.java (inside runners package)

MyTasks.java

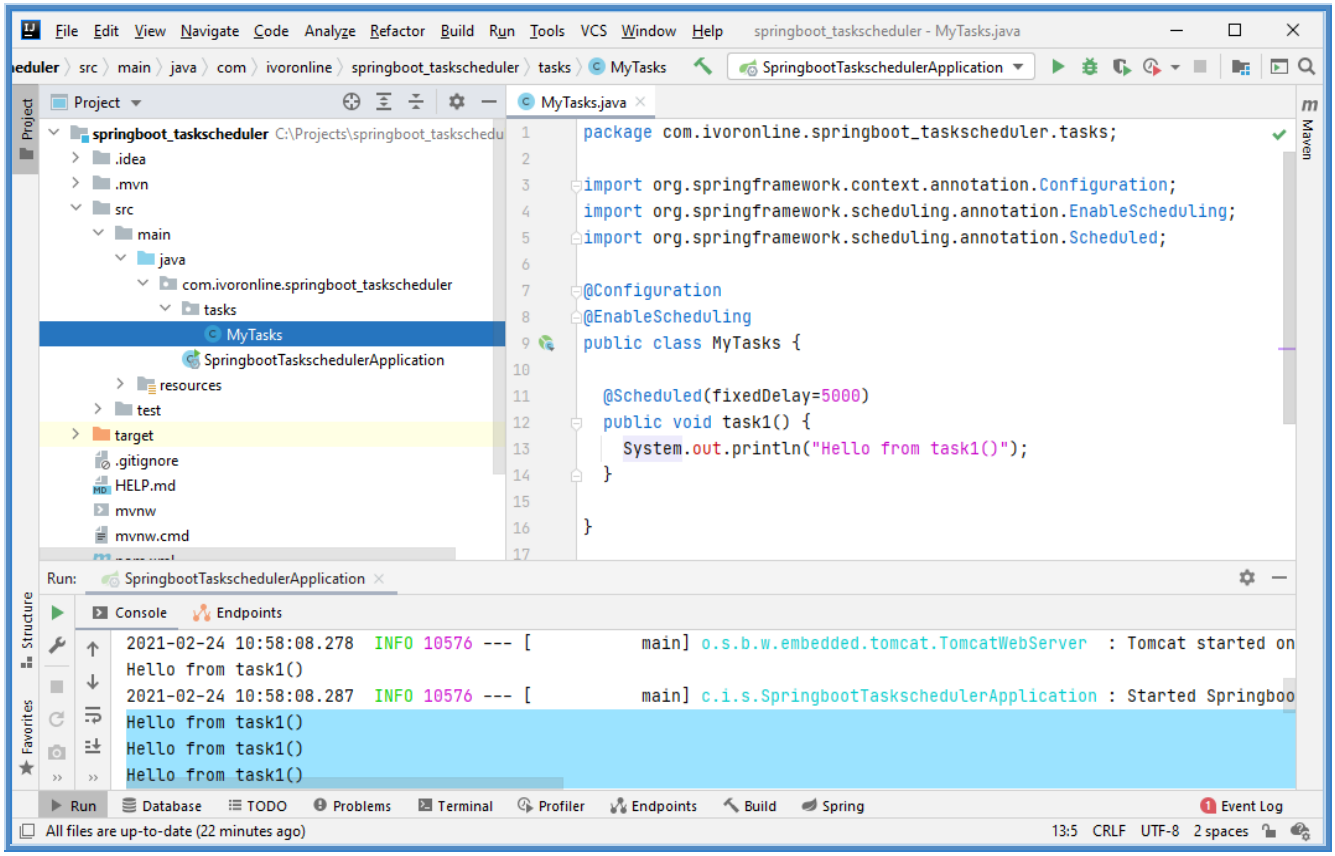
```
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;

@Configuration
@EnableScheduling
public class MyTasks {

  @Scheduled(fixedDelay=5000)
  public void task1() {
    System.out.println("Hello from task1()");
  }

}
```

Results

Console

```
Hello from task1()
Hello from task1()
```

Application Structure



pom.xml

```
<dependencies>

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

</dependencies>
```

1.3 Custom Banner

Info [G] [R]

- This tutorial shows how to create Custom Banner that is shown in the Console when Application starts.
  You can use different Web Sites to generate such banner from entered text (tutorial uses patorjk.com).
- By default file with banner should be located in src/main/resources/banner.txt.
  If default location and name are not used, custom parameter can be specified inside application.properties.

application.properties *(when using custom path and name)*

```
# CUSTOM PATH AND NAME
spring.banner.image.location = classpath:/src/main/resources/mybanner.txt
```

Design Banner

- http://patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20
- Type Custom Text: IVORONLINE.COM
- Test All
- Select & Copy (below to font you want)

Design Banner

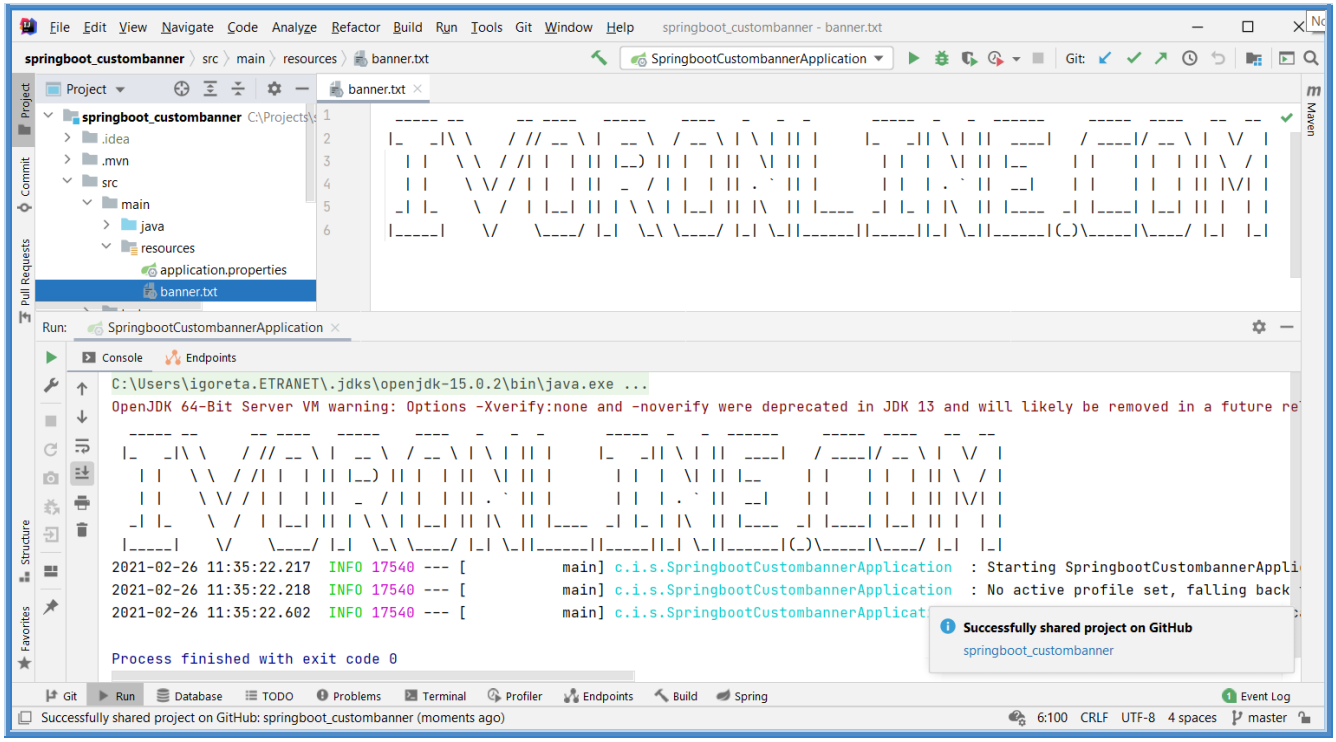
Add Banner

- Create Project: springboot_cuatombanner (there are no Spring Boot Starters)
- Create file: src/main/resources/**banner.txt**
- (Paste below Banner)
- Run

banner.txt

```
 _____ __      __ ____   _____    ____   _   _  _       _____  _   _  _____     _____  ____   __  __
|_   _|\ \    / // __ \ |  __ \  / __ \ | \ | || |     |_   _|| \ | ||  ____|   / ____|/ __ \ |  \/  |
  | |   \ \  / /| |  | || |__) || |  | ||  \| || |       | |  |  \| || |__     | |    | |  | || \  / |
  | |    \ \/ / | |  | ||  _  / | |  | || . ` || |       | |  | . ` ||  __|    | |    | |  | || |\/| |
 _| |_    \  /  | |__| || | \ \ | |__| || |\  || |____  _| |_ | |\  || |____  _| |____| |__| || |  | |
|_____|    \/    \____/ |_|  \_\ \____/ |_| \_||______||_____||_| \_||______|(_)\_____|\____/ |_|  |_|
```

Results



2 Filters & Interceptors

Info

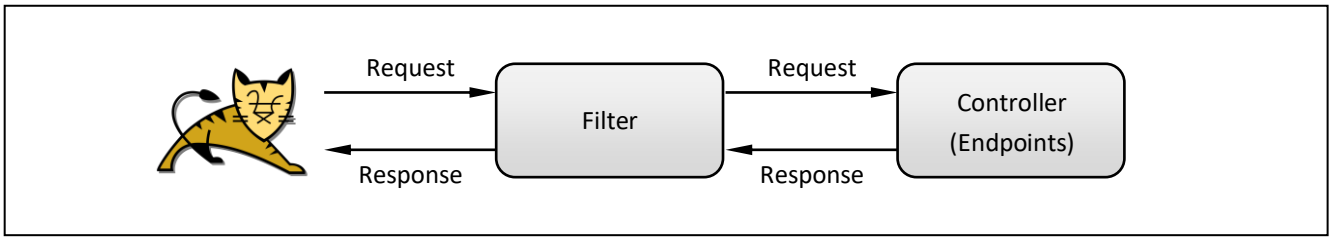
- Following tutorials show how to use Filters & Interceptors.

2.1 Filters

Info [R]

- **Filter** is **Java Class** that is used to **intercept HTTP Requests** and **Responses**.
- Filters perform additional operations before sending
  - HTTP Request to the Controller (it can modify HTTP Request before it gets to Controller)
  - HTTP Response from the Controller (it can modify HTTP Response before it is returned to the User)
- Every Filter is **called twice** (because HTTP Requests and Responses pass through the same Filters)
  - first during HTTP Request
  - then during HTTP Response
- Your custom **Filter Class** needs
  - to **implement** Filter Interface (so that Spring would know how to use it)
  - to **@Override** doFilter() Method (contains actual useful code performed by the Filter)
  - **@Component** Annotation (for Spring to detect it, create Filter Object, ads it to Filter Chain)
- Filters are used for (same as Interceptors)
  - **Security** (create Authentication Object from JWT Authorities)
  - **Logging** (log HTTP Requests/Responses: User, Endpoint, HTTP Response)
  - **Error Handling** (give feedback to User if HTTP Request has invalid Parameters/Format)

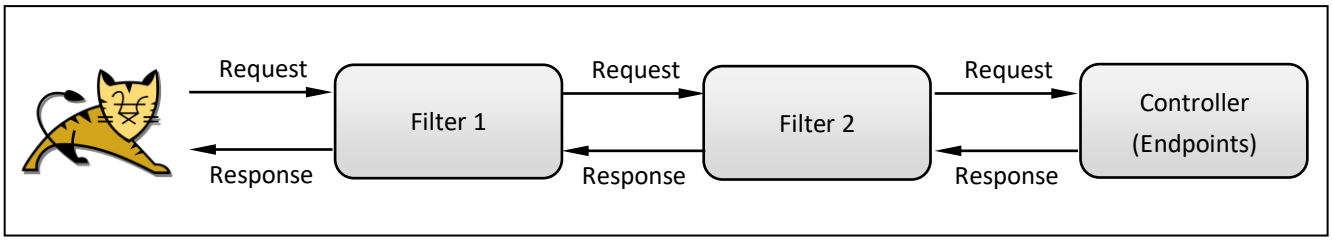
Filter is called twice



Filter Chain

- **Filter Chain** is ordered set of Filters that are
  - called in **sequence** before sending HTTP Request to the Controller
  - called in **reverse order** before sending HTTP Response from the Controller
- Filter Chain allows you to organize complex Filter Logic into multiple Filter Classes (for easier maintenance).

Filters are called in reverse order during HTTP Response



chain.doFilter(request, response);

- Call to chain.doFilter(request, response) Method is used to separate code that is execute during Request or Response.
- To return before reaching Controller don't call chain.doFilter(request, response) Method.

Request and Response Code

- Call to chain.doFilter(request, response) represents border between code that is execute during Request or Response
  - Code before chain.doFilter(request, response) is executed during HTTP Request
  - Code after chain.doFilter(request, response) is executed during HTTP Response
- Method chain.doFilter() calls
  - dofilter() Method of the next filter in chain
  - Endpoint if there are no more filters in chain
- When end of doFilter() Method is reached code returns to the next line after the call to chain.doFilter() in previous Filter
  - causing subsequent code to be executed during Http Response (code after call to chain.doFilter())
  - causing filters to be executed in reverse order during Http Response (but only code after call to chain.doFilter())

MyFilter.java

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
  System.out.println("MyFilter: Code for HTTP Request");     //CODE FOR HTTP REQUEST
  chain.doFilter(request, response);                         //DIVIDES CODE FOR HTTP REQUEST AND RESPONSE
  System.out.println("MyFilter: Code for HTTP Response");    //CODE FOR HTTP RESPONSE
}
```

Return before reaching Controller

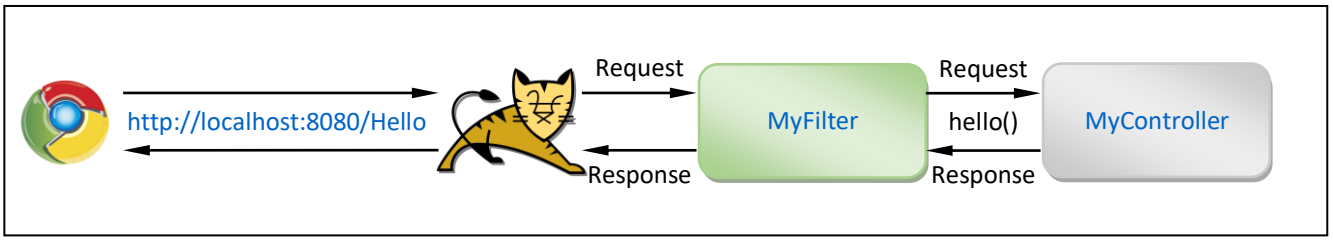
- If you don't call chain.doFilter(request, response) then Filter returns to previous Filter after its call to chain.doFilter(). This way Filter can stop propagation of HTTP Request to subsequent Filters or Controller and initiate HTTP Response.
- For instance Filter can analyze HTTP Request Parameters and if they are not valid
  - Filter can prevent Request from reaching Controller (or subsequent Filters down the Filter chain)
  - return HTTP Response instructing User which Parameters were wrong

2.1.1 Create Filter - Using Implements Filter

Info [G]

- This tutorial shows how to create Filter Class by using implements Filter.
- Code before chain.doFilter(request, response) is executed during HTTP Request.
  Code after chain.doFilter(request, response) is executed during HTTP Response.

Application Schema *[Results]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_filter_create (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Package: filters (inside main package)
  - Create Class: MyFilter.java (inside controllers package)

MyController.java

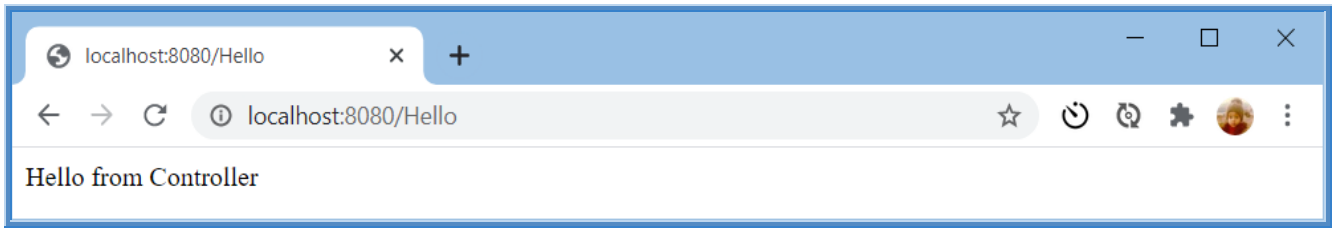
```
package com.ivoronline.springboot_filter_create.controllers;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    System.out.println("Controller: Code from Controller");
    return "Hello from Controller";
  }
}
```

MyFilter.java

```
package com.ivoronline.springboot_filter_create.filters;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import org.springframework.stereotype.Component;

@Component
public class MyFilter implements Filter {

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {

    //THIS CODE IS EXECUTED DURING HTTP REQUEST
    System.out.println("MyFilter  : Code for HTTP Request");

    //CALLS
    //  - doFilter() METHOD OF THE NEXT   FILTER  IN CHAIN
    //  - Endpoint   IF THERE ARE NO MORE FILTERS IN CHAIN
    //RETURNS FROM THIS METHOD DURING HTTP RESPONSE
    //  - CAUSING SUBSEQUENT CODE TO BE EXECUTED DURING HTTP RESPONSE
    //  - CAUSING FILTERS TO BE EXECUTED IN REVERSE ORDER DURING HTTP RESPONSE (BUT ONLY BELOW CODE)
    chain.doFilter(request, response);  //DIVIDES HTTP REQUEST AND RESPONSE CODE

    //THIS CODE IS EXECUTED DURING HTTP RESPONSE
    System.out.println("MyFilter  : Code for HTTP Response");

