Spring Boot 2.7.8

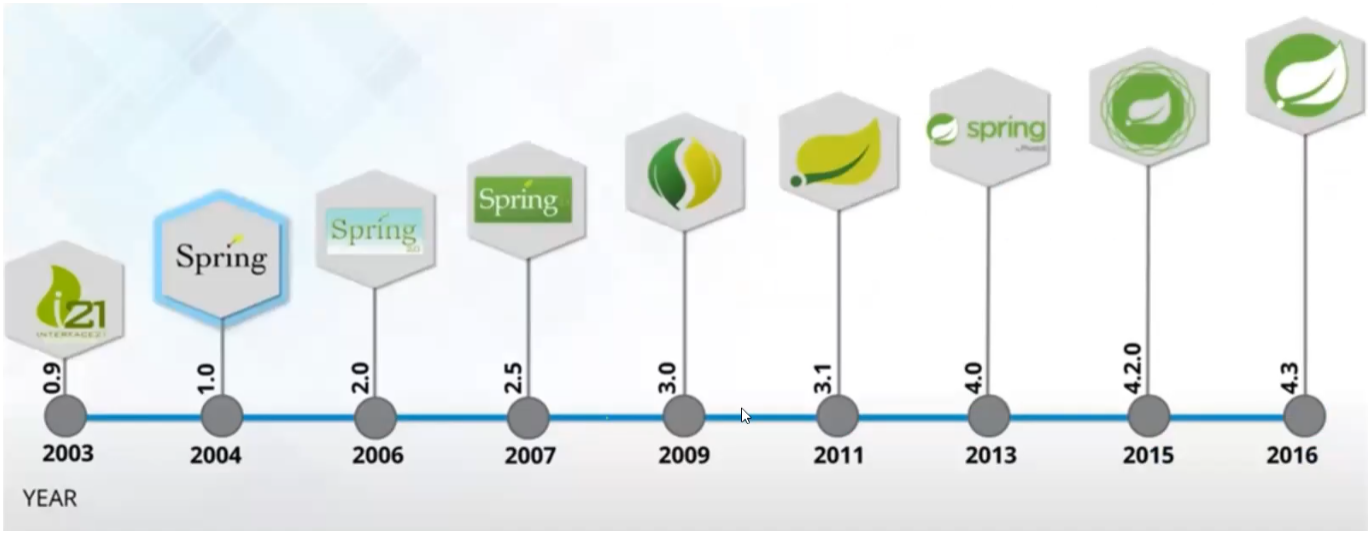
Spring History

It was developed by Rod Johnson in 2003.(https://en.wikipedia.org/wiki/Spring_Framework)

Spring Versions

- The first version was written by Rod Johnson, who released the framework with the publication of his book Expert One-on-One J2EE Design and Development in October 2002.
- The framework was first released under the Apache 2.0 license in June 2003. The first production release, 1.0, was released in March 2004.
- The Spring 1.2.6 framework won a Jolt productivity award and a JAX Innovation Award in 2006.

- Spring 2.0 was released in October 2006, Spring 2.5 in November 2007, Spring 3.0 in December 2009, Spring 3.1 in December 2011, and Spring 3.2.5 in November 2013.
- Spring Framework 4.0 was released in December 2013.
- Notable improvements in Spring 4.0 included support for Java SE (Standard Edition) 8, Groovy 2, some aspects of Java EE 7, and WebSocket.



What is Spring

- Spring is a complete and a modular framework for developing enterprise application in java
- Spring framework can be user for all layer implementation for a real time appllication
- It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc.
- The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc

What is spring Framework ?

* Java Framework is a platform of pre-written codes used by java developers to develop the application(java or web application)

- Java Framework Principle abides the hollywood principle that is **Don't call us ,we'll call you**.Also called inversion of flow or inverstion of Control

  ![Spring :java-Framework-Principle](images/Java-Framework-principle.png)

"Let's take plain old java, here Class A has a dependency upon Class B and Class C to instantiate we use a new operator and it will become tight coupling. To avoid the tight coupling we are injecting the object at run time and trying to make it loose coupling."

Why Framework

A framework will help us to reuse the code in the application field because it is well-tested code by 100-1000

developers and the community. That's why we always see if a framework is available if not we should write custom code.

Why Spring is popular ?

- Simplicity - It is simple because as it in non-invasive.it uses POJO or POJI
- Testability - For developing and testing Spring Server(container) is not mandatory
- Loose Coupling - Spring objects are loosely coupled.It will inject the object at runtime in the IOC container or application context.

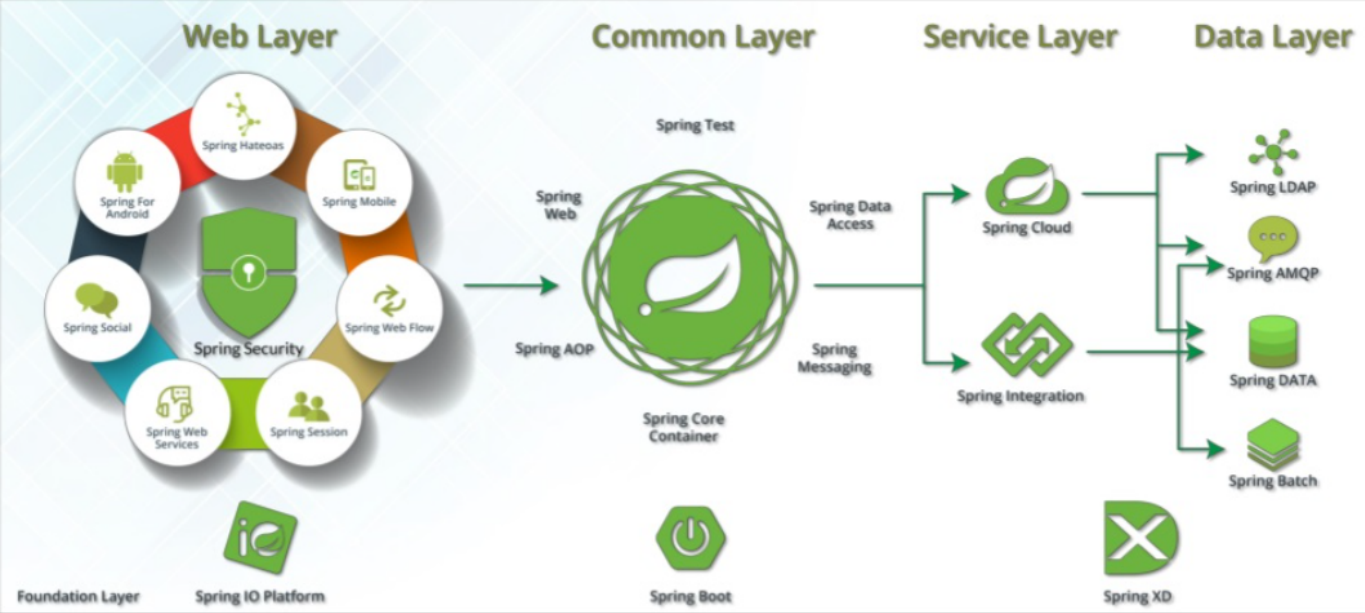
Different Java Framework

Spring Features

- Lightweight: Spring Framework is lightweight with respect to size and transparency.