  }

}
```

Results

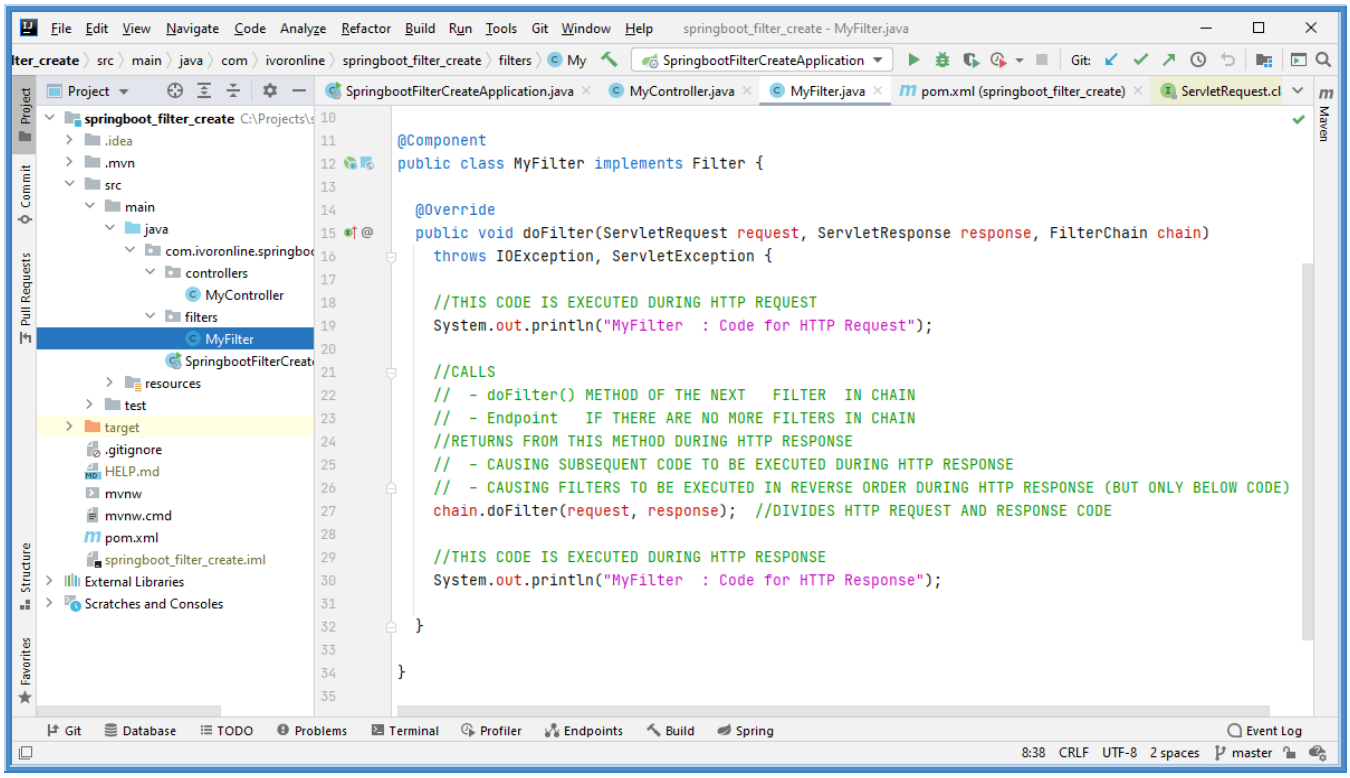
http://localhost:8080/Hello



Console

```
MyFilter  : Code for HTTP Request
Controller: Code from Controller
MyFilter  : Code for HTTP Response
```

Application Structure



pom.xml

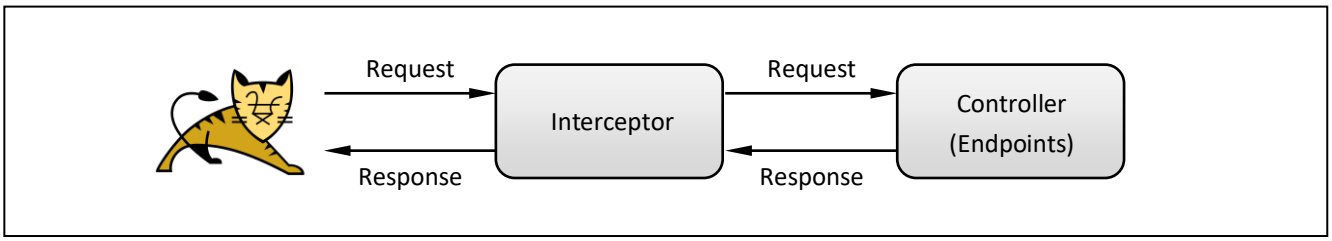
```
<dependencies>

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

</dependencies>
```

2.2 Interceptors

Info [R]

- **Interceptor** is **Java Class** that is used to **intercept HTTP Requests** and **Responses**.
- Interceptors perform additional operations before sending
  - HTTP Request to the Controller (they can modify HTTP Request before it gets to Controller)
  - HTTP Response from the Controller (it can modify HTTP Response before it is returned to the User)
- Every Interceptor is **called twice** (because HTTP Requests & Responses pass through same Interceptors)
  - first during HTTP Request
  - then during HTTP Response
- Your custom **Interceptor Class** needs
  - to **implement** HandlerInterceptor Interface (so that Spring would know how to use it)
  - to **@Override** preHandle() Method (contains actual useful code performed by the Filter)
  - to **@Override** postHandle() Method (contains actual useful code performed by the Filter)
  - to **@Override** afterCompletion() Method (contains actual useful code performed by the Filter)
  - **@Component** Annotation (for Spring to detect it and create Interceptor Object)
- Interceptors are used for (same as Filters)
  - **Security** (create Authentication Object from JWT Authorities)
  - **Logging** (log HTTP Requests/Responses: User, Endpoint, HTTP Response)
  - **Error Handling** (give feedback to User if HTTP Request has invalid Parameters/Format)

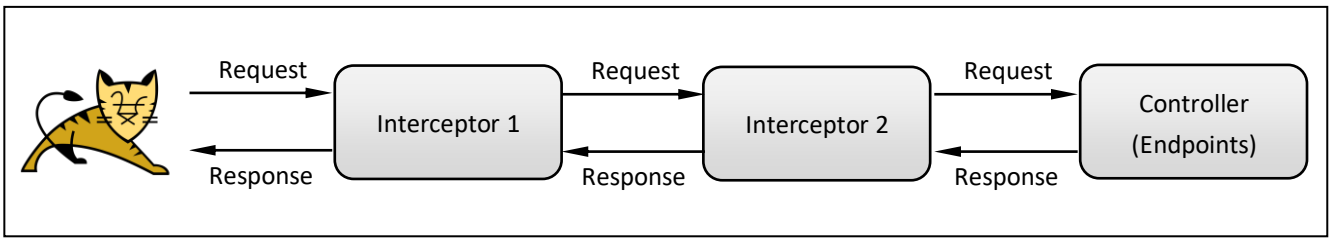
Filter is called twice



Interceptor Chain

- **Interceptor Chain** is ordered set of Interceptors that are
  - called in **sequence** before sending HTTP Request to the Controller
  - called in **reverse order** before sending HTTP Response from the Controller
- Interceptor Chain allows you to organize complex Interceptor Logic into multiple Interceptor Classes.

Interceptors are called in reverse order during HTTP Response

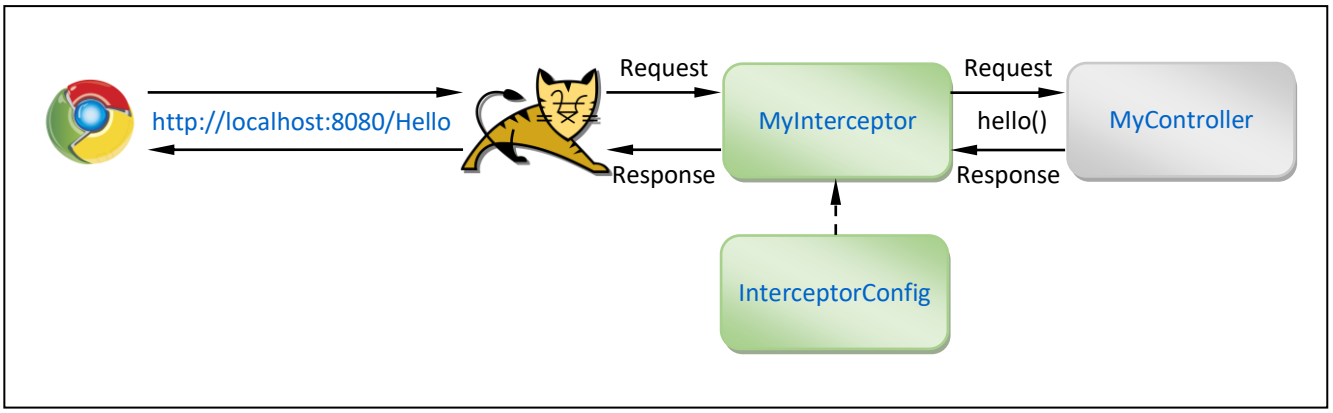


2.2.1 Create Interceptor - Implements HandlerInterceptor

Info [G]

This tutorial shows how to create Interceptor Class by using implements HandlerInterceptor.

Application Schema *[Results]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_filter_create (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Package: interceptors (inside main package)
  - Create Class: MyInterceptor.java (inside controllers package)
  - Create Class: InterceptorConfig.java (inside controllers package)

MyController.java

```
package com.ivoronline.springboot_interceptor.controllers;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    System.out.println("Controller:");
    return "Hello from Controller";
  }

}
```

MyInterceptor.java

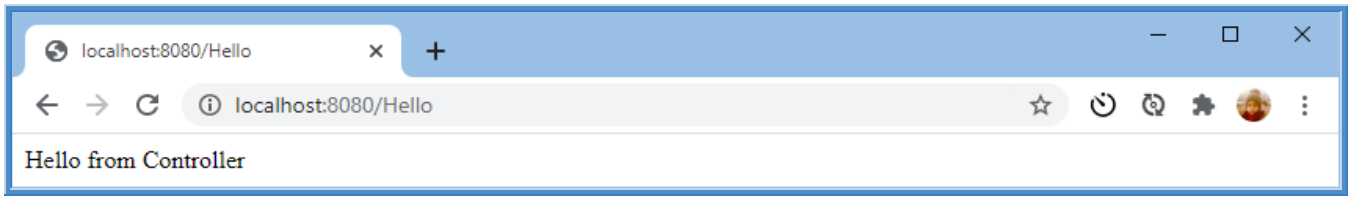
```
package com.ivoronline.springboot_interceptor.interceptors;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component
public class MyInterceptor implements HandlerInterceptor {

  //=====================================================================
  // PRE HANDLE
  //=====================================================================
  @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
    System.out.print("MyInterceptor: preHandle()       ");
    System.out.println(request.getMethod());
    return true;
  }

  //=====================================================================
  // POST HANDLE
  //=====================================================================
  @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
ModelAndView modelAndView) {
    System.out.print("MyInterceptor: postHandle()      ");
    System.out.println(response.getStatus());
  }

  //=====================================================================
  // AFTER COMPLETION
  //=====================================================================
  @Override
  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
Exception exception) {
    System.out.print("MyInterceptor: afterCompletion() ");
    System.out.println(response.getStatus());
  }

}
```

InterceptorConfig.java

```
package com.ivoronline.springboot_interceptor.interceptors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Component
public class InterceptorConfig extends WebMvcConfigurationSupport {

  @Autowired MyInterceptor myInterceptor;

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(myInterceptor);
  }
}
```

Results

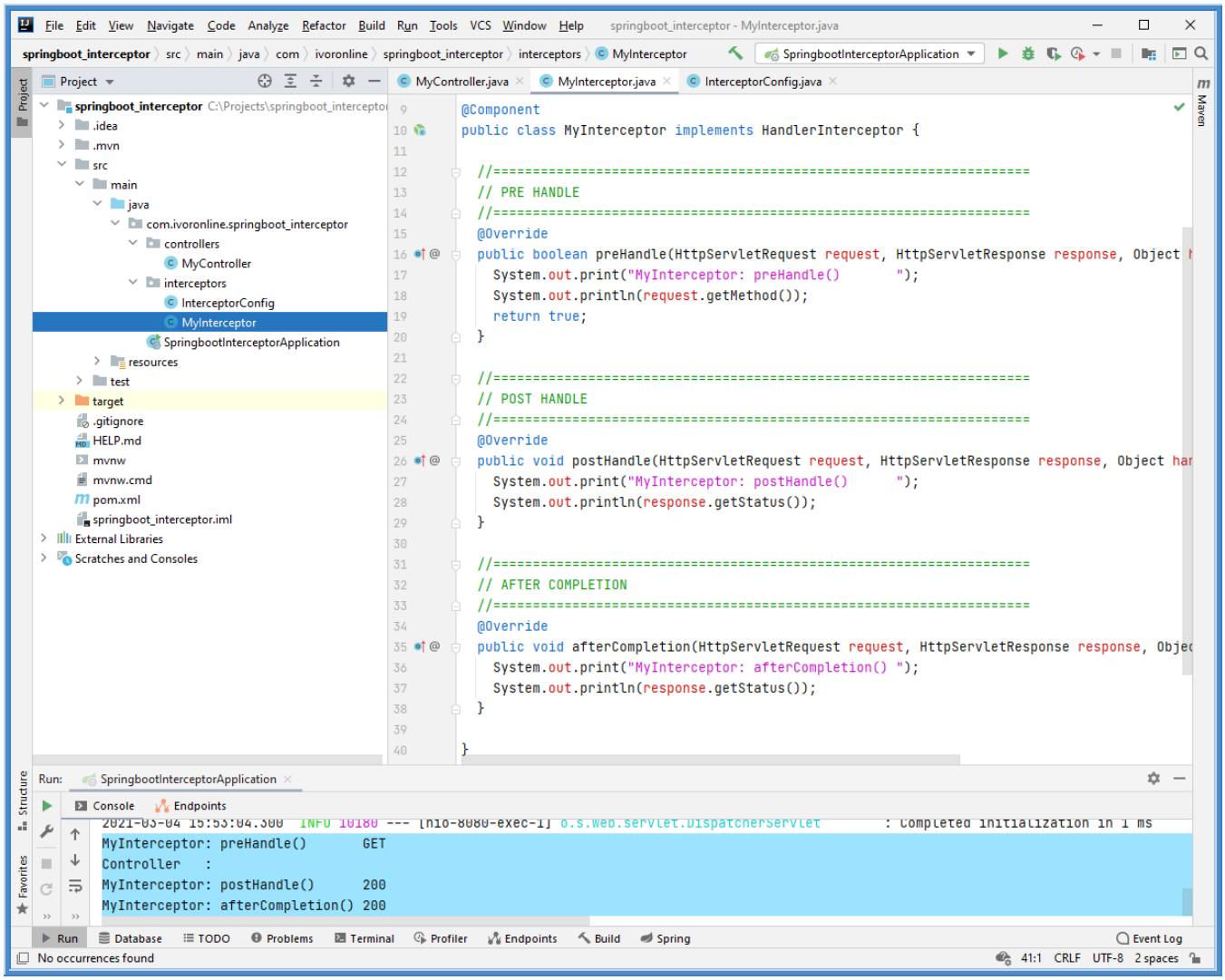
http://localhost:8080/Hello



Console

```
MyInterceptor: preHandle()       GET
Controller:
MyInterceptor: postHandle()      200
MyInterceptor: afterCompletion() 200
```

Application Structure



pom.xml

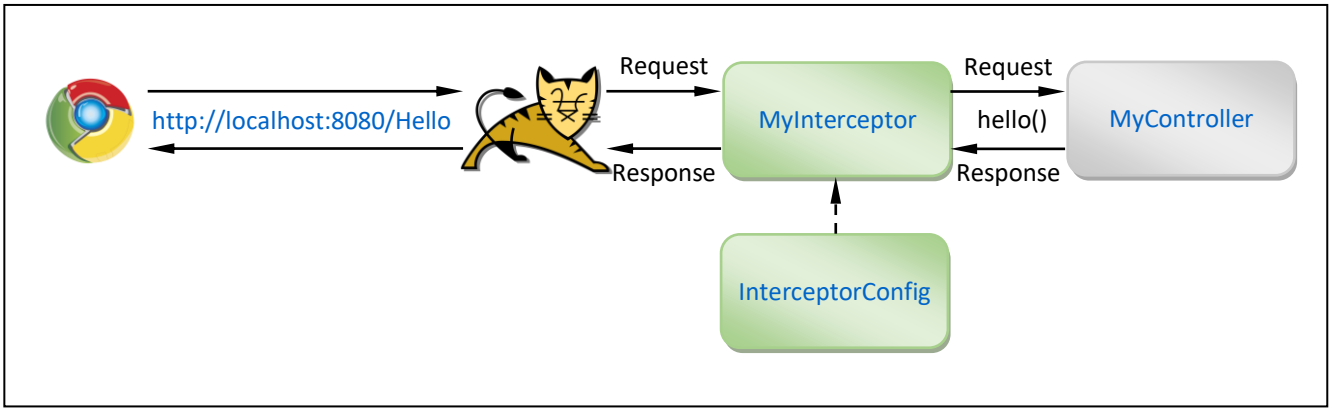
```
<dependencies>

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

</dependencies>
```

2.2.2 HTTP Request/Response - Get Parameters

Info [G]

- This tutorial shows how to use Interceptor to get HTTP Request/Response Parameters.

Application Schema *[Results]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_filter_create (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Package: interceptors (inside main package)
  - Create Class: MyInterceptor.java (inside controllers package)
  - Create Class: InterceptorConfig.java (inside controllers package)

MyController.java

```
package com.ivoronline.springboot_interceptor_gethttprequestresponseparameters.controllers;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    System.out.println("Controller:");
    return "Hello from Controller";
  }

}
```

MyInterceptor.java

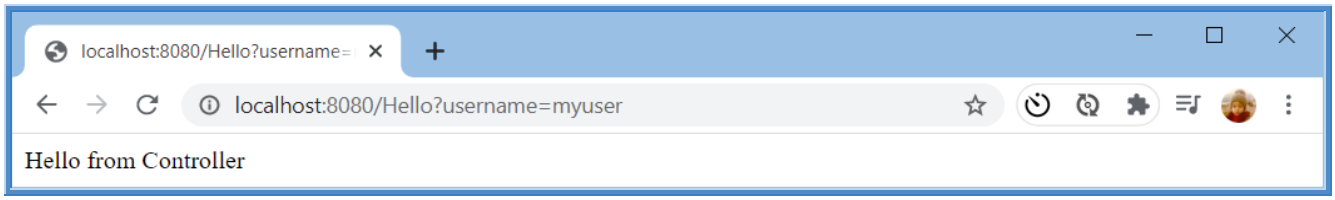
```
package com.ivoronline.springboot_interceptor_gethttprequestresponseparameters.interceptors;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component
public class MyInterceptor implements HandlerInterceptor {

  //====================================================================
  // PRE HANDLE
  //====================================================================
  @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
    System.out.println("MyInterceptor: preHandle()       ");
    System.out.println(request.getMethod    ());          //GET
    System.out.println(request.getProtocol  ());          //HTTP/1.1
    System.out.println(request.getServerName());          //localhost
    System.out.println(request.getServerPort());          //8080
    System.out.println(request.getParameter("username")); //myuser
    return true;
  }

  //====================================================================
  // POST HANDLE
  //====================================================================
  @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
ModelAndView modelAndView) {
    System.out.println("MyInterceptor: postHandle()      ");
    System.out.println(response.getStatus());   //200
  }

  //====================================================================
  // AFTER COMPLETION
  //====================================================================
  @Override
  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
Exception exception) {
    System.out.println("MyInterceptor: afterCompletion() ");
    System.out.println(response.getStatus());
  }

}
```

InterceptorConfig.java

```
package com.ivoronline.springboot_interceptor_gethttprequestresponseparameters.interceptors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Component
public class InterceptorConfig extends WebMvcConfigurationSupport {

  @Autowired MyInterceptor myInterceptor;

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(myInterceptor);
  }

}
```

Results

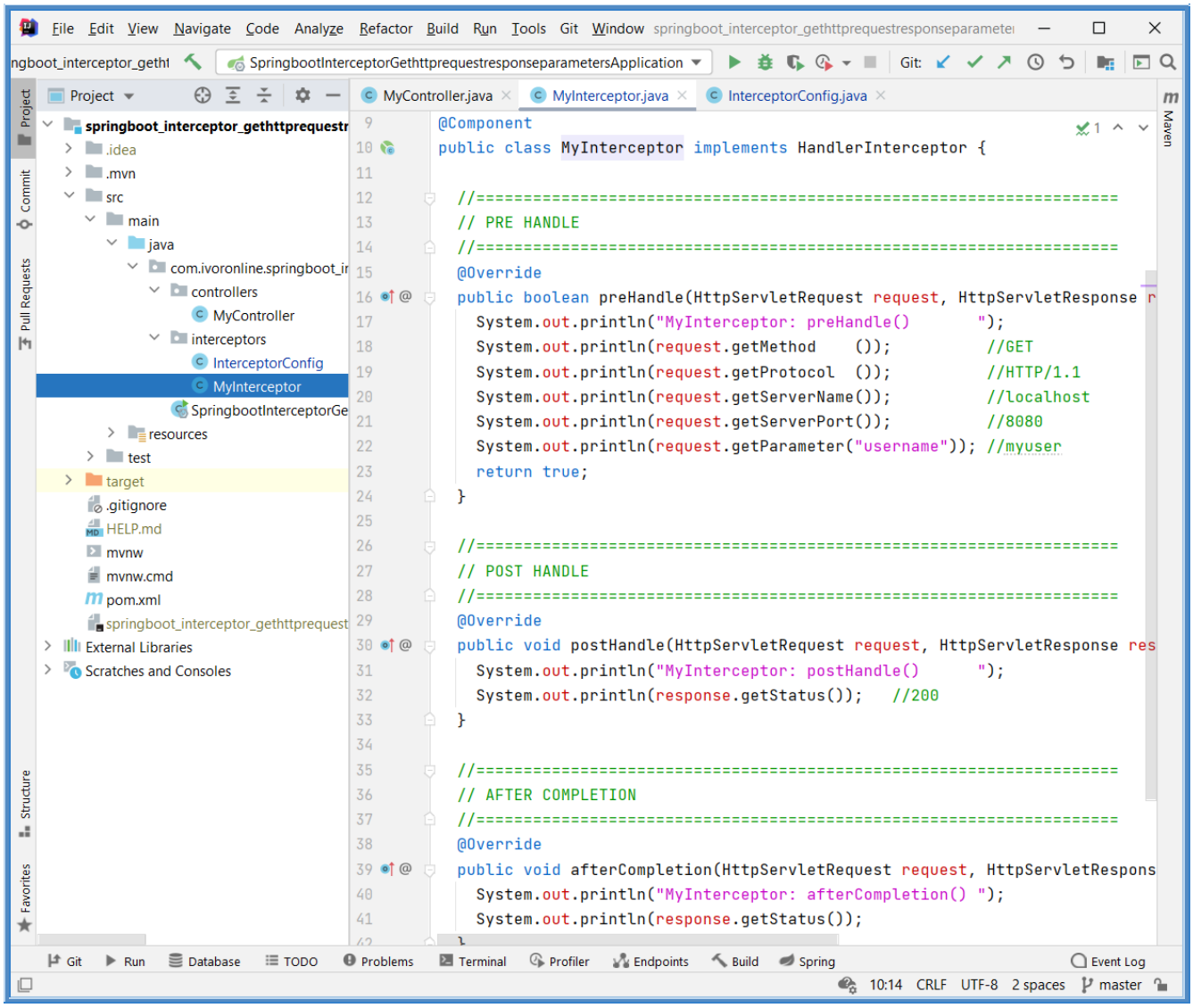
http://localhost:8080/Hello?username=myuser



Console

```
MyInterceptor: preHandle()
GET
HTTP/1.1
localhost
8080
myuser

Controller:

MyInterceptor: postHandle()
200

MyInterceptor: afterCompletion()
200
```

Application Structure



pom.xml

```
<dependencies>

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

</dependencies>
```

3 Logging

Info

- Following tutorials show how to use
  - Slf4j with Logback or Log4j to log events to Console, File or Database
  - AOP Annotations to separate logging from business logic

3.1 Slf4j

Info

- Following tutorials show how to use **SLF4J** with either
  - default **Logback** implementation (if no other logging dependencies can be detected in pom.xml)
  - **Log4j2** implementation (if you include Log4j2 dependencies in pom.xml)
- For its internal logging Spring Boot uses
  - **Jakarta Commons logging (JCL)** interface (equivalent to SLF4J)
  - **Logback** implementation (equivalent to Log4J)
- Available **log levels** are : ERROR, WARN, INFO, DEBUG, TRACE.
  Selected Log Level will display log lines belonging to that Log Level and those **above** it.
  Default log level is INFO which displays: ERROR, WARN, INFO.
- Logging can be **configured** through application.properties as shown below.
- Log Configuration allows you to choose
  - **Log Level** (display log lines belonging to that Log Level and those above it)
  - **Appender** (defines where to log data: Console, File, DatedFile. You can choose multiple destinations)
  - **Layout** (defines how to format log lines: Simple, Pattern, HTML)

Pattern Parameters

PARAMETER	EXAMPLE	DESCRIPTION
%p	INFO	Print priority of the logging event
%m	Some error occurred	Print message of the logging event
%c{2}	controllers.MyController	Print last 2 components of Class Path/Name
%d	2021-03-16 18:17:17,091	Print date
%%	%	Print %
%n	\n	Print new line character

Configuration through Properties

- How Properties are used is very confusing because of following three problems
  - each Appender has two references (instead of one)
    - MyAppender1 is used to reference all the lines that belong to the same Appender
    - MyConsoleAppender1Name is used to reference Appender from the Logger (instead of using MyAppender1)
  - appenderRef has confusing name, it should be called appenderRefs since it holds references to multiple Appenders
  - appenderRef is followed by reference name (like myappender1) even though these names are never used

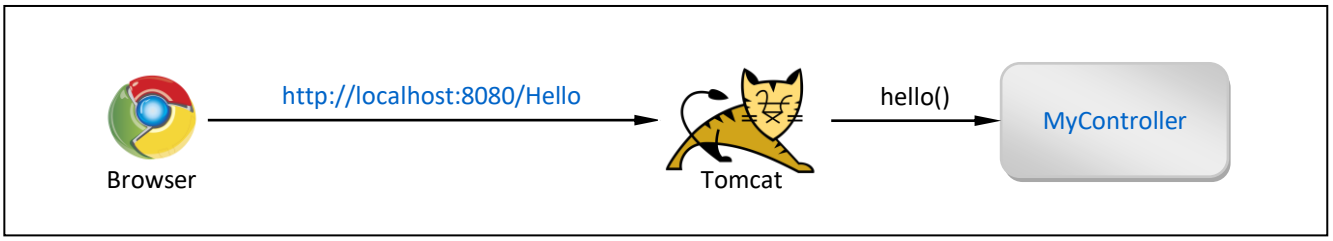
```
#APPENDERS
appender.MyAppender1.type                    = Console
appender.MyAppender1.name                    = MyConsoleAppender1Name

appender.MyAppender2.type                    = Console
appender.MyAppender2.name                    = MyConsoleAppender2Name
appender.MyAppender2.layout.type             = PatternLayout
appender.MyAppender2.layout.pattern          = MyAppender2: %d %p %c{2} %m %n

#LOGERS
logger.MyLogger1.name                        = com.ivoronline.springboot_log_log4j_config_xml.controllers
logger.MyLogger1.level                       = info
logger.MyLogger1.appenderRef.myappender1.ref = MyConsoleAppender1Name
logger.MyLogger1.appenderRef.myappender2.ref = MyConsoleAppender2Name

#DISABLE ROOT LOGGER
rootLogger.level = error
```

3.1.1 Using @Slf4j

Info [G] [R]

- This tutorial shows how to use Lombok's **@Slf4j** Annotation to work with Slf4J.
- This Annotation automatically generates following code so that we don't have to repeat it in every Class we want to log
  `private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExampleOther.class);`

Application Schema *[Results]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @RequestMapping, Tomcat Server
Developer Tools	**Lombok**	Enables: **@Slf4j** (injects LoggerFactory)

Procedure

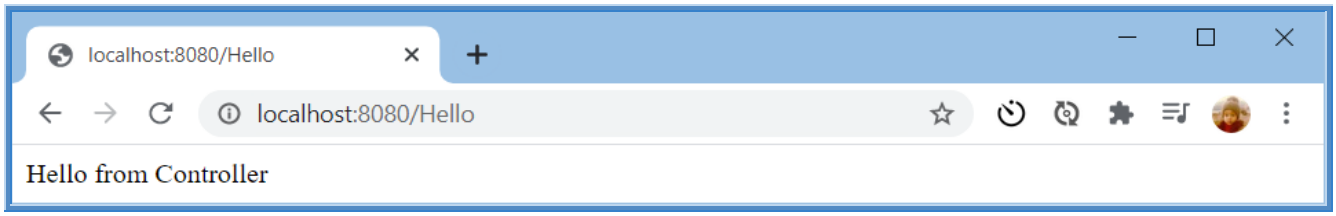
- Create Project: springboot_log_lombok (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)

MyController.java

```
package com.ivoronline.springboot_log_lombok.controllers;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Slf4j
@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    log.info("Hello from Controller");
    return "Hello from Controller";
  }

}
```

Results

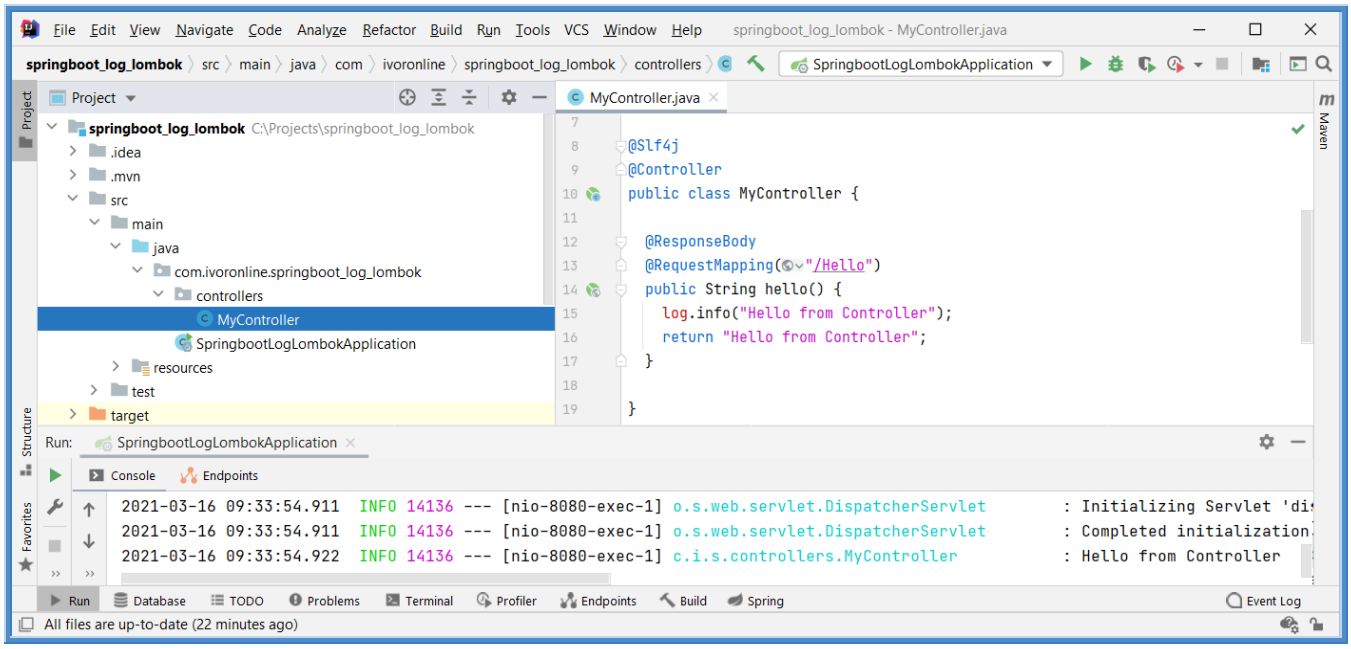
http://localhost:8080/Hello



Console

```
2021-03-16 08:59:53.267 INFO 14180 - [nio-8080-exec-2] c.i.s.controllers.MyController : Hello from Controller
```

Application Structure



pom.xml

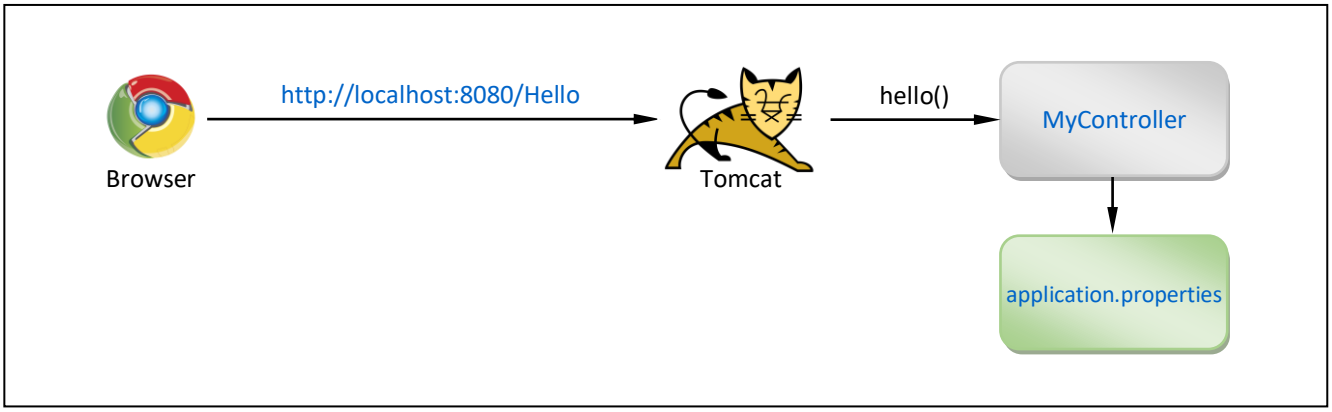
```
<dependencies>

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

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>

</dependencies>
```

3.1.2 Logback - Configure - Properties

Info **[G] [R]**

- This tutorial shows how to use application.properties to configure **Logback**.

Application Schema *[Results]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: @RequestMapping, Tomcat Server

Procedure

- Create Project: springboot_log_configuration (add Spring Boot Starters from the table)
- Edit File: application.properties (inside main package)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)

application.properties

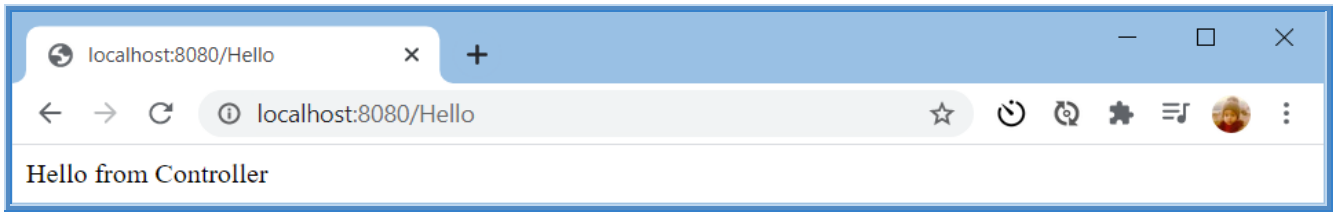
```
# CREATE GROUP
logging.group.mygroup        = com.ivoronline, com.ivoronline.springboot_log_logback_config_properties

# LOG LEVELS (for Group, Root, Package, Class)
logging.level.mygroup        = error
logging.level.root           = error
logging.level.com.ivoronline = error
logging.level.com.ivoronline.springboot_log_logback_config_properties.controllers.MyController = info

# CONSOLE LOGGER
logging.pattern.console      = My Console Log: %d %p %c{0} %m %n

# FILE LOGGER
logging.file.name            = logs/FIle.log
logging.pattern.file         = My File Log: %d %p %c{0} %m %n

# ROLLING FILE LOGGER
logging.logback.rollingpolicy.file-name-pattern      = logs/archived/RollingFile_%d{dd.MM.yyyy}_%i.log
logging.logback.rollingpolicy.max-history            = 30
logging.logback.rollingpolicy.max-file-size          = 1KB
logging.logback.rollingpolicy.total-size-cap         = 10MB
logging.logback.rollingpolicy.clean-history-on-start = false
```

MyController.java

```
package com.ivoronline.springboot_log_logback_config_properties.controllers;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  Logger log = LoggerFactory.getLogger(MyController.class);

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    log.error("Some error occured");
    log.warn ("Some warn  occured");
    log.info ("Some info  occured");
    log.debug("Some debug occured");
    log.trace("Some trace occured");
    return "Hello from Controller";
  }

}
```

Results

http://localhost:8080/Hello



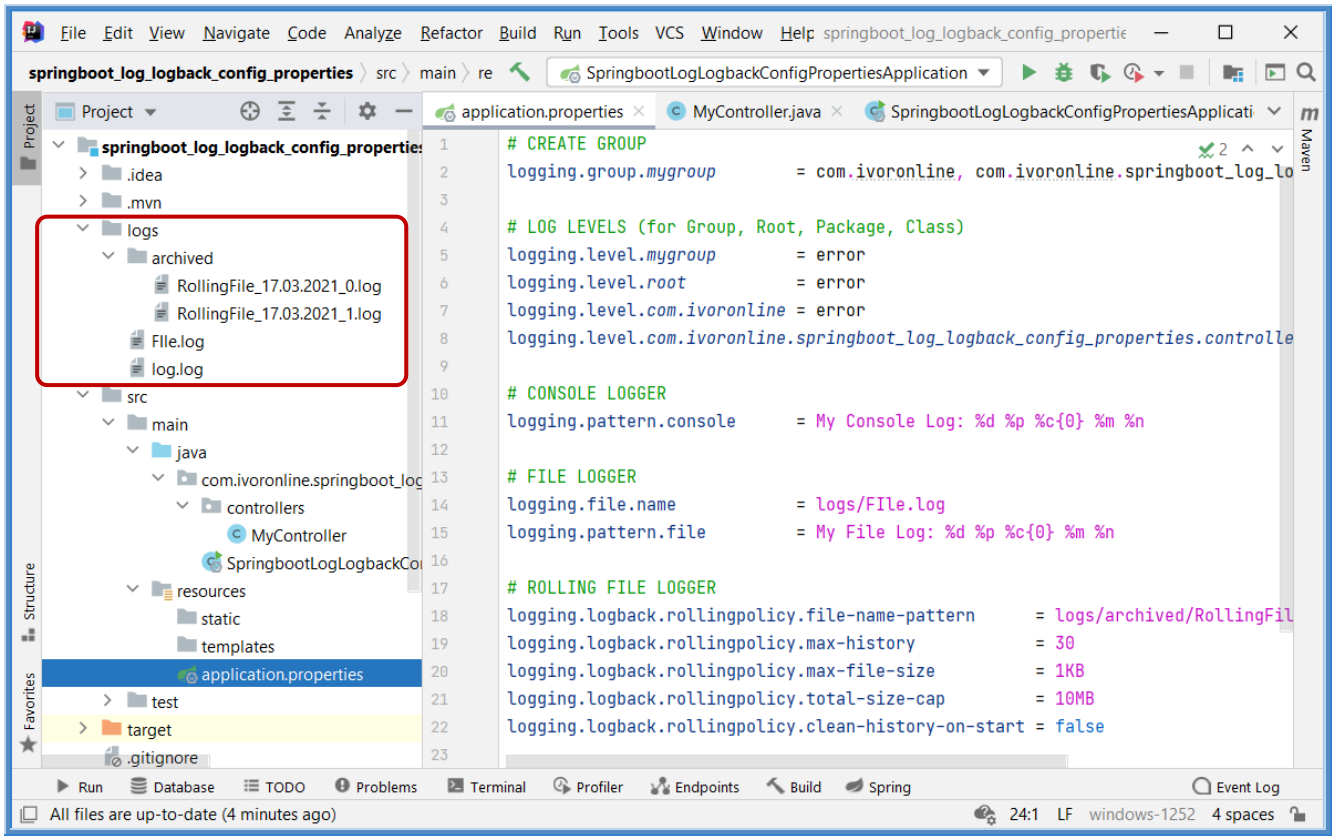
Console