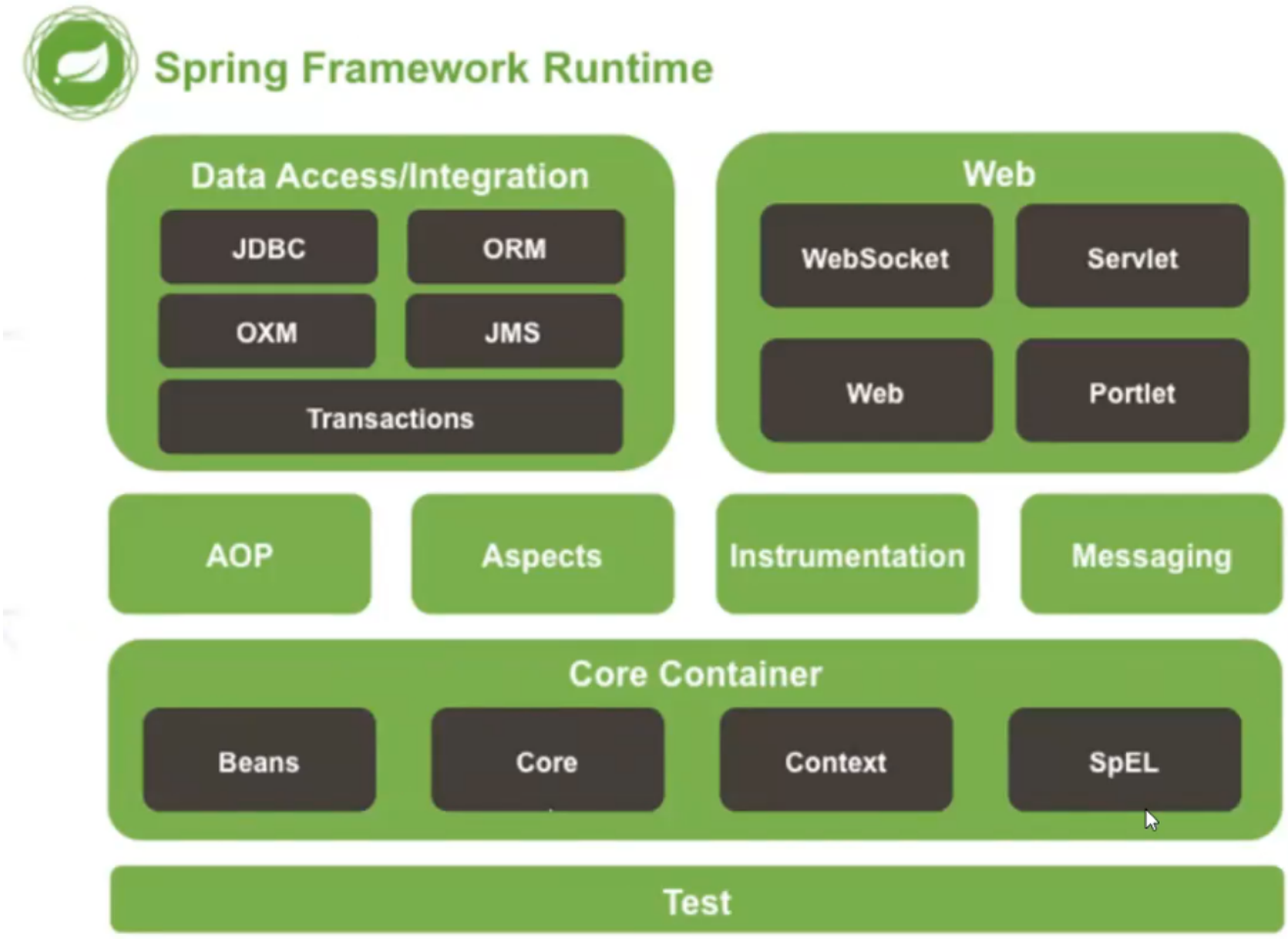
- Inversion Of Control (IoC): In Spring Framework, loose coupling is achieved using Inversion of Control. The objects give their own dependencies instead of creating or looking for dependent objects.
- Aspect Oriented Programming (AOP): By separating application business logic from system services, Spring Framework supports Aspect Oriented Programming and enables cohesive development.
- Container: Spring Framework creates and manages the life cycle and configuration of application objects.
- MVC Framework: Spring Framework is a MVC web application framework. This framework is configurable via interfaces and accommodates multiple view technologies.
- Transaction Management: For transaction management, Spring framework provides a generic abstraction layer. It is not tied to J2EE environments and it can be used in container-less environments,for refrences: (https://www.geeksforgeeks.org/spring-boot-transaction-management-using-transactional-annotation/).
- JDBC Exception Handling: The JDBC abstraction layer of the Spring Framework offers an exception hierarchy, which simplifies the error handling strategy.

Spring Framework Ecosystem



For More reference :(https://springtutorials.com/spring-ecosystem/)

Spring Architecture/Module



Spring Core Container Layer

The Spring Core container contains core, beans, context and expression language (EL) modules.

- Core and Beans: These modules provide IOC and Dependency Injection features.
- Context: This module supports internationalization EJB, JMS, Basic Remoting.
- Expression Language: It is an extension to the EL defined in JSP. It provides support to setting and getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrieval of objects by name etc.

AOP, Aspects and Instrumentation Layer:

These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.

The aspects module provides support to integration with AspectJ.

The instrumentation module provides support to class instrumentation and classloader implementations.

for refrences: (https://docs.spring.io/spring-framework/docs/2.5.5/reference/aop.html)

Data Access / Integration Layer:

This group comprises of JDBC, ORM, OXM, JMS and Transaction modules. These modules basically provide support to interact with the database.

- The JDBC module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service JMS module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web Layer

This group comprises of Web, Web-Servlet, Web-Struts and Web-Portlet. These modules provide support to create web application.

- The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The Web-MVC module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The Web-Socket module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Model View Controller

![Spring :MVC](images/mvc.png)

MVC Explanation

Model :- It contains the data of the application

View :- view repesents the particular format (HTML,JSTL etc)

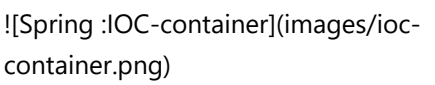
Controller :- it cantains the business logic

Understanding the flow of the MVC :-

The controller gets the request from the user, and based on the request controller decides whether it should send a view or get some information from the model. If the information is needed from the model, it will send the request to the model and get the data from the model and send the data to the user through the view page.

IOC Container

- The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided.
- The configuration metadata can be represented either by XML, Java annotations, or Java code.
- The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.
- configured by loading the xml files or by detecting specific java annotations on configuration classes.



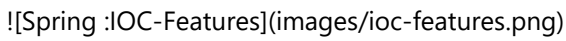
Types of IOC Container

1. **BeanFactory Container** This is the simplest container providing the basic support for DI and is defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

2. **ApplicationContext Container** This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContext interface.