```
My Console Log: 2021-03-17 15:00:31,057 ERROR MyController Some error occured
My Console Log: 2021-03-17 15:00:31,059 WARN MyController  Some warn  occured
My Console Log: 2021-03-17 15:00:31,059 INFO MyController  Some info  occured
```

logs\log.log

```
My File Log: 2021-03-17 15:00:31,057 ERROR MyController Some error occured
My File Log: 2021-03-17 15:00:31,059  WARN MyController Some warn  occured
My File Log: 2021-03-17 15:00:31,059  INFO MyController Some info  occured
```

Application Structure



pom.xml

```
<dependencies>

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

</dependencies>
```

3.2 AOP Annotations

Info [R]

- Following tutorials show how to use AOP Annotations to log events.
- AOP stands for Aspect Orientated Programming.
- AOP is a way for adding behavior to existing code without modifying that code.
- By using Annotations business logic code isn't polluted with logging code.

Examples

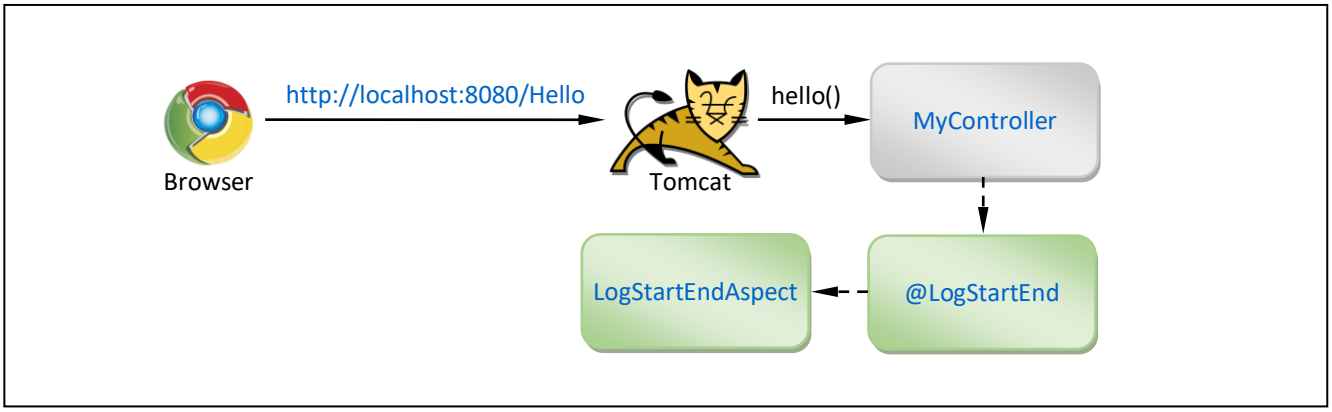
```
Signature method         = joinPoint.getSignature();      //String com.ivoronline...MyController.hello()
String    classPathName  = method.getDeclaringTypeName(); //com.ivoronline...controllers.MyController
String    methodName     = method.getName();              //hello

Class     methodClass    = method.getDeclaringType();     //class com.ivoronline...controllers.MyController
String    classPathName2 = methodClass.getName();         //com.ivoronline...controllers.MyController
String    packageName    = methodClass.getPackageName();  //com.ivoronline...controllers
String    className      = methodClass.getSimpleName();   //MyController
```

3.2.1 Log Start End

Info [G] [R]

- This tutorial shows how to use AOP Annotation to Log Method's Execution Time.

Application Schema *[Result]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_aop_logging (add Spring Boot Starters from the table)
- Edit File: pom.xml (add AOP dependency)
- Create Package: log (inside main package)
  - Create Class: LogStartEnd.java (inside log package)
  - Create Class: LogStartEndAspect.java (inside log package)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)

pom.xml

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

LogStartEnd.java

```
package com.ivoronline.springboot_aop_log_startend.log;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogStartEnd { }
```

LogStartEndAspect.java

```
package com.ivoronline.springboot_aop_log_startend.log;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LogStartEndAspect {

  @Around("@annotation(LogStartEnd)")
  public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

    //GET METHOD PARAMETERS
    String methodName = joinPoint.getSignature().getName();                          //hello
    String className  = joinPoint.getSignature().getDeclaringType().getSimpleName(); //MyController

    //EXECUTE BEFORE METHOD
    System.out.println("STARTED METHOD: " + className + "." + methodName + "()");

    //EXECUTE METHOD
    Object proceed = joinPoint.proceed();

    //EXECUTE AFTER METHOD
    System.out.println("ENDED   METHOD: " + className + "." + methodName + "()");

    //RETURN METHOD RESULT
    return proceed;

  }

}
```

MyController.java

```
import com.ivoronline.springboot_aop_log_startend.log.LogStartEnd;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @LogStartEnd
  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    return "Hello from Controller";
  }

}
```

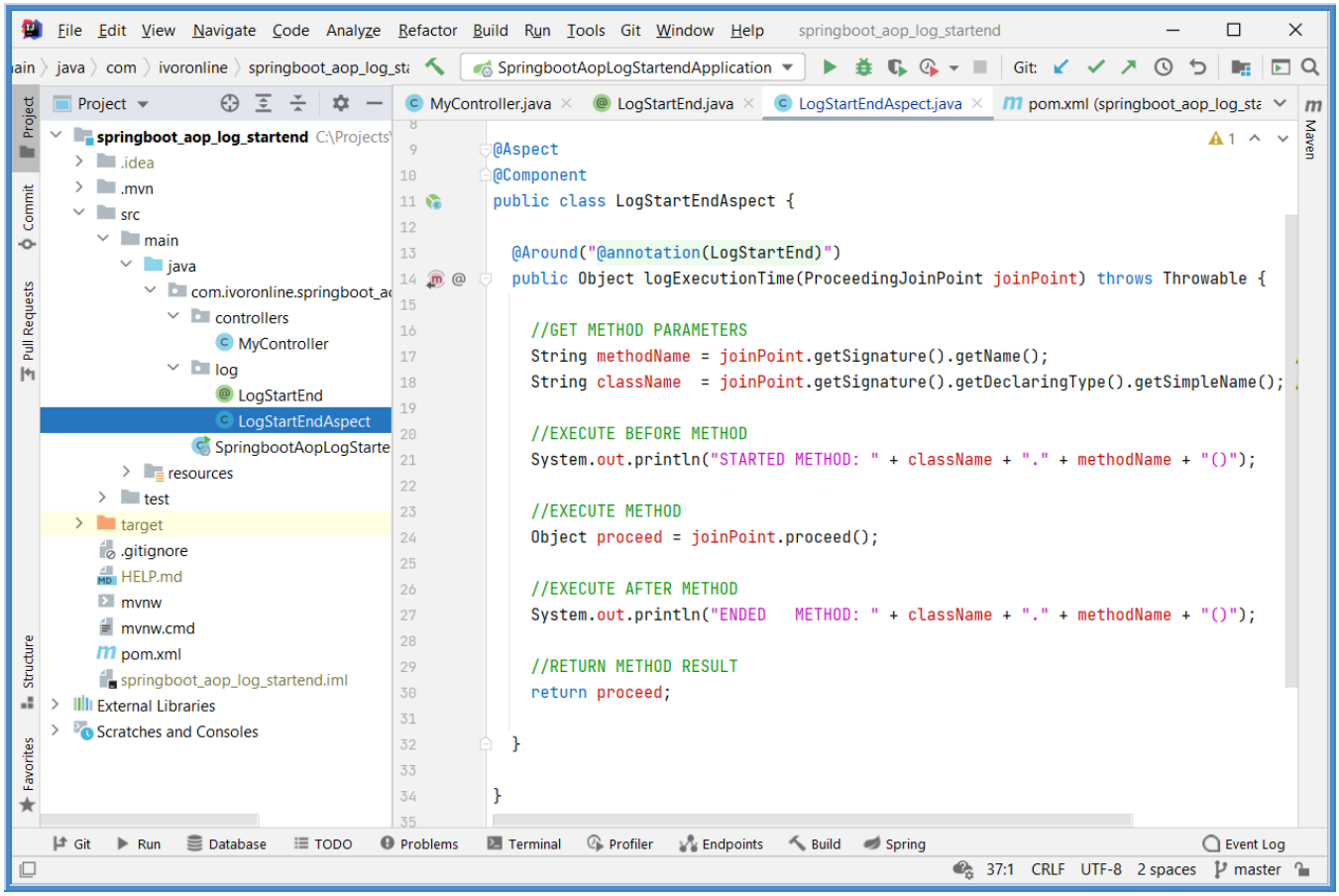
Result

http://localhost:8080/Hello

Console

```
STARTED METHOD: MyController.hello()
ENDED   METHOD: MyController.hello()
```

Run Test Class: PersonTest



pom.xml

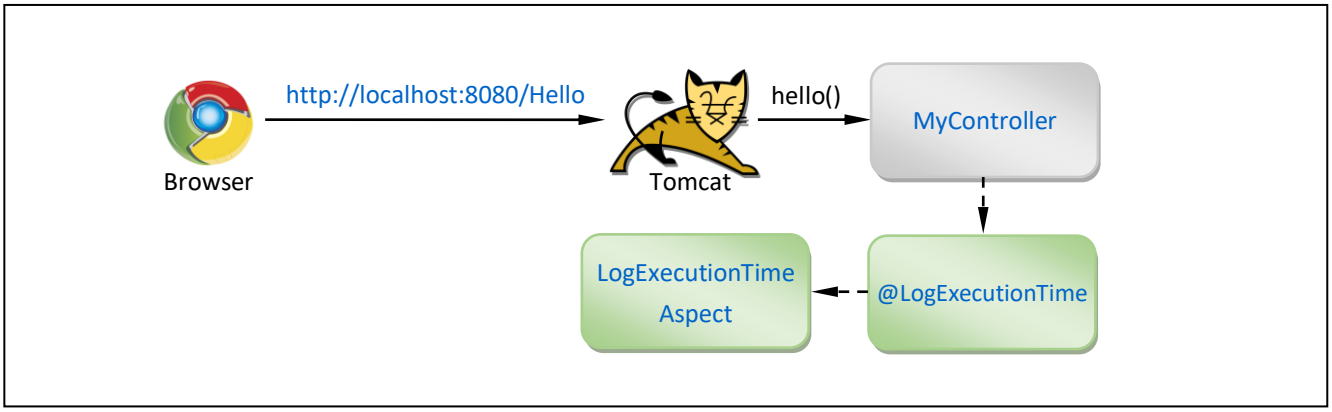
```
<dependencies>

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

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

</dependencies>
```

3.2.2 Log Execution Time

Info [G] [R]

- This tutorial shows how to use AOP Annotation to Log Method's Execution Time.

Application Schema *[Result]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_aop_logging (add Spring Boot Starters from the table)
- Edit File: pom.xml (add AOP dependency)
- Create Package: log (inside main package)
  - Create Class: LogExecutionTime.java (inside log package)
  - Create Class: LogExecutionTimeAspect.java (inside log package)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)

pom.xml

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

LogExecutionTime.java

```
package com.ivoronline.springboot_aop_logging.log;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogExecutionTime { }
```

LogExecutionTimeAspect.java

```
package com.ivoronline.springboot_aop_logging.log;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LogExecutionTimeAspect {

  @Around("@annotation(LogExecutionTime)")
  public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

    //EXECUTE BEFORE METHOD
    long start = System.currentTimeMillis();

    //EXECUTE METHOD
    Object proceed = joinPoint.proceed();

    //EXECUTE AFTER METHOD
    long executionTime = System.currentTimeMillis() - start;
    System.out.println(joinPoint.getSignature() + " executed in " + executionTime + "ms");

    //RETURN METHOD RESULT
    return proceed;

  }

}
```

MyController.java

```
package com.ivoronline.springboot_aop_logging.controllers;

import com.ivoronline.springboot_aop_logging.log.LogExecutionTime;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @LogExecutionTime
  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    return "Hello from Controller";
  }

}
```

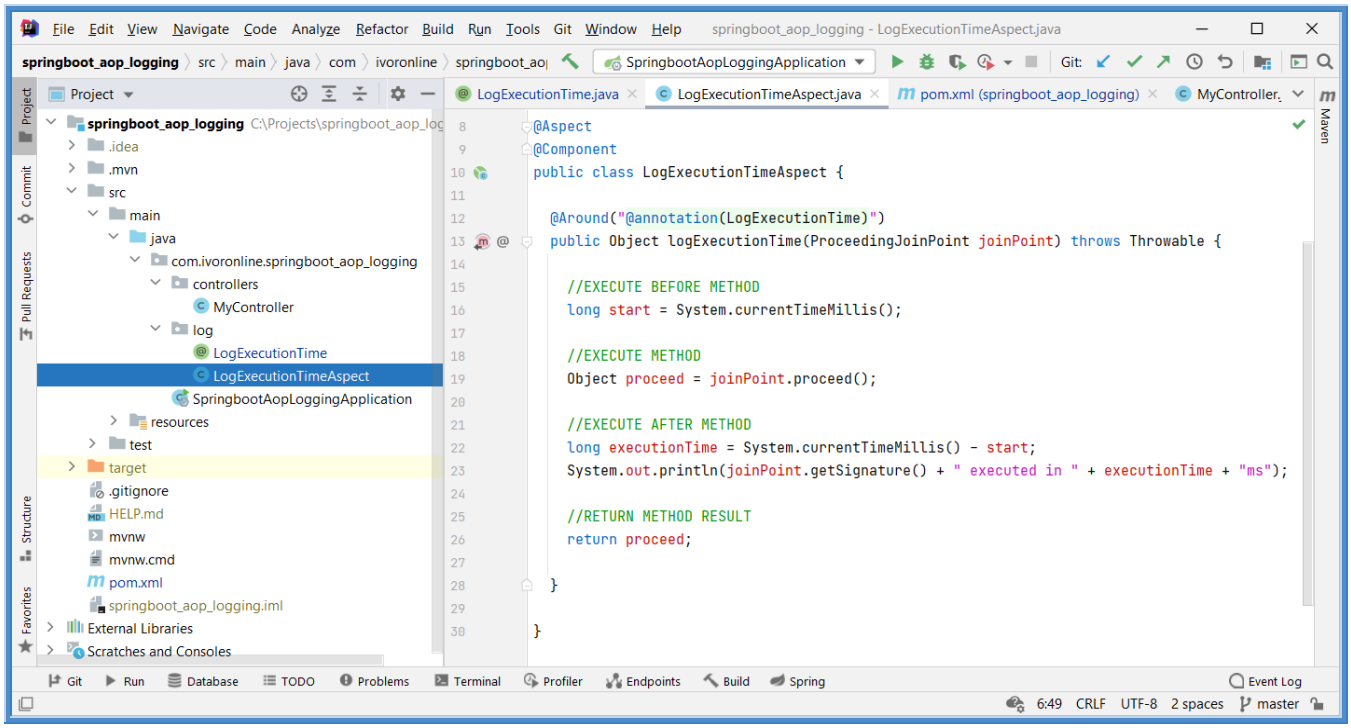
Result

http://localhost:8080/Hello

Console

```
String com.ivoronline.springboot_aop_logging.controllers.MyController.hello() executed in 10ms
```

Run Test Class: PersonTest



pom.xml

```
<dependencies>

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

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

</dependencies>
```

4 Testing

Info

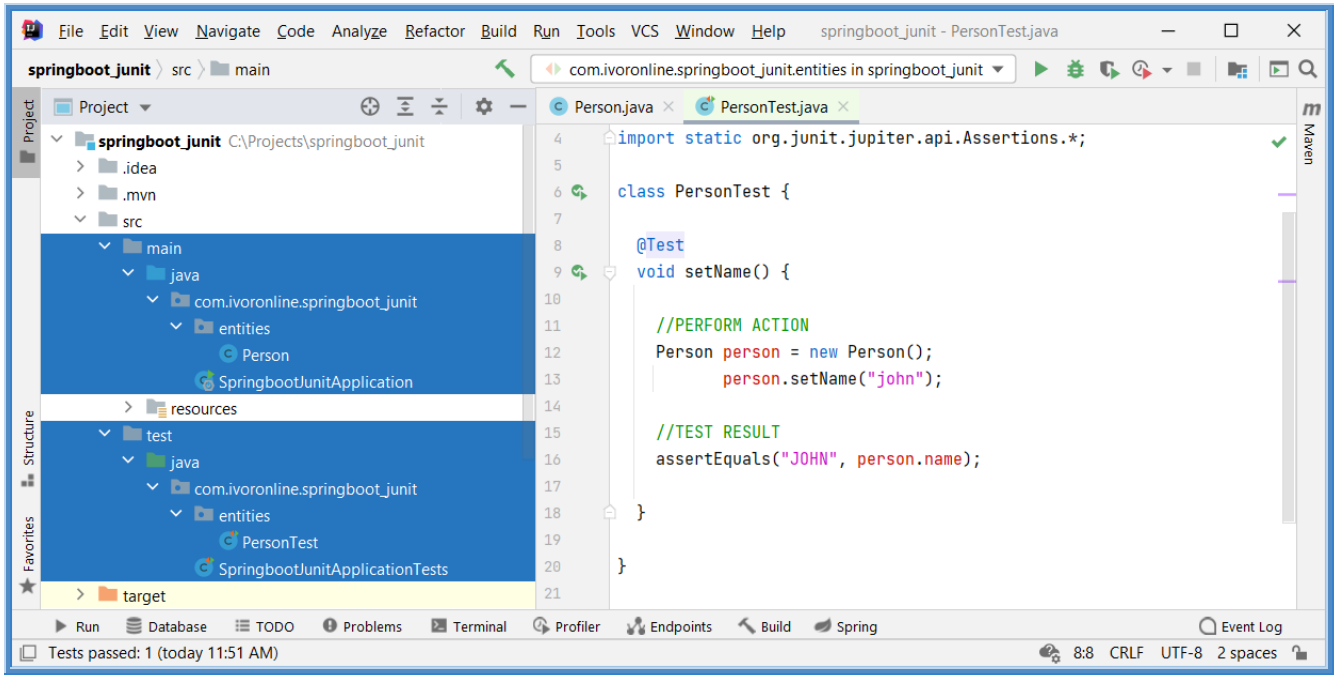
- Following tutorials show how to use different technologies to test Application.

4.1 JUnit 5

Info

- Following tutorials show how to use **JUnit 5** to test Application.
- Both Application and Test Classes will have the same Package structure (for easier navigation) where
  - **Application Classes** are organized under **src\main**
  - **Test Classes** are organized under **src\test**
- You can **Create Test Class** either
  - Manually (manually create Package and Test Class under src\test\java)
  - Automatically (IntelliJ automatically recreates Packages and creates Test Class under src\test\java)
- You can Run Tests by
  - Right Clicking on the Package or Class (inside Project Hierarchy)
  - Right Clicking on the Class or Method Name (inside opened Class File)
  - Clicking left from Class or Method Name (inside opened Class File)

Application Structure *(created Test Packages and Classes)*



4.1.1 Create Test Class - Automatically

Info

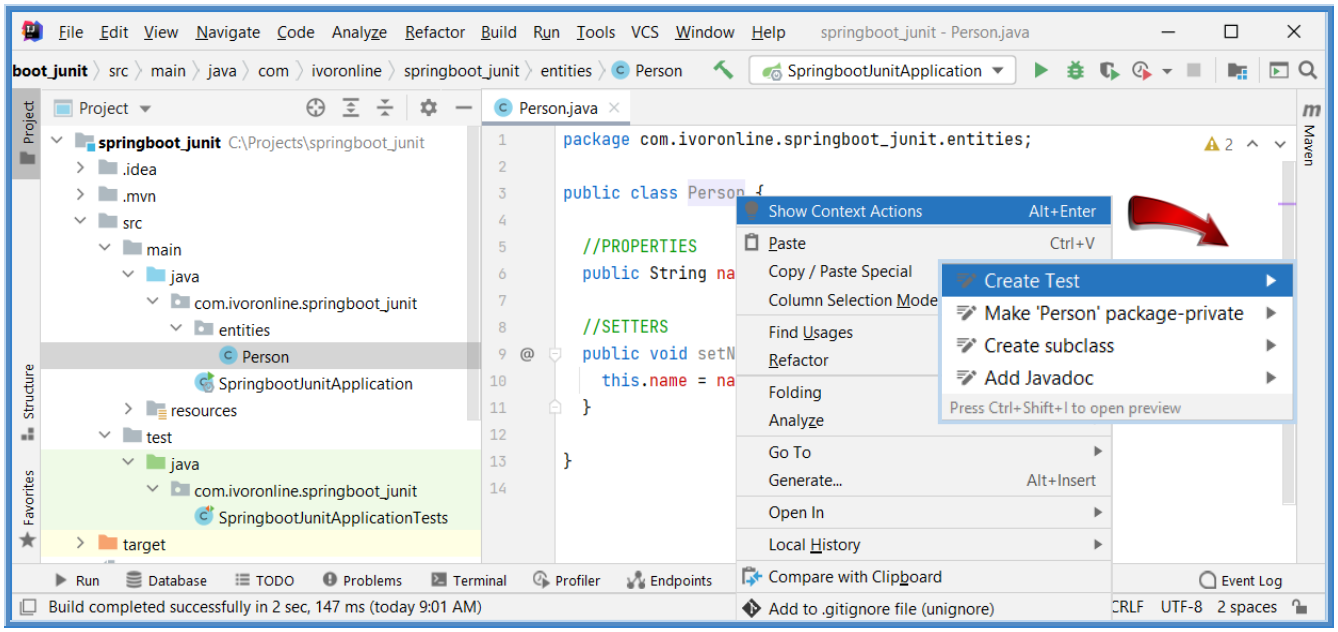
- This tutorial shows how IntelliJ can automatically create Test Class by selecting Methods you want to test.
- IntelliJ will create new Test Class under main\test\com.ivoronline.springboot_junit
  - created Test Class will have the same name as the Class that contains selected Methods but with additional **Test** suffix
    (if selected Methods were from the Class **Person** then IntelliJ will create Test Class named **PersonTest**)
  - IntelliJ will also recreate the same Packages
    (if Class Person was in the Package **entities**, then Class PersonTest will also be placed in the Package entities)

Content

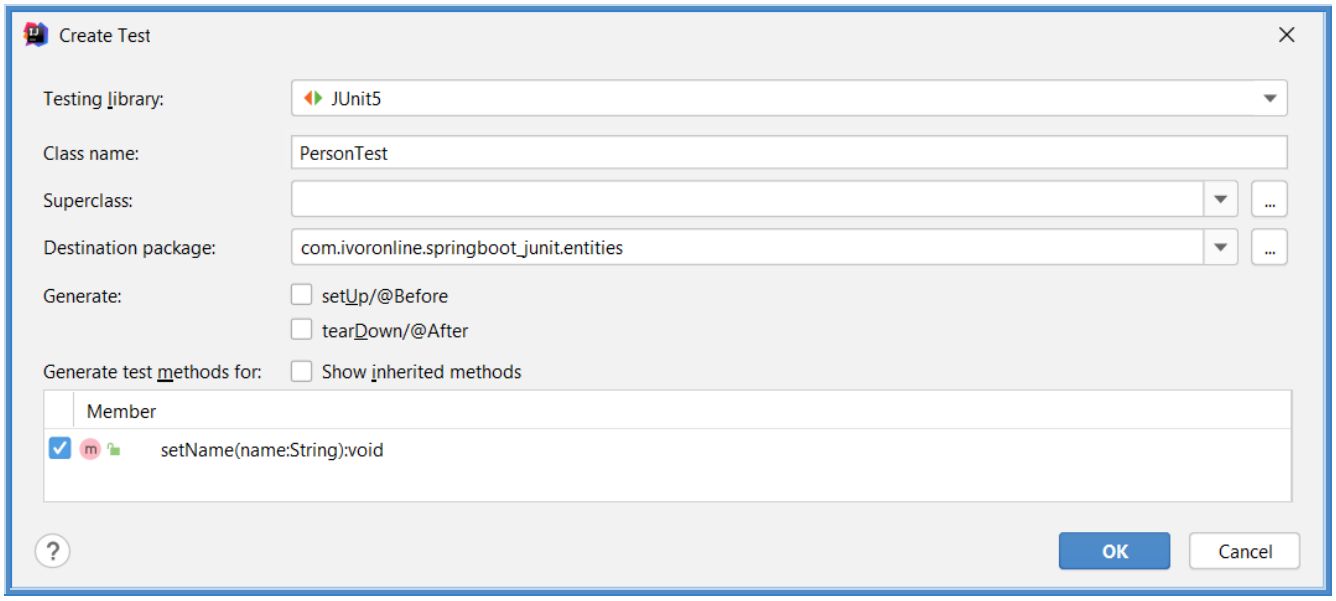
Create Test Class

- Choose one
  - RC on Class - Show Context Actions (on Class which Methods you want to test)
  - LC on Class - Alt + Enter
- Create Test
- Select Methods (which you want to test)
- OK (creates Test Class under the same Package structure as Class being tested)

RC on Class

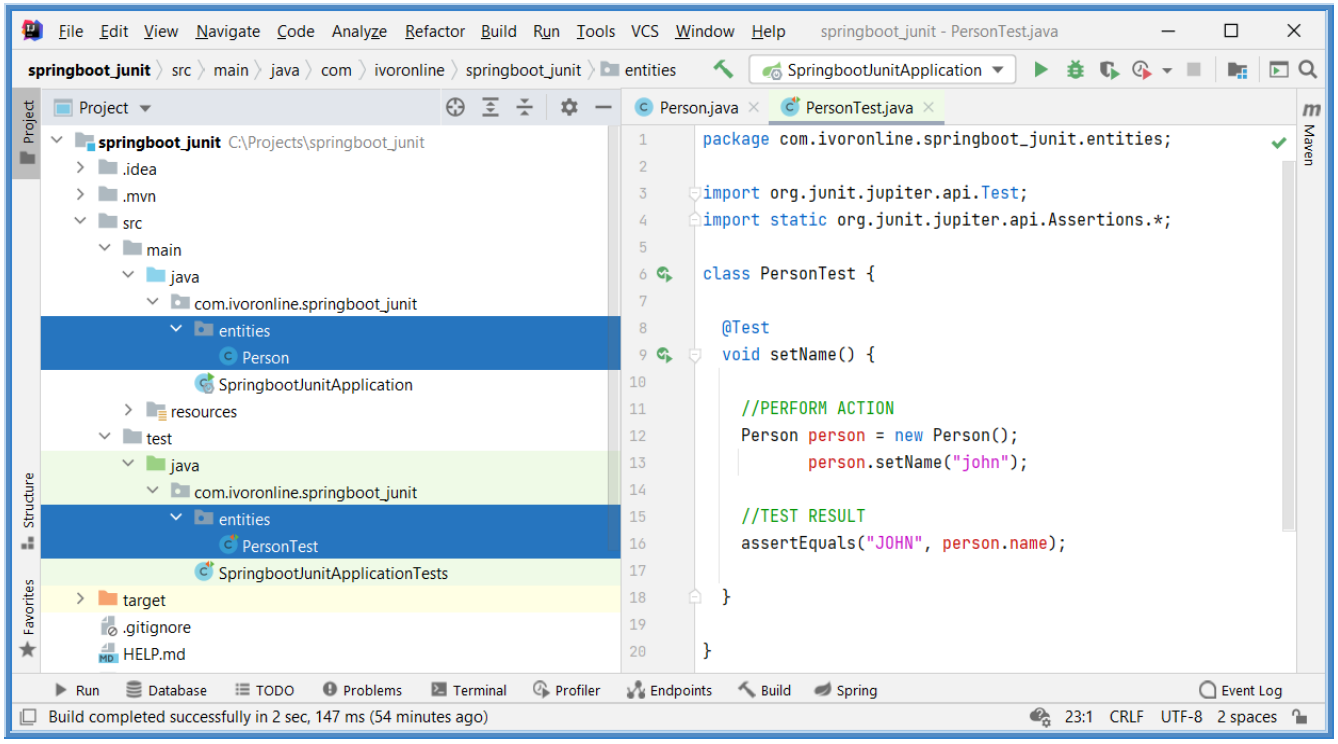


Select Methods *(which you want to test)*



Result

Automatically created Test Class *(with the same Package structure)*



4.1.2 Run Tests

Info

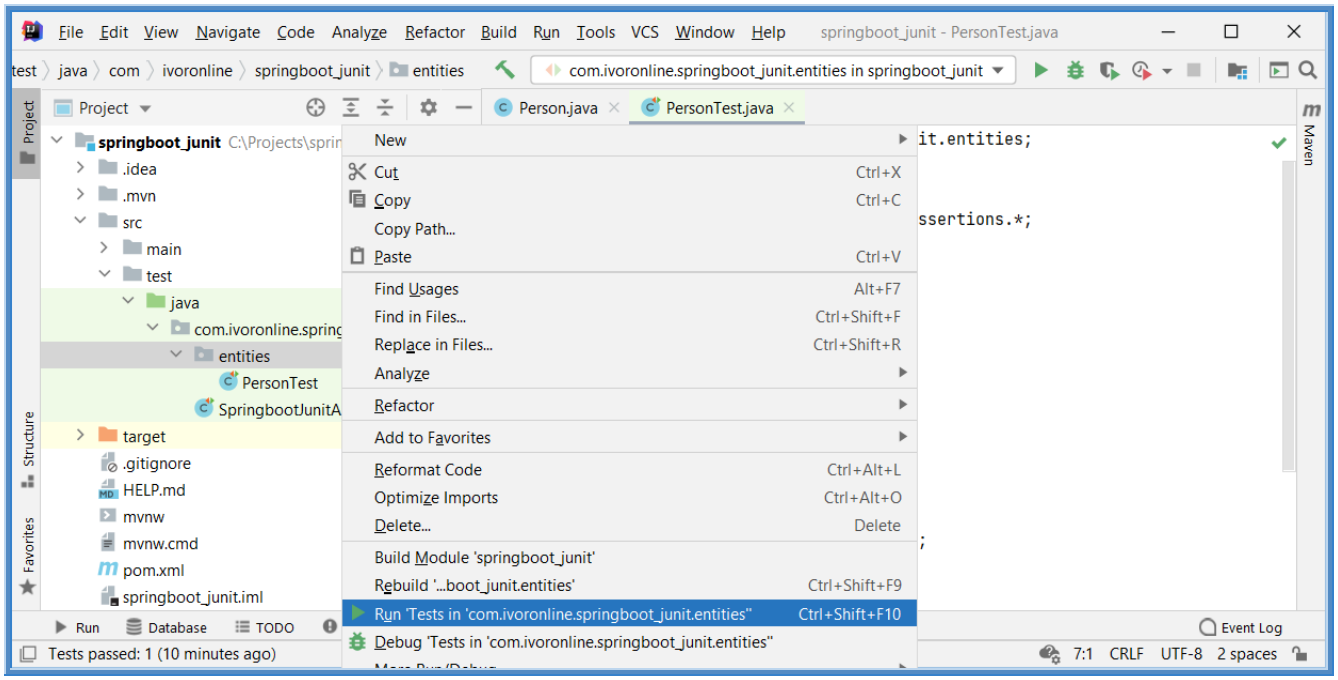
- This tutorial shows how to run all Tests within selected Test **Package** or **Class**, or how to run a single Test **Method**.
- You can Run Tests by
  - Right Clicking on the Package or Class (inside Project Hierarchy)
  - Right Clicking on the Class or Method Name (inside opened Class File)
  - Clicking left from Class or Method Name (inside opened Class File)

Content

Run Tests within Package

- RC on entities
- Run Tests in com.ivoronline.springboot_junit.entities

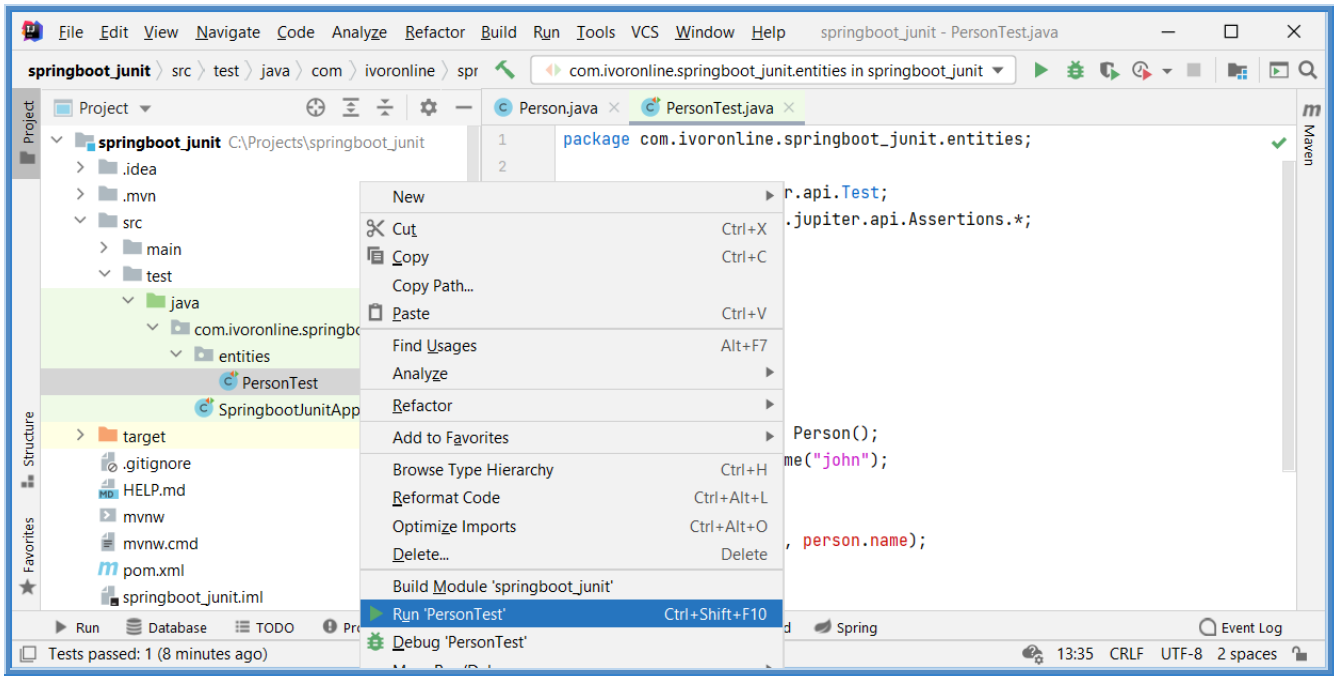
RC on Test Package - Run Tests in com.ivoronline.springboot_junit.entities



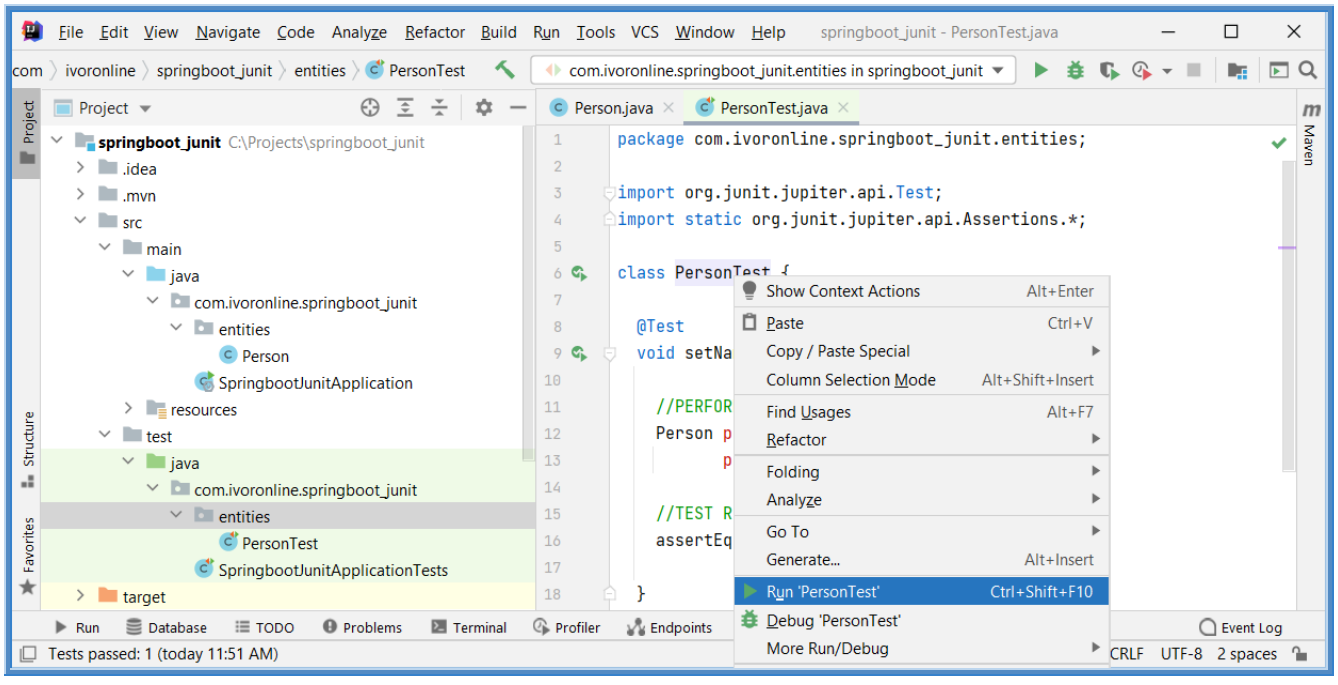
Run Tests within Class

- Option 1
  - RC on Test Class
  - Run PersonTest
- Option 2
  - Open Test Class
  - RC on Class Name
  - Run PersonTest
- Option 3
  - Open Test Class
  - Click on Run Test (left from Class Name)
  - Run PersonTest

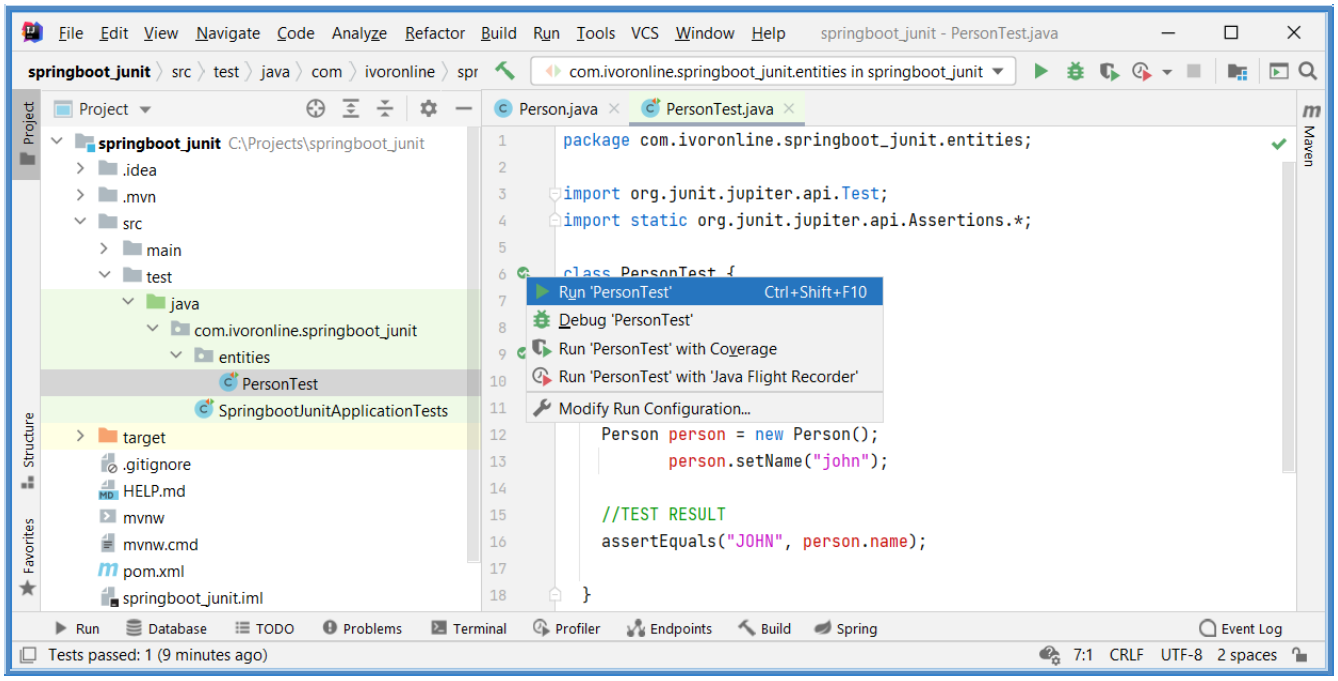
Option 1: RC on Test Class - Run PersonTest *(inside Project Explorer)*



Option 2: Open Test Class - RC on Class Name - Run PersonTest *(inside opened Class File)*



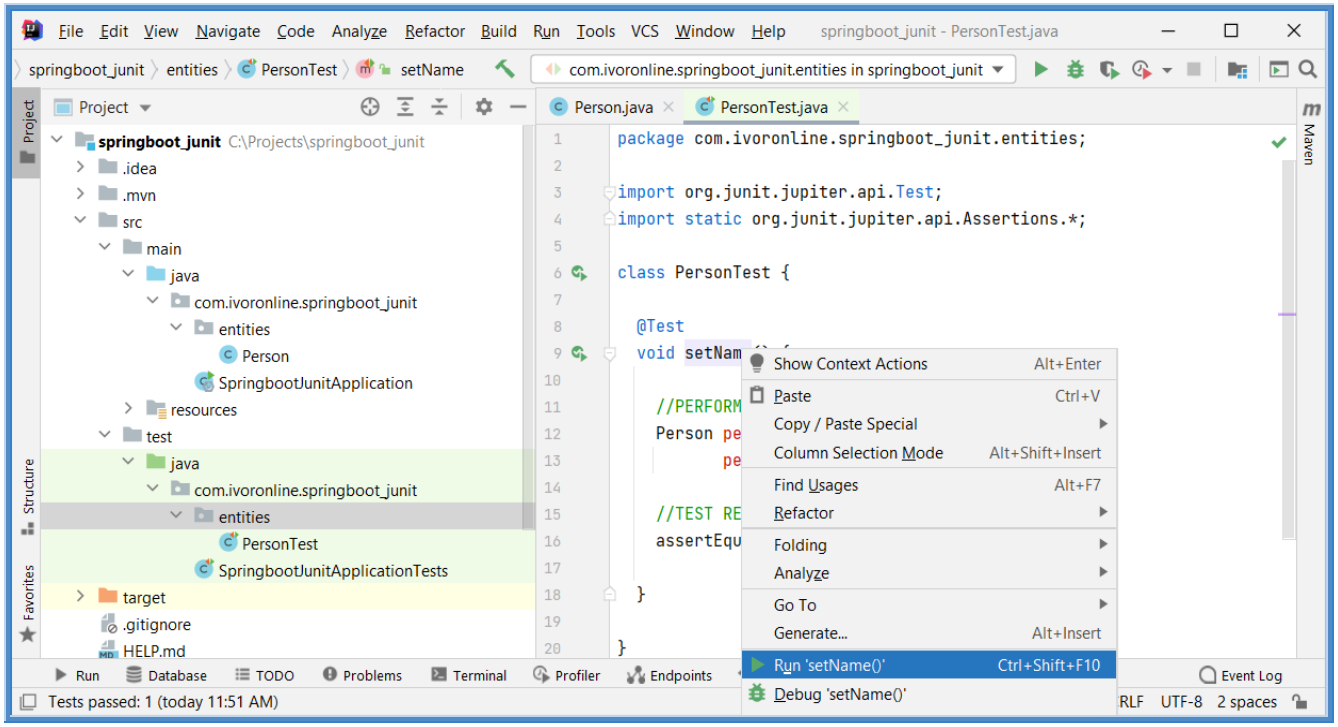
Option 3: Open Test Class - Click on Run Test (left from Class Name) - Run PersonTest *(inside opened Class File)*



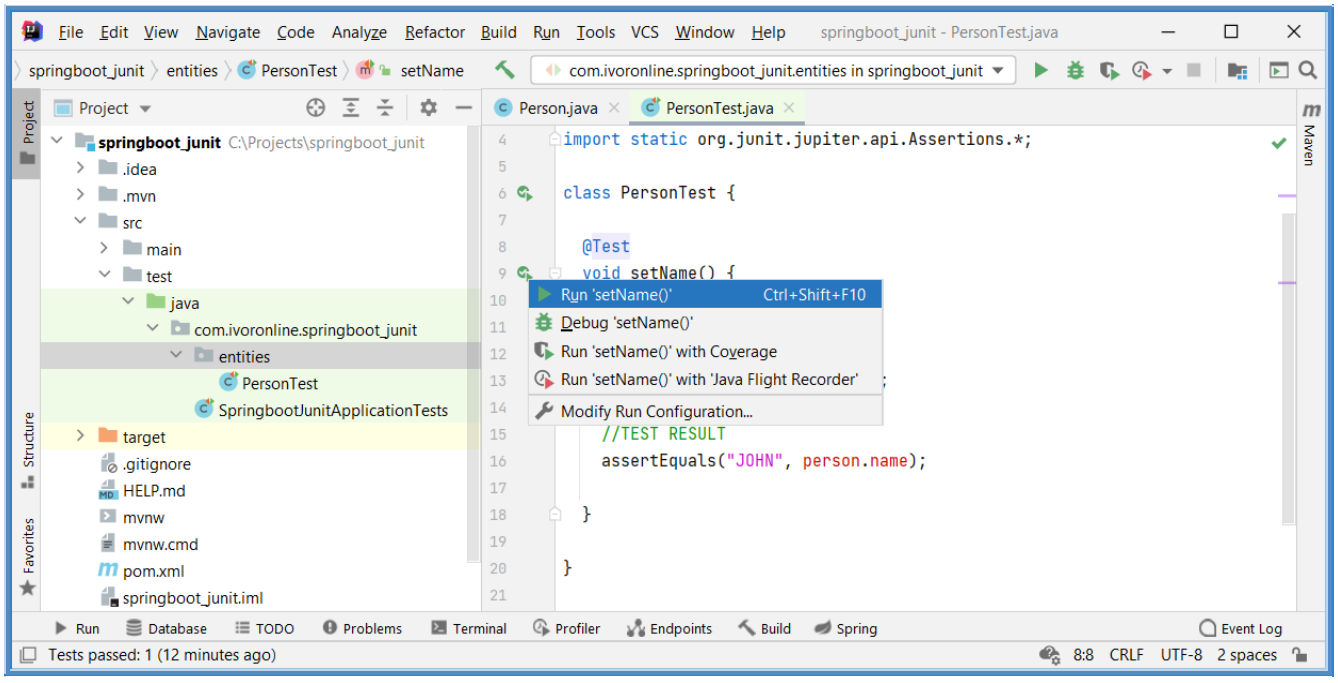
Run specific Test Method

- Option 1
  - Open Test Class
  - RC on Method Name
  - Run setName()
- Option 2
  - Open Test Class
  - Click on Run Test (left from Method Name)
  - Run setName()

Option 1: Open Test Class - RC on Method Name - Run setName()



Option 2: Open Test Class - Click on Run Test (left from Method Name) - Run setName()

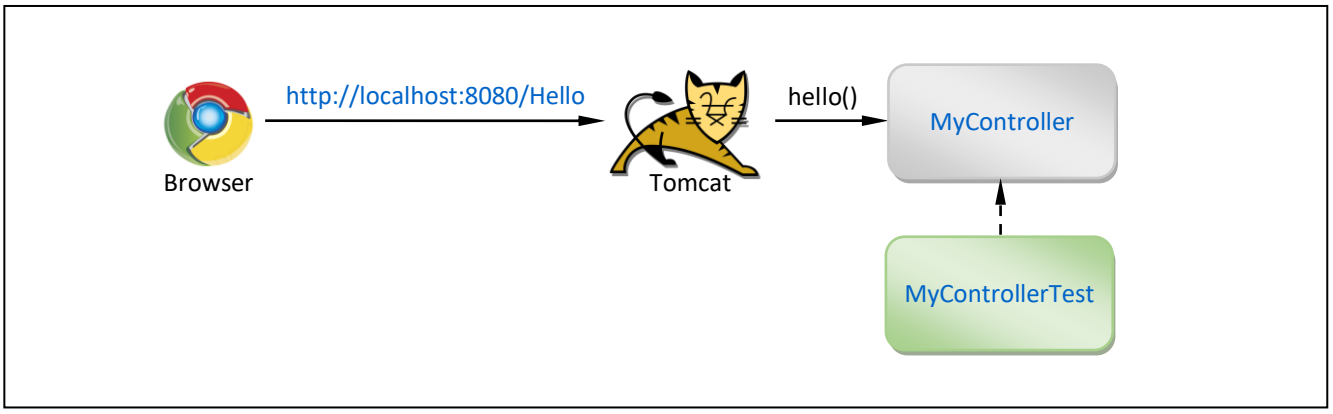


4.1.3 Example

Info [G]

- This tutorial shows a basic example of JUnit 5 Test Method.
- Tutorial isn't using @SpringBootTest which means that @Autowired isn't work which is why we use new MyController().
- This approach is useful when we don't need Spring Boot Context so that tests could run faster.

Application Schema *[Result]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_junit (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Test Class: MyControllerTest.java

MyController.java