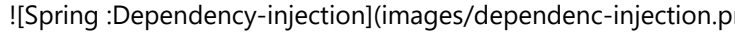
IOC Features

- It creates the objects
- configures and assembles their dependencies
- manages their entire life cycle



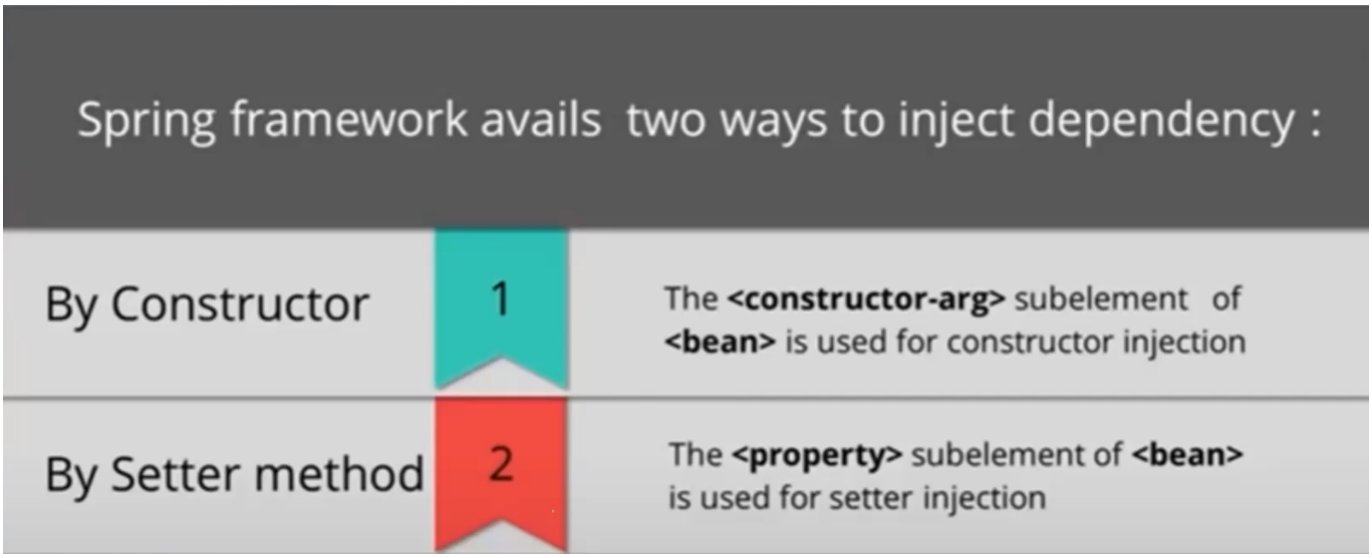
Dependency Injection

Dependency injection is a fundamental aspect of the Spring framework, through which the Spring container **injects** objects into other objects or **dependencies**. Simply put, this allows for loose coupling of components and moves responsibility for managing components onto the container.



Types Of Dependency Injection

1. [Constructor Based Dependency Injection](#constructor-based-dependency-injection) 2. [Setter Based Dependency Injection](#setter-based-dependency-injection)



Constructor Based Dependency Injection

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

We can inject the dependency by constructor. The subelement of is used for constructor injection. Here we are going to inject

- primitive and String-based values
- Dependent object (contained object)
- Collection values etc.

Setter Based Dependency Injection

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

We can inject the dependency by setter method also. The subelement of is used for setter injection. Here we are going to inject

- primitive and String-based values
- Dependent object (contained object)
- Collection values etc.

Bean

- Beans are the objects that from the backbone of the Application and are managed by the Spring Iox container.

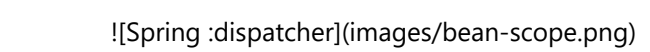
- Spring Ioc Container instantiates,assembles and manages The bean Object
- The Configuration metadata that are supplied to the container are used to create the beans object.
- **Bean Dependency Injection**:

1. It’s a design pattern which removes the dependency from the Programming code ,that makes the application easy to manage and test.
2. Dependency injection makes our Programming code loosely coupled, which means change in implementation does not affect the user or invoking application

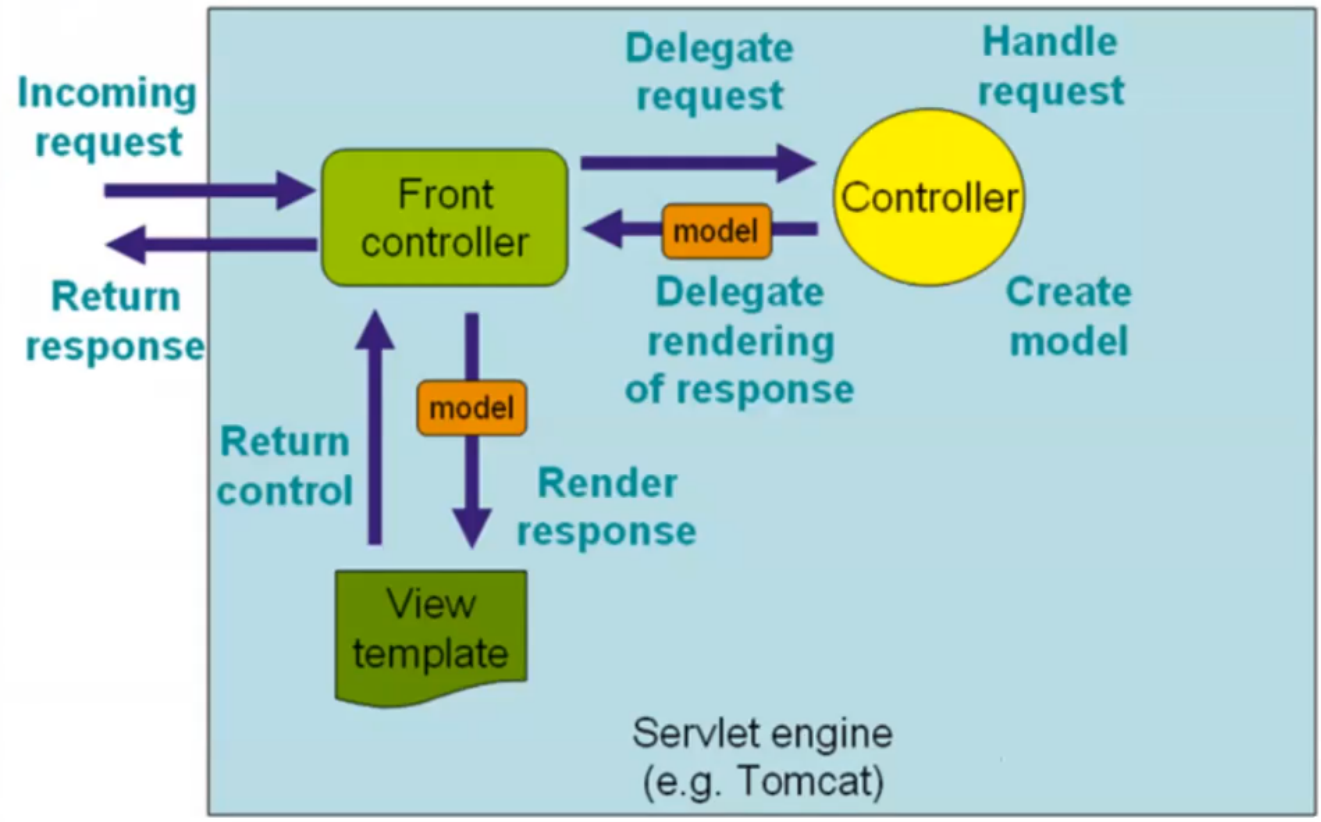
bean life cycle

- The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.
- To define setup and teardown for a bean, we simply declare the with initmethod and/or destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroymethod specifies a method that is called just before a bean is removed from the container.