```
package com.ivoronline.springboot_junit.controllers;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    return "Hello from Controller";
  }

}
```

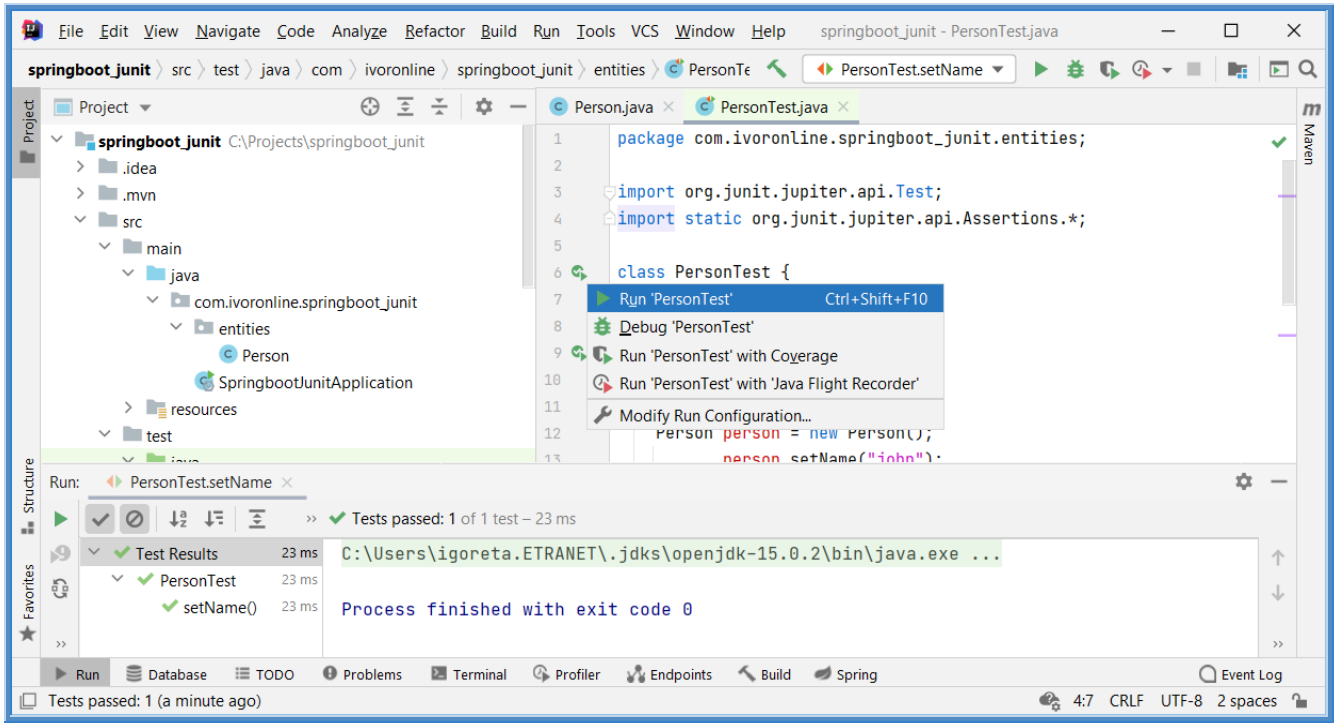
MyControllerTest.java

```
package com.ivoronline.springboot_junit.controllers;

import com.ivoronline.springboot_junit.controllers.MyController;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class MyControllerTest {

  @Test
  void hello() {

    //PERFORM ACTION
    MyController myController = new MyController();
    String       result       = myController.hello();

    //TEST RESULT
    assertEquals("Hello from Controller", result);

  }

}
```

Result

http://localhost:8080/Hello

Run Test Class: PersonTest



pom.xml

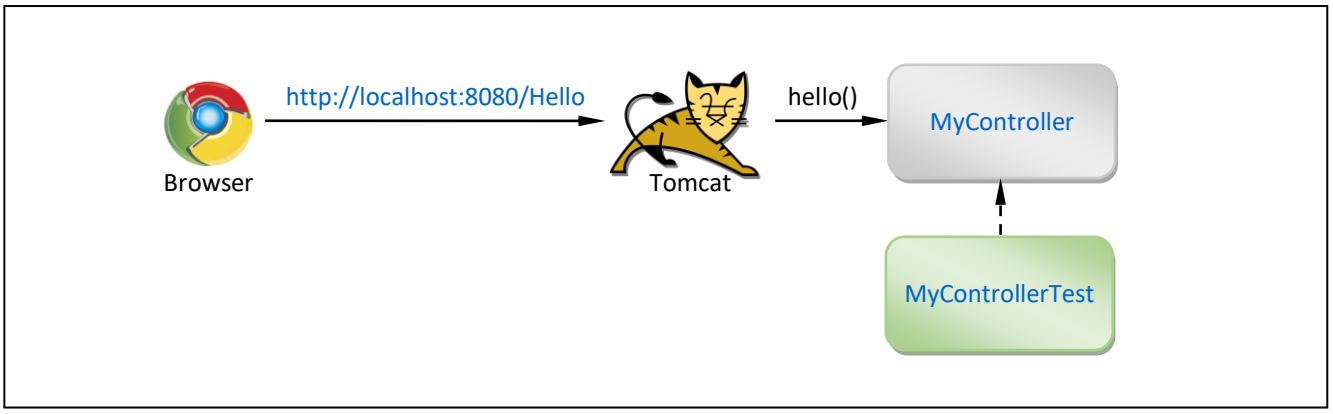
```
<dependencies>

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

</dependencies>
```

4.1.4 @SpringBootTest

Info [G]

- This tutorial shows how to use `@SpringBootTest` to create Spring Boot **Context** for testing.
  This will instantiate Controller and other Beans making `@Autowired` work inside Application and `@Test` Methods.
- Otherwise
  - `@Autowired` wouldn't work neither inside the Application nor in the Test Methods
  - therefore we would need to use `Controller controller = new Controller()` inside the `@Test` Method instead
  - and if inside Controller we have `@Autowired Repository repository`, `repository` will not get instantiated
- Creating Spring Boot Context is time consuming operation but it is done only once (and not for every Test Method).
  In other words Context gets cached between Test Methods remembering changes done by previous Test Methods.

Application Schema *[Result]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Procedure

- Create Project: springboot_junit_test_controller (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Test Class: MyControllerTest.java

MyController.java

```
package com.ivoronline.springboot_junit_test_controller.controllers;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    return "Hello from Controller";
  }