Bean Scope



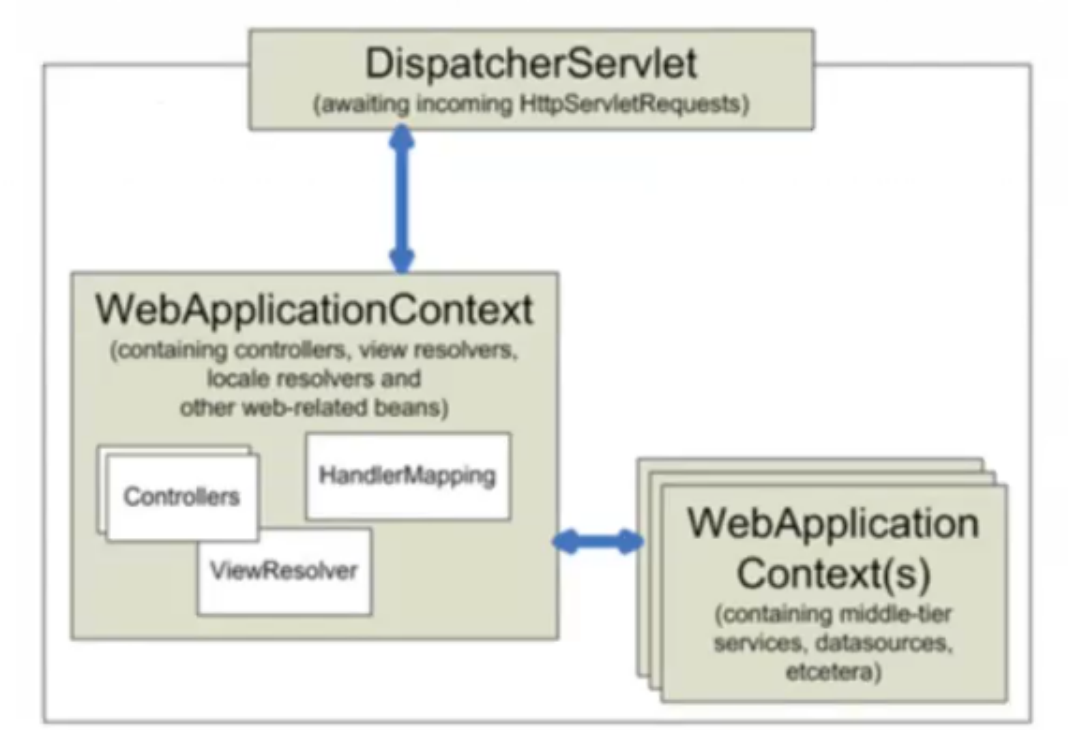
Dispatcher Servlet



Dispatcher Servlet Working

Servlet : is an specification as part of jee ascepts . servlet are used to creating the dispatcher servlets.

- In Spring MVC all incoming requests go through a single servlet is called Dispatcher Servlet (front controller). The front controller is a design pattern in web application development. A single servlet receives all the request and transfers them to all other components of the application.
- The job of DispatcherServlet is to take an incoming URI and find the right combination of handlers (Controller classes) and views (usually JSPs). When the DispatcherServlet determines the view, it renders it as the response. Finally, the DispatcherServlet returns the Response Object to back to the client.



Spring Conditional Configuration

1. [ConditionOnBean](#conditiononbean)
    2. ConditionOnProperty
    3. ConditionOnMissingBean
    4. ConditionOnClass

ConditionOnBean

@Conditional that only matches when beans meeting all the specified requirements are already contained in the BeanFactory. All the requirements must be met for the condition to match, but they do not have to be met by the same bean.

```
factory method:

 @Configuration

 public class MyAutoConfiguration {

     @ConditionalOnBean
     @Bean
     public MyService myService() {

         ...
```

```
    }
}
```

In the sample above the condition will match if a bean of type MyService is already contained in the BeanFactory.

ConditionOnProperty

@Conditional that checks if the specified properties have a specific value. By default the properties must be present in the Environment and not equal to false. The havingValue() and matchIfMissing() attributes allow further customizations. The havingValue attribute can be used to specify the value that the property should have. The table below shows when a condition matches according to the property value and the havingValue() attribute:

Property value	having value= ""	having value="true"	having value ="false	havingvalue ="foo"
"True"	yes	yes	no	no
"false"	no	no	yes	no
"foo"	yes	no	no	yes

If the property is not contained in the Environment at all, the matchIfMissing() attribute is consulted. By default missing attributes do not match. This condition cannot be reliably used for matching collection properties. For example, in the following configuration, the condition matches if spring.example.values is present in the Environment but does not match if spring.example.values[0] is present.

```
@ConditionalOnProperty(prefix = "spring", name = "example.values")
class ExampleAutoConfiguration {
}
```

ConditionOnMissingBean

@Conditional that only matches when no beans meeting the specified requirements are already contained in the BeanFactory. None of the requirements must be met for the condition to match and the requirements do not have to be met by the same bean.

```
@Configuration
public class MyAutoConfiguration {

    @ConditionalOnMissingBean
    @Bean
    public MyService myService() {
        ...
    }

}
```

In the sample above the condition will match if no bean of type MyService is already contained in the BeanFactory. The condition can only match the bean definitions that have been processed by the application context so far and, as such, it is strongly recommended to use this condition on auto-configuration classes only. If a candidate bean may be created by another auto-configuration, make sure that the one using this condition runs after.

ConditionOnClass

@Conditional that only matches when the specified classes are on the classpath. A Class value can be safely specified on @Configuration classes as the annotation metadata is parsed by using ASM before the class is loaded. This only holds true if @ConditionalOnClass is used on a class. Extra care must be taken when using @ConditionalOnClass on @Bean methods: the value attribute must not be used, instead the name attribute can be used to reference the class which must be present as a String. Alternatively create a separate @Configuration class that isolates the condition.

For example:

```
@AutoConfiguration
public class MyAutoConfiguration {

   @Configuration(proxyBeanMethods = false)
   @ConditionalOnClass(SomeService.class)
   public static class SomeServiceConfiguration {

       @Bean
       @ConditionalOnMissingBean
       public SomeService someService() {
           return new SomeService();
       }
   }

}
```

Bootstrapping Spring boot project

What is spring boot ?

- Spring Boot makes it easy to create stand-alone, production-grade Spring based applications that you can "just run".
- we take an opinionated view of the spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration
- Spring Boot enables developers to focus on the business logic behind their microservice.It aims to take care of all the nitty-gritty technical details involved in developing microservices.

![Spring :defination-of-spring](images/defination-of-spring.png)

Spring Vs Spring Boot

Spring Initializr

Spring Initializr provides a lot of flexibility in creating projects. You have options to do the following:

1. Choose your build tool: Maven or Gradle. – Choose the Spring Boot version you want to use.
2. Configure a Group ID and Artifact ID for your component.
3. Choose the starters (dependencies) that you would want for your project.
4. Choose how to package your component: JAR or WAR.
5. Choose the Java version you want to use.
6. Choose the JVM language you want to use.