}
```

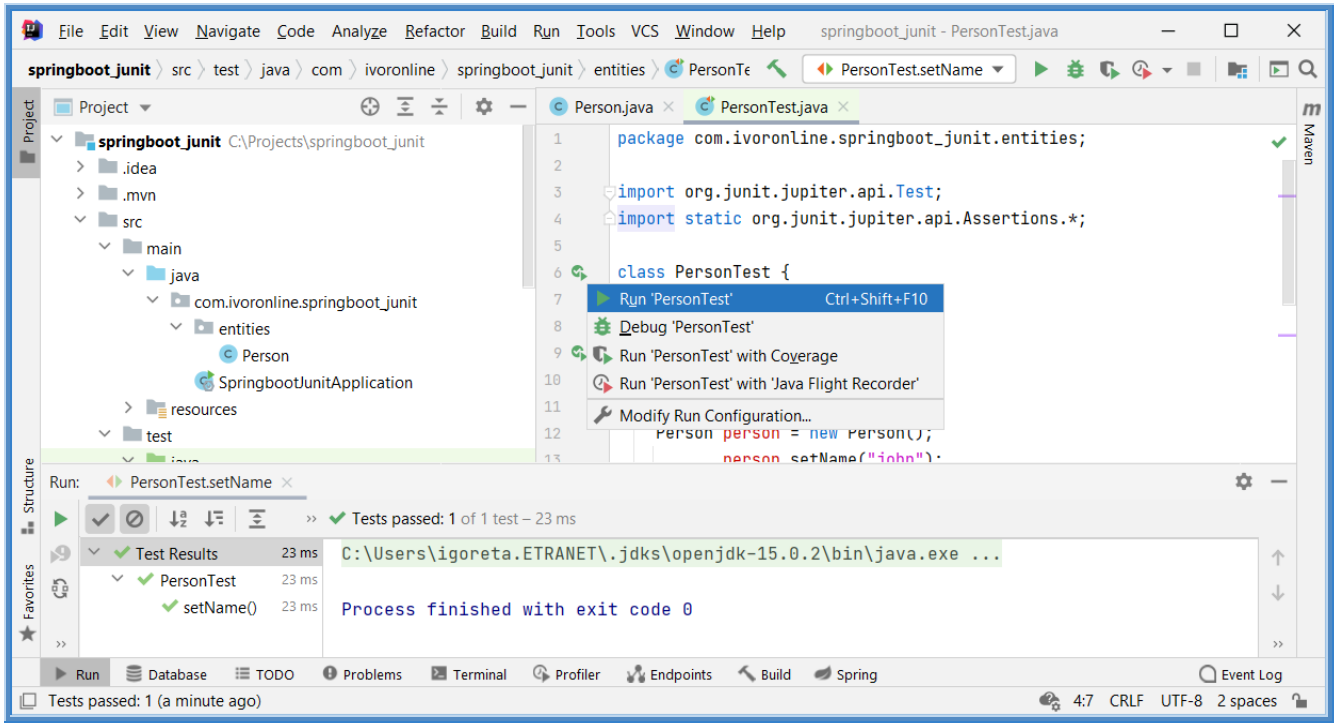
MyControllerTest.java

```
package com.ivoronline.springboot_junit_test_controller.controllers;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
class MyControllerTest {

  @Autowired MyController myController;

  @Test
  void hello() {
    String result = myController.hello();
    assertEquals("Hello from Controller", result);
  }

}
```

Result

http://localhost:8080/Hello

Run Test Class: PersonTest



pom.xml

```
<dependencies>

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

</dependencies>
```

4.2 Mockito

Info

- Following tutorials show how to use Mockito. (inside JUnit Tests)
- Mockito is used to **Mock Dependencies** (Classes and Methods that are called by the Method being tested)
  - first you **Mock Class** (you need to Mock Class in order to be able to Mock its Methods)
  - then you **Mock Methods** (of Mocked Class)
- To **Mock Method** means to **specify return value for specific Input Parameters**.
  To Mock Method first you need to Mock Class to which this Method belongs to.
  Class can be Mocked using either **@Mock** or **@Spy**.
  In both cases Mocked Method returns what is defined by Mockito and not what the actual implementation would return.

Mock Object

```
@Mock PersonRepository personRepositoryMock;
@Spy  PersonRepository personRepositoryMock;
```

Inject Mocked Objects *(in place of @Autowired)*

```
@InjectMocks MyController myController;
```

Mock Method

```
when (personRepositoryMock.getPersonById(1)).thenReturn(new Person(1, "Susan", 50));
given(personRepositoryMock.getPersonById(1)).willReturn(new Person(1, "Susan", 50)); /Alias        for when()
doReturn(new Person(1, "Susan", 50)).when(personRepositoryMock).getPersonById(1);    //Alternative for when()
```

@Mock vs @Spy [R]

- The difference is that with
  - **@Mock** Real Method is **never called**
  - **@Spy** Real Method is **always called** & executed but return value is intercepted & Mocked Value is returned instead
- We could say that
  - **@Mock** creates **fully** Mocked Object (Object only has Mocked Methods defined by Mockito)
  - **@Spy** creates **partially** Mocked Object (Object has combination of Real and Mocked Methods)
- The other subtle difference is that with **@Spy** Real Methods are **always** called - calls to Real Methods are not blocked.
  - If Method was Mocked it will return what was defined by Mockito after it gets executed.
  - If Method was not Mocked it will return what was defined by its actual implementation.

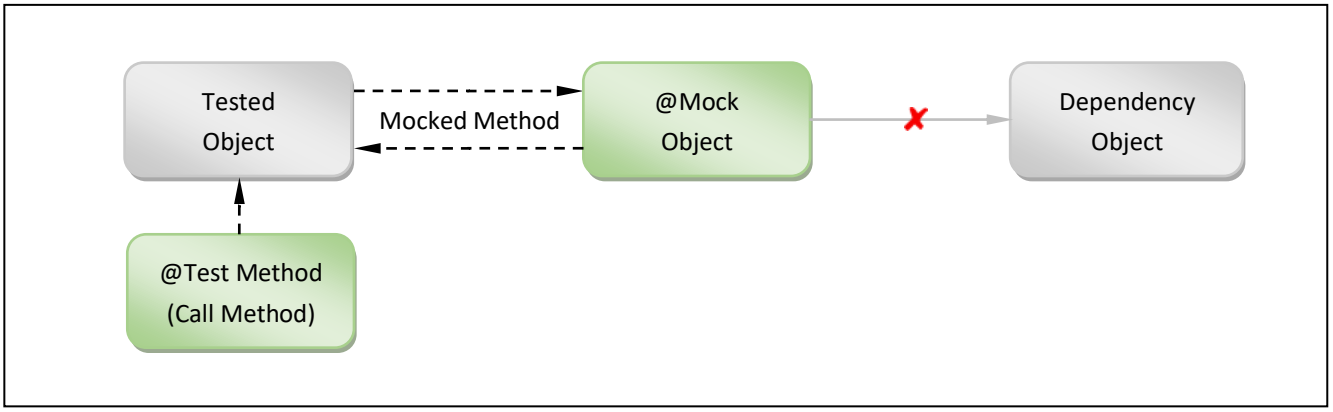
Syntax

```
//MOCK DEPENDENCY CLASS                          (choose @Mock or @Spy)
@Mock PersonRepository personRepositoryMock;
@Spy  PersonRepository personRepositorySpy;

//INJECT MOCKS                                   (into Class being tested where @autowired is used)
@InjectMocks MyController myController;

//MOCK METHOD OF DEPENDENCY CLASS                (Methods are mocked in the same way for both @Mock and @Spy)
when(personRepositoryMock.getPersonById(1)).thenReturn(new Person(1, "Susan", 50));
when(personRepositorySpy .getPersonById(1)).thenReturn(new Person(1, "Susan", 50));
```

@Mock

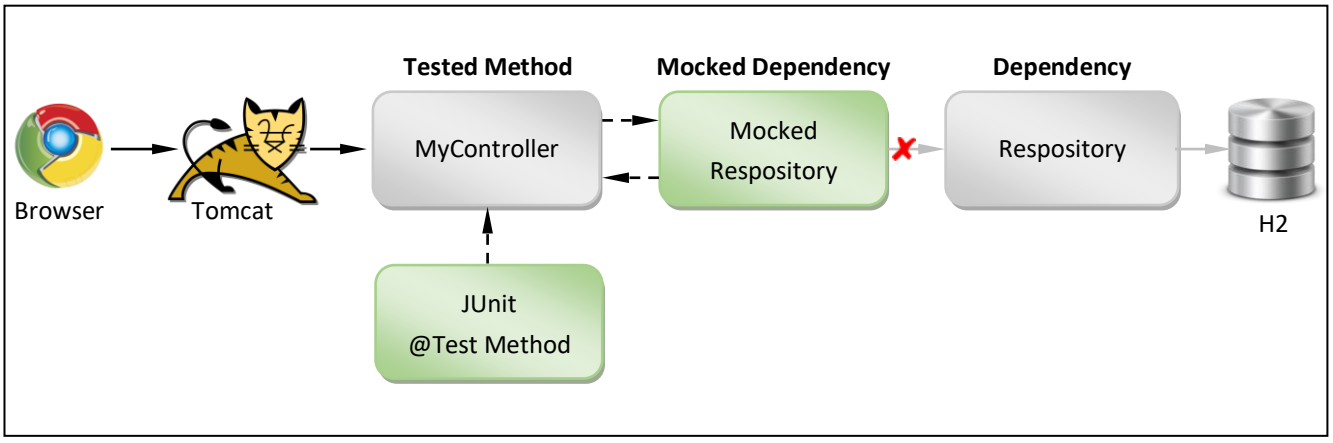
- @Mock creates fake Object without any Methods except those that you introduce through when() and doReturn().
- In @Mock Object default behavior of the Method is to do nothing.
- @Mock is used for **Functional Testing** - to test functionality of a component in isolation from other components.
- This means that during tests Mocked Methods are called instead of real ones.
  Calls to Mocked Methods are intercepted by Mockito which then returns whatever we specify.
  This way we can test how tested Method would behave when Dependent Methods return certain values.
- Such approach is useful during parallel development of interdepended components.
  Each developer can test his component by Mocking dependent components that are still under development.

@Mock



- For instance if we are testing Endpoint that calls Repository, then we would Mock Repository to return specific value.
- That way we can test Endpoint's functionality without having access to the actual Repository or the underlying Database.
- And we can also test how Endpoint will behave upon receiving different values from the Repository without having to adjust Database to actually return these values.

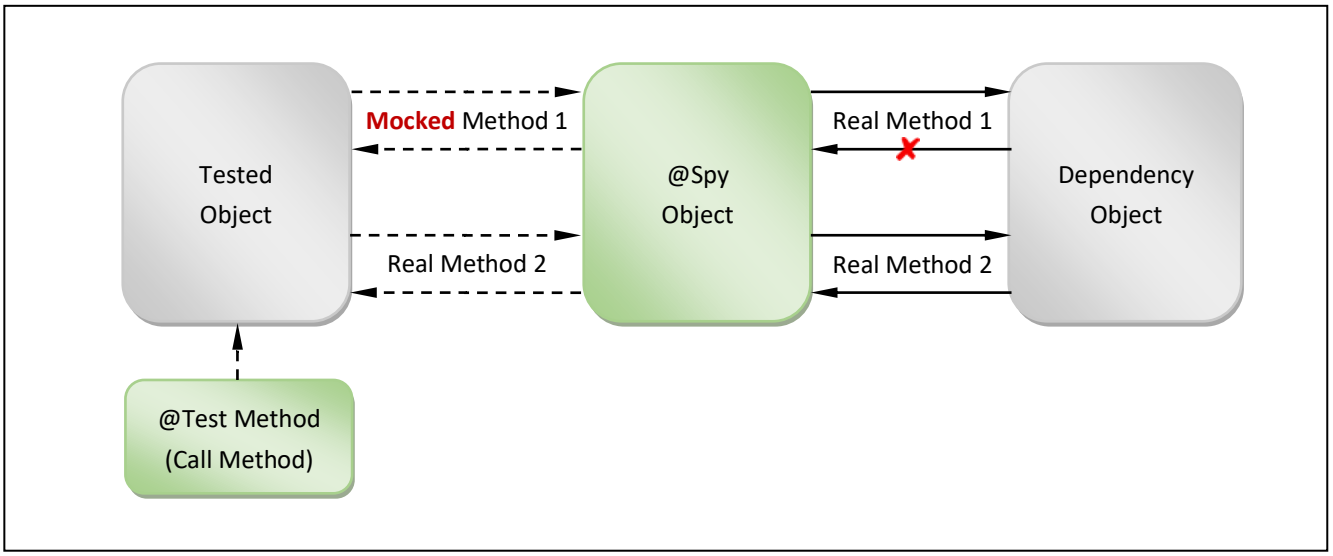
Example



@Spy

- @Spy creates Wrapper Object around real/existing Object that has its own working Methods.
  Then you use when() or doReturn() to change return value just for some of those Methods.
- In @Spy Object default behavior of the Method is the one implemented by the Application.
- @Spy is used for **Integration Testing** - to test how your component interacts/integrates with other components.
  So with @Spy you can test one component at the time by Mocking calls to those other components not being tested.

@Spy

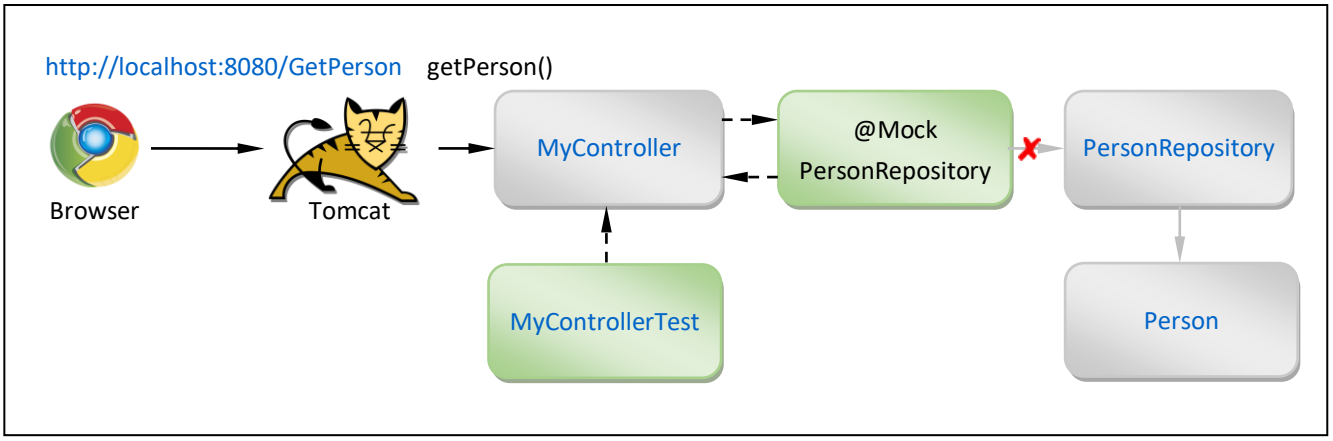


4.2.1 Mock - Object - @Mock

Info [G] [R]

- This tutorial shows how to use
  - **@Mock** to **Mock Class** `@Mock PersonRepository personRepository`
  - **@InjectMocks** to **Inject Instance of Mocked Class** `@InjectMocks MyController myController`
  - **when()** to **Mock Method of Mocked Class** `when(personRepository.getPersonById(1)).thenReturn(...);`
- `@InjectMocks` looks for `@Autowired PersonRepository personRepository` & inserts instance of Mocked Class in Controller
  When Controller calls `personRepository.getPersonById(1)` then Mocked Method of Mocked Class is called.

Application Schema *[Result]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Syntax

```
@SpringBootTest
class MyControllerTest {
  @Mock        PersonRepository personRepositoryMock;
  @InjectMocks MyController     myController;
```

Procedure

- Create Project: springboot_mockito_mock (add Spring Boot Starters from the table)
- Create Package: entities (inside main package)
  - Create Class: Person.java (inside entities package)
- Create Package: respositories (inside main package)
  - Create Class: PersonRepository.java (inside entities package)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Package: controllers (inside test package src**test**\java\com.ivoronline.springboot_junit)
  - Create Test Class: MyControllerTest.java (inside entities package)

Person.java

```
package com.ivoronline.springboot_mockito_mock.entities;

public class Person {

  //PROPERTIES
  public Integer id;
  public String  name;
  public Integer age;

  //CONSTRUCTOR
  public Person(Integer id, String name, Integer age) {
    this.id   = id;
    this.name = name;
    this.age  = age;
  }

}
```

PersonRepository.java

```
package com.ivoronline.springboot_mockito_mock.repositories;

import com.ivoronline.springboot_mockito_mock.entities.Person;
import org.springframework.stereotype.Component;
import java.util.HashMap;
import java.util.Map;

@Component
public class PersonRepository {

  //PROPERTY
  Map<Integer, Person> persons = new HashMap();

  //CONSTRUCTOR
  public PersonRepository() {
    persons.put(1, new Person(1, "John", 20));
    persons.put(2, new Person(2, "Bill", 30));
    persons.put(3, new Person(2, "Jack", 40));
  }

  //GET PERSON BY ID
  public Person getPersonById(Integer id) {
    return persons.get(id);
  }

}
```

MyController.java

```
package com.ivoronline.springboot_mockito_mock.controllers;

import com.ivoronline.springboot_mockito_mock.entities.Person;
import com.ivoronline.springboot_mockito_mock.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @Autowired PersonRepository personRepository;

  @ResponseBody
  @RequestMapping("/GetPerson")
  public String getPerson(@RequestParam Integer id) {

    //GET PERSON
    Person  person = personRepository.getPersonById(id);
    String  name   = person.name;
    Integer age    = person.age;

    //RETURN SOMETHING
    return name + " is " + age + " years old";

  }

}
```

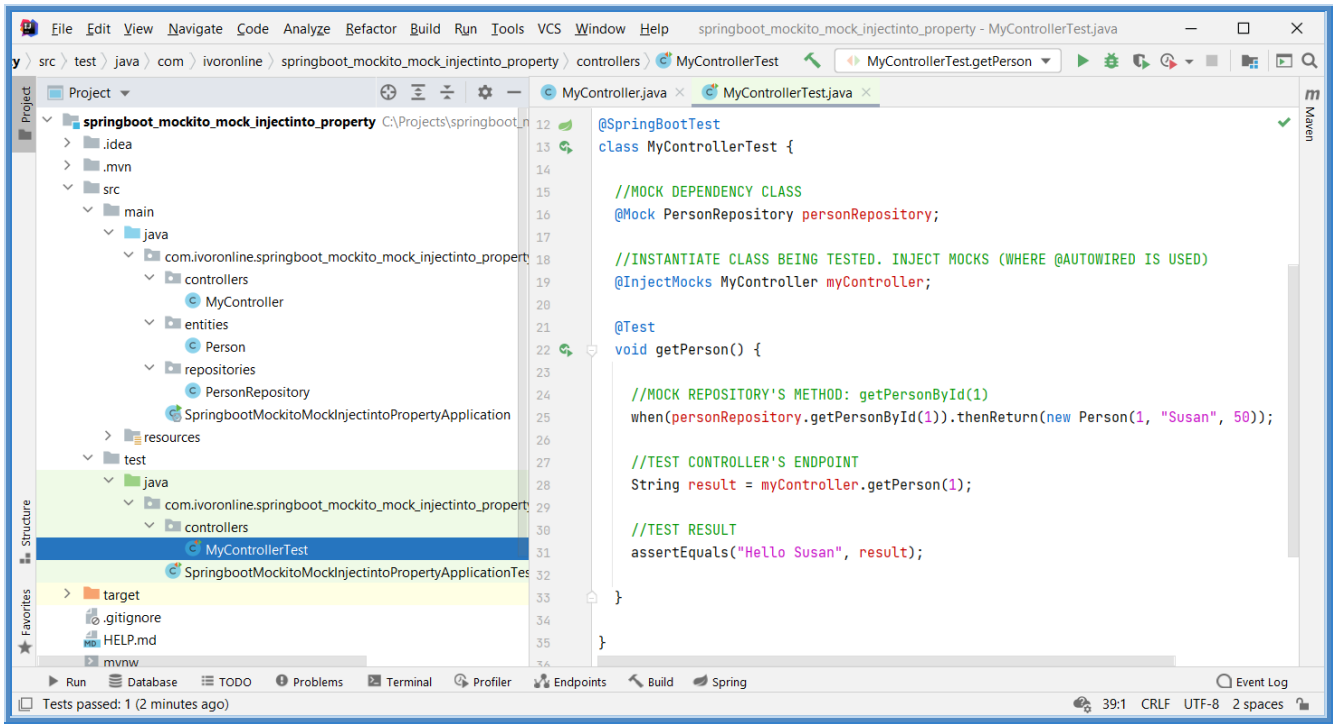
MyControllerTest.java

```
package com.ivoronline.springboot_mockito_mock.controllers;

import com.ivoronline.springboot_mockito_mock.entities.Person;
import com.ivoronline.springboot_mockito_mock.repositories.PersonRepository;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.boot.test.context.SpringBootTest;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.when;

@SpringBootTest
class MyControllerTest {

  //MOCK DEPENDENCY CLASS
  @Mock PersonRepository personRepositoryMock;

  //INJECT MOCKS (where @autowired is used)
  @InjectMocks MyController myController;

  @Test
  void getPerson() {

    //MOCK REPOSITORY METHOD (getPersonById(1) returns Susan instead John for id=1)
    when(personRepositoryMock.getPersonById(1)).thenReturn(new Person(1, "Susan", 50));

    //TEST CONTROLLER ENDPOINT
    String result = myController.getPerson(1);

    //TEST RESULT
    assertEquals("Susan is 50 years old", result);

  }

}
```

Result

http://localhost:8080/GetPerson?id=1

Run Test Class: MyControllerTest.java



pom.xml

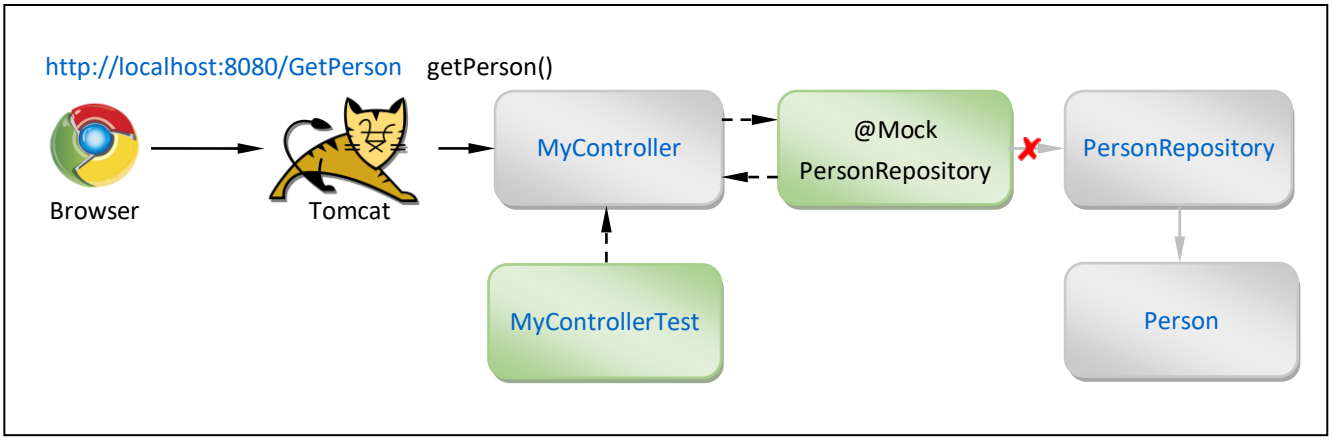
```
<dependencies>

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

</dependencies>
```

4.2.2 Mock - Object - @MockBean

Info [G] [R]

- This tutorial shows how to use
  - **@Mock** to **Mock Class** `@MockBean PersonRepository personRepository`
  - **@InjectMocks** to **Inject Instance of Mocked Class** `@Autowired MyController myController`
  - **when()** to **Mock Method of Mocked Class** `when(personRepository.getPersonById(1)).thenReturn(...);`
- `@InjectMocks` looks for `@Autowired PersonRepository personRepository` & inserts instance of Mocked Class in Controller
  When Controller calls `personRepository.getPersonById(1)` then Mocked Method of Mocked Class is called.

Application Schema *[Result]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Syntax

```
@SpringBootTest
class MyControllerTest {
  @MockBean  PersonRepository personRepositoryMock;
  @Autowired MyController     myController;
```

Procedure

- Create Project: springboot_mockito_mockbean (add Spring Boot Starters from the table)
- Create Package: entities (inside main package)
  - Create Class: Person.java (inside entities package)
- Create Package: respositories (inside main package)
  - Create Class: PersonRepository.java (inside entities package)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Package: controllers (inside test package)
  - Create Test Class: MyControllerTest.java (inside controllers package)

Person.java

```
package com.ivoronline.springboot_mockito_mockbean.entities;

public class Person {

  //PROPERTIES
  public Integer id;
  public String  name;
  public Integer age;

  //CONSTRUCTOR
  public Person(Integer id, String name, Integer age) {
    this.id   = id;
    this.name = name;
    this.age  = age;
  }

}
```

PersonRepository.java

```
package com.ivoronline.springboot_mockito_mockbean.repositories;

import com.ivoronline.springboot_mockito_mockbean.entities.Person;
import org.springframework.stereotype.Component;
import java.util.HashMap;
import java.util.Map;

@Component
public class PersonRepository {

  //PROPERTY
  Map<Integer, Person> persons = new HashMap();

  //CONSTRUCTOR
  public PersonRepository() {
    persons.put(1, new Person(1, "John", 20));
    persons.put(2, new Person(2, "Bill", 30));
    persons.put(3, new Person(2, "Jack", 40));
  }

  //GET PERSON BY ID
  public Person getPersonById(Integer id) {
    return persons.get(id);
  }

}
```

MyController.java

```
package com.ivoronline.springboot_mockito_mockbean.controllers;

import com.ivoronline.springboot_mockito_mockbean.entities.Person;
import com.ivoronline.springboot_mockito_mockbean.repositories.PersonRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MyController {

  @Autowired PersonRepository personRepository;

  @ResponseBody
  @RequestMapping("/GetPerson")
  public String getPerson(@RequestParam Integer id) {

    //GET PERSON
    Person  person = personRepository.getPersonById(id);
    String  name   = person.name;
    Integer age    = person.age;

    //RETURN SOMETHING
    return name + " is " + age + " years old";

  }

}
```

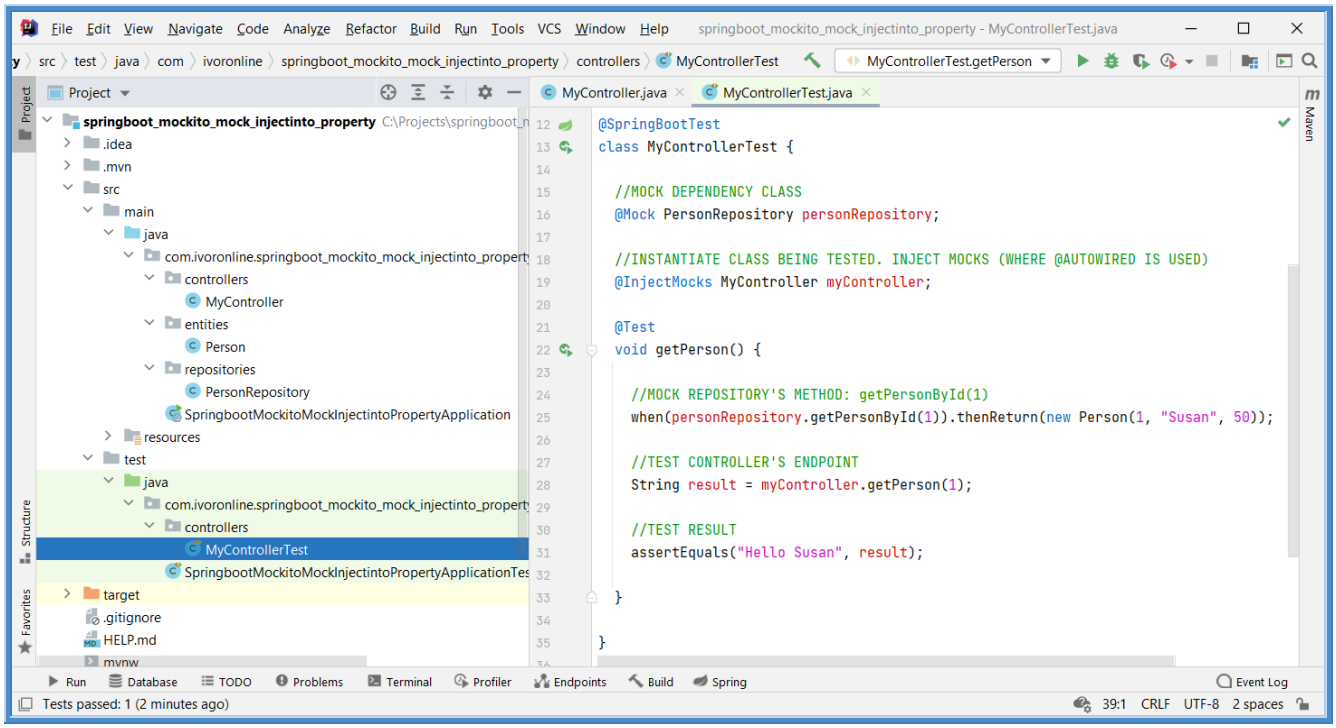
MyControllerTest.java

```
package com.ivoronline.springboot_mockito_mockbean.controllers;

import com.ivoronline.springboot_mockito_mockbean.entities.Person;
import com.ivoronline.springboot_mockito_mockbean.repositories.PersonRepository;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.mockito.Mock;
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.mockito.Mockito.when;
import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
class MyControllerTest {

  //MOCK DEPENDENCY CLASS
  @MockBean PersonRepository personRepositoryMock;

  //INJECT MOCKS (where @autowired is used)
  @Autowired MyController myController;

  @Test
  void getPerson() {

    //MOCK METHOD: getPersonById(1)
    when(personRepositoryMock.getPersonById(1)).thenReturn(new Person(1, "Susan", 50));

    //TEST CONTROLLER'S ENDPOINT
    String result = myController.getPerson(1);

    //TEST RESULT
    assertEquals("Susan is 50 years old", result);

  }

}
```

Result

http://localhost:8080/GetPerson?id=1

Run Test Class: MyControllerTest.java



pom.xml

```
<dependencies>

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

</dependencies>
```

4.3 MockMVC

Info

- Following tutorials show how to use MockMvc to test Controllers by
  - performing HTTP Requests
  - verifying HTTP Responses

Syntax

```
mockMvc.perform(get("/Hello"))          //perform HTTP Request
       .andExpect(status().isOk());     //verify  HTTP Response
```

Instantiate MockMvc

- MockMvc can be instantiated in different ways as shown below.

@SpringBootTest, @Autowired

```
@SpringBootTest
class MyControllerTest {
  @Autowired MockMvc mockMvc;
```

@WebMvcTest, @Autowired

```
@WebMvcTest
class MyControllerTest {
  @Autowired MockMvc mockMvc;
```

MockMvcBuilders

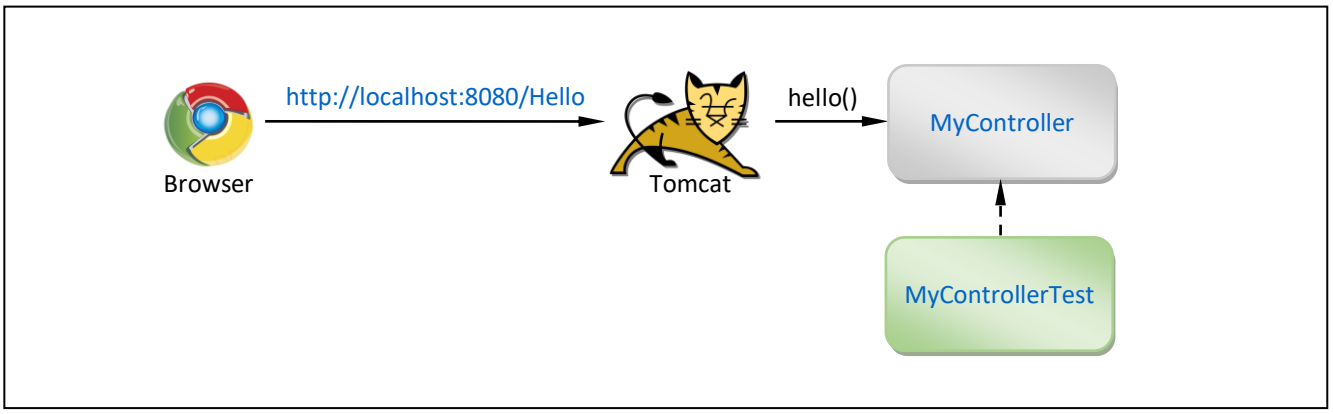
```
class MyControllerTest {
  MyController myController = new MyController();
  @Test
  void hello() throws Exception {
    MockMvc mockMvc = MockMvcBuilders.standaloneSetup(myController).build();
```

4.3.1 @WebMvcTest

Info [G] [R]

- This tutorial shows how to use `@WebMvcTest` which
  - creates Application Context that only contains Beans needed for testing web Controllers
  - instantiates `MockMvc` (as one such Bean)
- This means that `@WebMvcTest` for example will instantiate Classes that are Annotated with `@Controller`.
  This means that you **can** use `@Autowired MyController myController` to inject Instance of `MyController`.
- But `@WebMvcTest` for example will not instantiate Classes that are Annotated with `@Entity`.
  This means that you **can't** use `@Autowired PersonEntity personEntity` to inject Instance of `PersonEntity`.
- However you can use `@MockBean` or manually instantiate Classes that are not automatically pulled into Context.
  Purpose of `@WebMvcTest` is to keep tests faster than when using `@SpringBootTest` which loads everything into Context.
  But to also allow you to add additional instances if needed to fine tune your tests.

Application Schema *[Result]*



Spring Boot Starters

GROUP	DEPENDENCY	DESCRIPTION
Web	Spring Web	Enables: Controller Annotations, Tomcat Server

Syntax

```
@WebMvcTest
class MyControllerTest {
  @Autowired MockMvc      mockMvc;
  @Autowired MyController myController;
```

Procedure

- Create Project: springboot_test_mockmvc (add Spring Boot Starters from the table)
- Create Package: controllers (inside main package)
  - Create Class: MyController.java (inside controllers package)
- Create Test Class: MyControllerTest.java

MyController.java

```
package com.ivoronline.springboot_test_mockmvc_webmvctest.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MyController {

  @ResponseBody
  @RequestMapping("/Hello")
  public String hello() {
    return "Hello from Controller";
  }

}
```

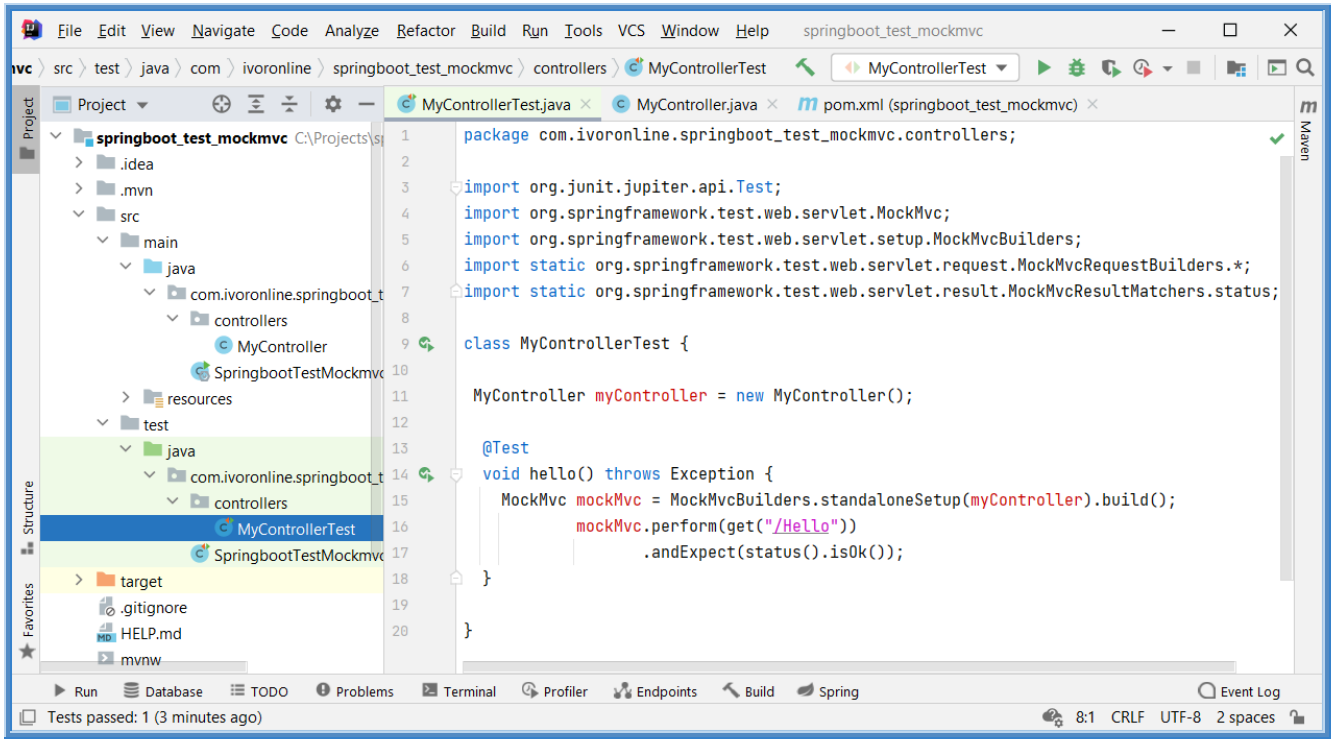
MyControllerTest.java

```
package com.ivoronline.springboot_test_mockmvc_webmvctest.controllers;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest
class MyControllerTest {

  @Autowired MockMvc      mockMvc;
  @Autowired MyController myController;

  @Test
  void hello() throws Exception {
    mockMvc.perform(get("/Hello"))
           .andExpect(status().isOk());
  }

}
```

Result

http://localhost:8080/Hello

Run Test Class: MyControllerTest.java



pom.xml

```
<dependencies>

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

</dependencies>
```