There are 3 ways to create Spring Boot project

- Spring Initializr web (https://start.spring.io/)
    1. installation Steps (https://www.youtube.com/watch?v=qhZtYAw0C9s)
- Spring Tool Suite (STS)
- Spring Boot CLI

spring-boot-starters

Starters are simplified dependency descriptors customized for different purposes.

- For example, spring-boot-starter-web is the starter for building web application, including RESTful, using Spring MVC. It uses Tomcat as the default embedded container.
- If I want to develop a web application using Spring MVC, all we would need to do is include spring-boot-starter-web in our dependencies, and we get the following automatically pre-configured:

1. Spring MVC
2. Compatible versions of jackson-databind (for binding) and hibernate-validator (for form validation)
3. spring-boot-starter-tomcat (starter project for Tomcat)

Spring-boot-starter-parent

* The spring-boot-starter-parent dependency is the parent POM providing dependency and plugin management for Spring Boot-based applications.

- A spring-boot-starter-parent dependency contains the default versions of Java to use, the default versions of dependencies that Spring Boot uses, and the default configuration of the Maven plugins

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.8</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>
```

Spring-boot-starter-web

This will add Spring MVC capabilities to Spring Boot

- Helps you build Spring MVC-based web applications or RESTful applications. It uses Tomcat as the default embedded servlet container.

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

Spring-boot-maven-plugin

when we build applications using Spring Boot, there are a couple of situations that are possible:

- We would want to run the applications in place without building a JAR or a WAR
- We would want to build a JAR and a WAR for later deployment
- The spring-boot-maven-plugin dependency provides capabilities for both of the preceding situations. The following snippet shows how we can configure spring-boot-maven-plugin in an application:

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

Spring boot first basic project

We will start with building our first Spring Boot application.

- We will use Maven to manage dependencies.
- The following steps are involved in starting up with a Spring Boot application:

1. Configure spring-boot-starter-parent in your pom.xml file.
2. Configure the pom.xml file with the required starter projects.
3. Configure spring-boot-maven-plugin to be able to run the application.
4. Create your first Spring Boot launch class

Folder structure

![Spring :folder-of-springbootapplication](images/folderfirstspringapplicationjava.png)

Spring Application

The SpringApplication class can be used to Bootstrap and launch a Spring application from a Java main method.

- The following are the steps that are typically performed when a Spring Boot application is bootstrapped:

1. Create an instance of Spring's ApplicationContext.
2. Enable the functionality to accept command-line arguments and expose them as Spring properties.
3. Load all the Spring beans as per the configuration

```
@SpringBootApplication
public class FirstspringbootApplication  {

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

    }

}
```

@SpringBootApplication

The @SpringBootApplication annotation is a shortcut for three annotations:

1. @Configuration: Indicates that this a Spring application context configuration file.
2. @EnableAutoConfiguration: Enables auto-configuration, an important feature of Spring Boot. We will discuss auto-configuration later in a separate section.
3. @ComponentScan: Enables scanning for Spring beans in the package of this class and all its sub packages.

![Spring :springbootapplication](images/Springbootapplication.png)

beans loading

Let's see what all beans are loaded at this point

- Where are these beans defined? & How are these beans created?
- That's the magic of Spring auto-configuration.
- Whenever we add a new dependency to a Spring Boot project, Spring Boot auto-configuration automatically tries to configure the beans based on the dependency

Code for bean Loading

```
@SpringBootApplication
public class FirstspringbootApplication  {


    public static void main(String[] args) {

        ConfigurableApplicationContext applicationContext
=SpringApplication.run(FirstspringbootApplication.class, args);
        SpringDemoComponent component =
applicationContext.getBean(SpringDemoComponent.class);
        System.out.println(component.getMessage());

        String[] beanNames =applicationContext.getBeanDefinitionNames();
        Arrays.sort(beanNames);//optional
        for(String beanName: beanNames) {
        System.out.println(beanName);
        }
        }

}
```

Spring-boot-application-working

![Spring :spring-boot-application-working](images/Spring-boot-application-working.png)

Anotations

The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context.

- To use this annotation, apply it over class as below:
- This should print in console, which indicates this component was instantiated, we can reprint beans and validate.

```
@Component
public class SpringDemoComponent {
```

```
    public SpringDemoComponent() {
        System.out.println("*********Constructor or SpringDemoComponent use
called****");
    }

}
```

@Component Vs @bean

@Component and @Bean do two quite different things, and shouldn't be confused.

- @Component (and @Service and @Repository) are used to auto-detect and auto-configure beans using classpath scanning. There's an implicit one-to-one mapping between the annotated class and the bean (i.e. one bean per class). Control of wiring is quite limited with this approach, since it's purely declarative.
- @Bean is used to explicitly declare a single bean, rather than letting Spring do it automatically as above. It decouples the declaration of the bean from the class definition, and lets you create and configure beans exactly how you choose.
- Component Preferable for component scanning and automatic wiring.

When @Bean is used

* When should you use @Bean?

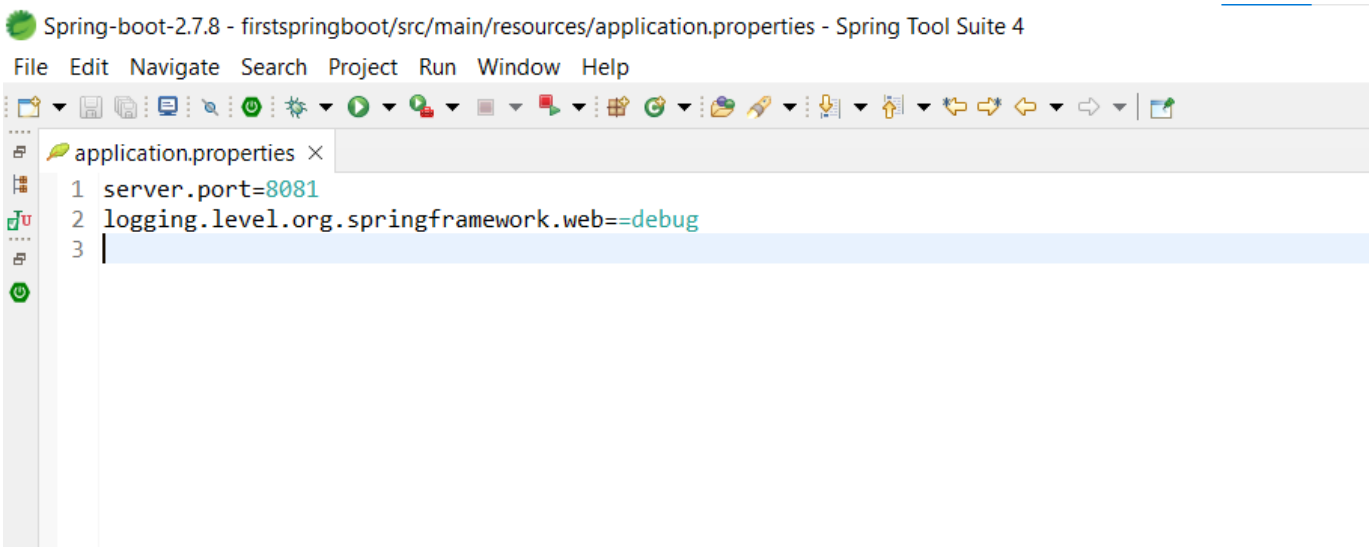
1. Sometimes automatic configuration is not an option. When? Let's imagine that you want to wire components from 3rd-party libraries (you don't have the source code so you can't annotate its classes with @Component), so automatic configuration is not possible.
2. The @Bean annotation returns an object that spring should register as bean in application context. The body of the method bears the logic responsible for creating the instance

```
@Configuration
public class SpringConfig {

    @Bean
    @Conditional(MyCondition.class)

    public SpringDemoComponent createBean() {
        return new SpringDemoComponent();
    }
}
```

inject from application.properties

* Add a property and let’s inject

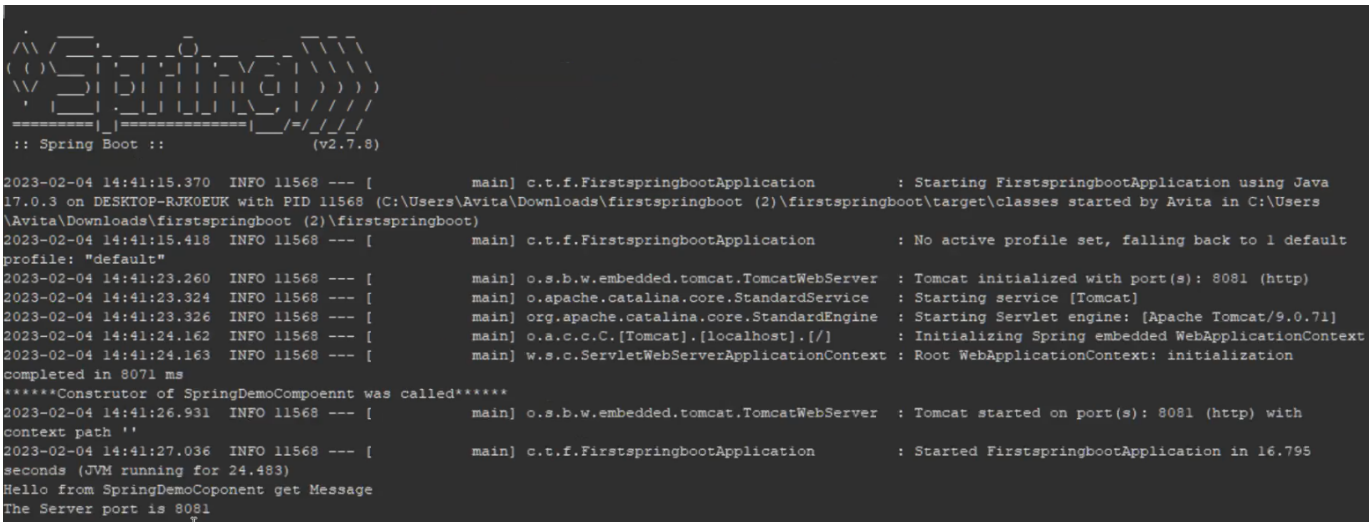
- server port = 8081 from Spring Boot in application.properties



Code of server port

```
@SpringBootApplication
public class FirstspringbootApplication implements CommandLineRunner {

    @Value("$(server.port)")
    private String message;

    public static void main(String[] args) {

        SpringApplication.run(FirstspringbootApplication.class, args);

    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("the server port is " + message);
    }

}
```

Output of server port



```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.8)

2023-02-04 14:41:15.370  INFO 11568 --- [           main] c.t.f.FirstspringbootApplication         : Starting FirstspringbootApplication using Java
17.0.3 on DESKTOP-RJK0EUK with PID 11568 (C:\Users\Avita\Downloads\firstspringboot (2)\firstspringboot\target\classes started by Avita in C:\Users
\Avita\Downloads\firstspringboot (2)\firstspringboot)
2023-02-04 14:41:15.418  INFO 11568 --- [           main] c.t.f.FirstspringbootApplication         : No active profile set, falling back to 1 default
profile: "default"
2023-02-04 14:41:23.260  INFO 11568 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8081 (http)
2023-02-04 14:41:23.324  INFO 11568 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-02-04 14:41:23.326  INFO 11568 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.71]
2023-02-04 14:41:24.162  INFO 11568 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-02-04 14:41:24.163  INFO 11568 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 8071 ms
******Construtor of SpringDemoCompoennt was called******
2023-02-04 14:41:26.931  INFO 11568 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8081 (http) with
context path ''
2023-02-04 14:41:27.036  INFO 11568 --- [           main] c.t.f.FirstspringbootApplication         : Started FirstspringbootApplication in 16.795
seconds (JVM running for 24.483)
Hello from SpringDemoCoponent get Message
The Server port is 8081
```

Creating Web Project (Rest API)

1. [Starters in spring Boot](#starters-in-spring-boot)
    2. Autowired
    3. Autowired by Web-starter
    4. Creating a Spring firstspringproject
    5. CommandLineRunner

Starters in Spring Boot

Starters are simplified dependency descriptors customized for different purposes.

- For example, spring-boot-starter-web is the starter for building web application, including RESTful, using Spring MVC. It uses Tomcat as the default embedded container.
- If I want to develop a web application using Spring MVC, all we would need to do is include spring-boot-starter-web in our dependencies, and we get the following automatically pre-configured:

1. Spring MVC
2. Compatible versions of jackson-databind (for binding) and hibernate-validator (for form validation)
3. spring-boot-starter-tomcat (starter project for Tomcat)

What are the other starters

(https://www.geeksforgeeks.org/spring-boot-starters/)

Autowired

- The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished.

- The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

Autowired by Web-starter

When we add a dependency in spring-boot-starter-web, the following beans are auto-configured:

- basicErrorController, handlerExceptionResolver: It is the basic exception handling. It shows a default error page when an exception occurs.
- beanNameHandlerMapping: It is used to resolve paths to a handler (controller).
- characterEncodingFilter: It provides default character encoding UTF-8. – dispatcherServlet: It is the front controller in Spring MVC applications.

Autowired by Web-starter

- jacksonObjectMapper: It translates objects to JSON and JSON to objects in REST services.
- messageConverters: It is the default message converters to convert from objects into XML or JSON and vice versa.
- multipartResolver: It provides support to upload files in web applications.
- mvcValidator: It supports validation of HTTP requests. – viewResolver: It resolves a logical view name to a physical view.
- propertySourcesPlaceholderConfigurer: It supports the externalization of application configuration.
- requestContextFilter: It defaults the filter for requests.
- restTemplateBuilder: It is used to make calls to REST services.
- tomcatEmbeddedServletContainerFactory: Tomcat is the default embedded servlet container for Spring Boot-based web applications

Creating a Spring firstspringproject

Tomcat server is launched on port 8080 - Tomcat started on port(s): 8080 (http).

- DispatcherServlet is configured. This means that Spring MVC Framework is ready to accept requests--Mapping servlet: 'dispatcherServlet' to [/].
- Four filters are enabled by default

1. characterEncodingFilter
2. hiddenHttpMethodFilter,
3. httpPutFormContentFilter
4. requestContextFilter

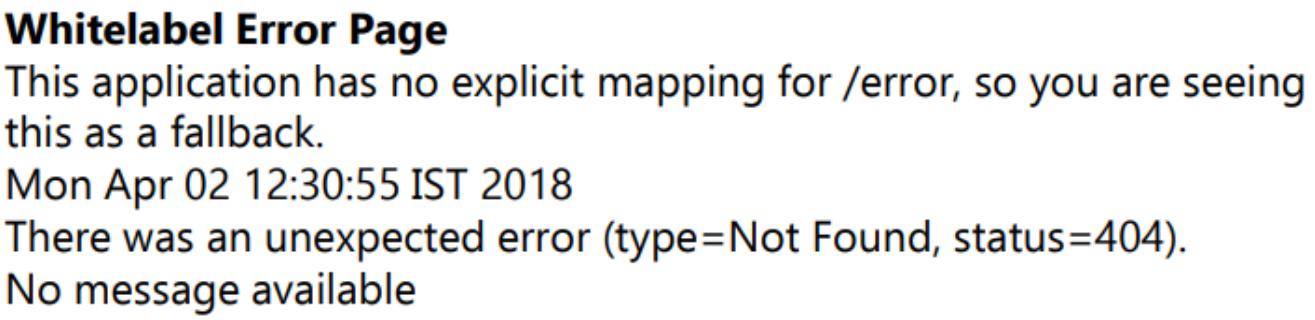
* The default error page is configured

1. Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>>
2. org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)

- WebJars are autoconfigured. WebJars enable dependency management for static dependencies such as Bootstrap and query--Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

output

* If we now open a browser and go to http://localhost:8081/ you will notice the default white label error page.



CommandLineRunner

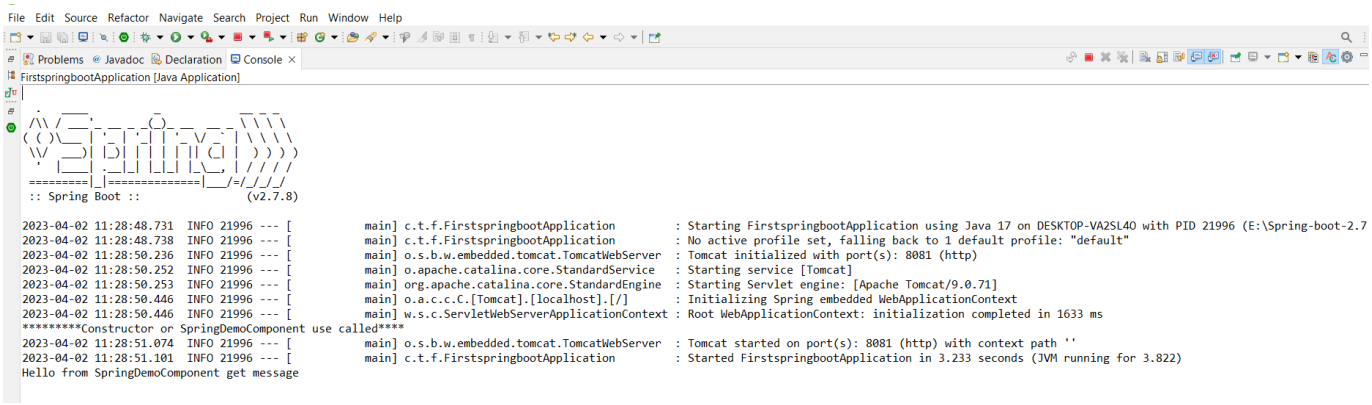
Command line runner to show Dependency injection

Interface used to indicate that a bean should run when it is contained within a SpringApplication.

Code Example:

```
@SpringBootApplication
public class FirstspringbootApplication implements CommandLineRunner {

    @Autowired
    SpringDemoComponent springDemoComponent;


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

    }

    @Override
    public void run(String[] args) throws Exception {
        System.out.println(springDemoComponent.getMessage());
    }

}
```

Output of commandline runner



Rest

Representational State Transfer (REST) is basically an architectural style for the web. REST specifies a set of constraints. These constraints ensure that clients (service consumers and browsers) can interact with servers in flexible ways.

- **Terminology**:

1. Server: Service provider. Exposes services which can be consumed by clients.
2. Client: Service consumer. Could be a browser or another system.
3. Resource: Any information can be a resource: a person, an image, a video, or a product you want to sell.
4. Representation: A specific way a resource can be represented. For example, the product resource can be represented using JSON, XML, or HTML. Different clients might request different representations of the resource.

Rest Constraints

* Client-Server: There should be a server (service provider) and a client (service consumer). This enables loose coupling and independent evolution of the server and client as new technologies emerge.

- Stateless: Each service should be stateless. Subsequent requests should not depend on some data from a previous request being temporarily stored. Messages should be self-descriptive.
- Uniform interface: Each resource has a resource identifier. In the case of web services, we use this URI example: /users/Jack/Todos/1. In this, URI Jack is the name of the user. 1 is the ID of the Todo we would want to retrieve.
- Cacheable: The service response should be cacheable. Each response should indicate whether it is cacheable.
- Layered system: The consumer of the service should not assume a direct connection to the service provider. Since requests can be cached, the client might be getting the cached response from a middle layer.
- Manipulation of resources through representations: A resource can have multiple representations. It should be possible to modify the resource through a message with any of these representations.
- Hypermedia as the engine of application state (HATEOAS): The consumer of a RESTful application should know about only one fixed service URL. All subsequent resources should be discoverable from the links included in

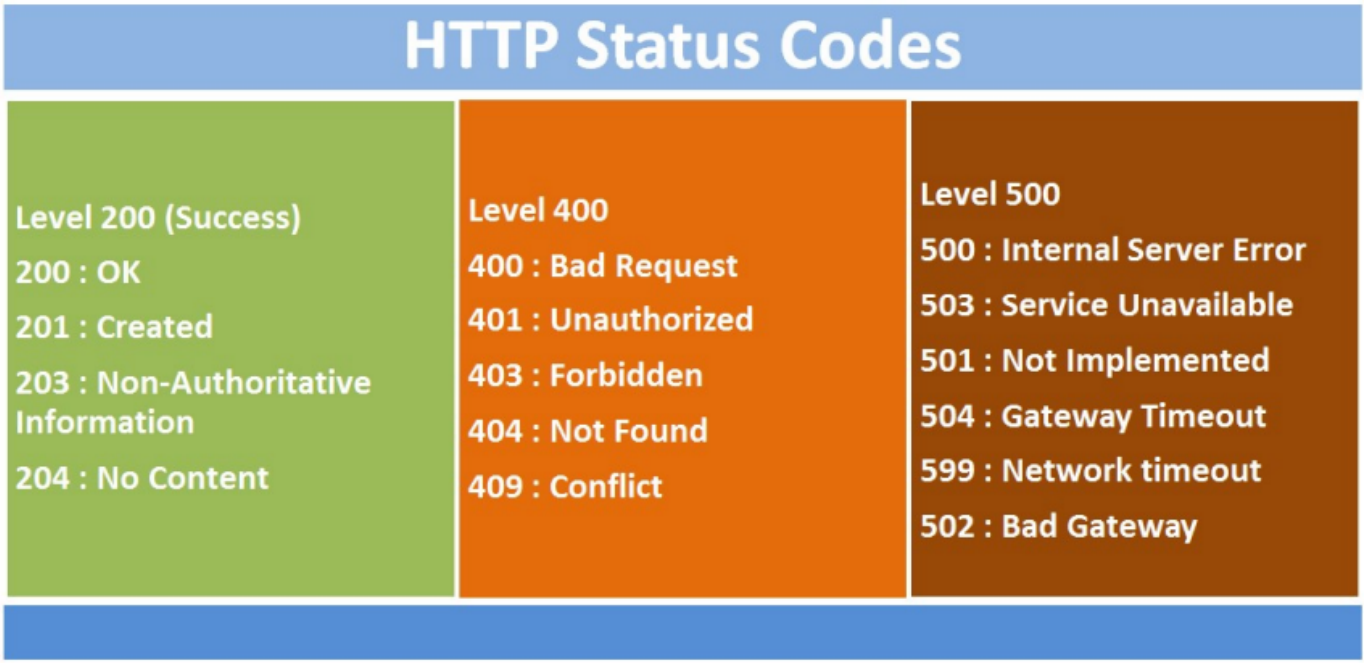
the resource representations.

HTTP Status Codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

- informational responses
- successful responses
- redirects
- client errors
- servers errors.
- Go to: (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)

Response of HTTP Status codes



HTTP-Rest-Vocabulary

HTTP Methods supported by REST:

- GET – Requests a resource at the request URL

```
- Should not contain a request body, as it will be discarded.

- May be cached locally or on the server.

- May produce a resource, but should not modify on it.
```

- POST – Submits information to the service for processing

```
- Should typically return the new or modified resource.
```

- PUT – Add a new resource at the request URL
- DELETE – Removes the resource at the request URL
- OPTIONS – Indicates which methods are supported
- HEAD – Returns meta information about the request URL

Spring Boot Rest Project

* Let's start with creating a simple REST service returning a welcome message

- creating a simple REST Controller method returning a string:

```
@RestController
public class HelloWorldController {


    @GetMapping("/hello")
    public String sayHello() {
        return "Hello web";

    }
    )
```

Explanation

* RestController: The @RestController annotation provides a combination of @ResponseBody and @Controller annotations. This is typically used to create REST Controllers.

- @GetMapping("welcome"): @GetMapping is a shortcut for @RequestMapping(method = RequestMethod.GET). This annotation is a readable alternative. The method with this annotation would handle a Get request to the welcome URI.
- Due to the latest changes in Spring boot 2.1 it's needs the following line to be added to application.properties to see mappings of URL

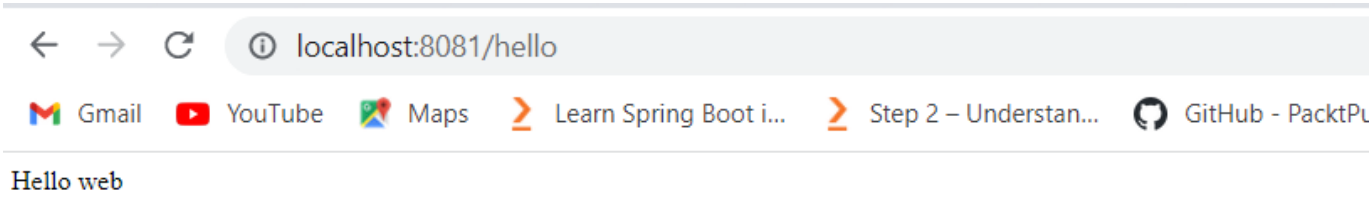
```
logging.level.org.springframework.web=trace
```

- When you execute this Spring application you will notice :

```
2019-01-02 05:18:56.637 TRACE 8089 --- [ main]
s.w.s.m.m.a.RequestMappingHandlerMapping :
c.c.f.c.HelloController:
{GET /hello}: sayHello()
```

- Now if you open a browser and browse to http://localhost:8081/hello

Output



Spring Boot Rest Project with object

Let's add a new mapping and return a HelloWeb Object

```
@RestController
public class HelloWorldController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello web";

    }

    @GetMapping("/helloweb")
    public HelloWeb sayHelloWithObject() {
        HelloWeb helloWeb = new HelloWeb();
        helloWeb.setId(1);
        helloWeb.setMessage("Hello web!");
        return helloWeb;

    }
}
```

or

```
@RestController
public class HelloWorldController {
    @Autowired
    HelloWeb helloWeb;

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello web";

    }

    @GetMapping("/helloweb")
    public HelloWeb sayHelloWithObject() {
        helloWeb.setId(1);
        helloWeb.setMessage("Hello web!");
        return helloWeb;

    }
}
```

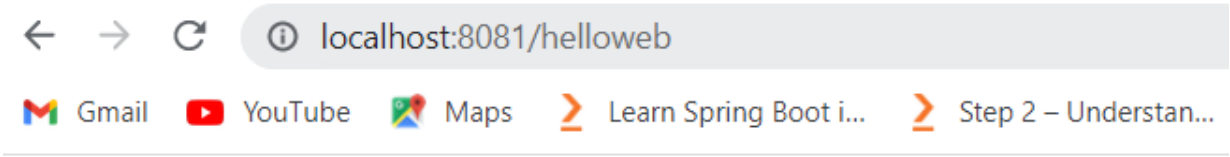
HelloWeb class

```
public class HelloWeb {

    private Integer id;

    private String message ;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Output

Open a browser and go to : http://localhost:8080/helloweb

```
Content-Type →application/json;charset=UTF-8
Date →Mon, 02 Apr 2018 07:43:38 GMT
Transfer-Encoding →chunked
```




```
{
    id: 1,
    message: "Hello web!"
}
```


- As observered with out any intervention Spring Web starter is allowing us to create JSON responses from Java Objects
- Again, it's the magic of Spring Boot auto-configuration. If Jackson is on the classpath of an application, instances of the default object to JSON (and vice versa) converters are auto-configured by Spring Boot.

Todo Application

We will focus on creating REST services for a basic Todo management system. We will create services for the following:

– Retrieving a list of Todos for a given user

– Retrieving details for a specific Todo

– Creating a Todo for a user

- To keep things simple, this service does not talk to the database. It maintains an in-memory array list of Todos. This list is initialized using a static initializer.
- We are exposing a couple of simple retrieve methods and a method to add a to-do.

Generic interaction semantics for REST resources

HTTP specifies methods or actions for the resources. The most commonly used HTTP methods or actions are POST, GET, PUT, and DELETE. This clearly simplifies the REST API design and makes it more readable.

- In a RESTful system, we can easily map our CRUD actions on the resources to the appropriate HTTP methods such as POST, GET, PUT, and DELETE

Data action	HTTP equivalent
`CREATE`	`POST` or `PUT`
`READ`	`GET`
`UPDATE`	`PUT` or `PATCH`
`DELETE`	`DELETE`

Todos REST Mapping

Let's quickly map the services that we want to create to the appropriate request methods:

- **Retrieving a list of Todos for a given user**: This is READ. We will use GET. We will use a URI: /users/{name}/Todos. One more good practice is to use plurals for static things in the URI: users, Todo, and so on. This results in more readable URIs.
- **Retrieving details for a specific Todo**: Again, we will use GET. We will use a URI /users/{name}/Todos/{id}. You can see that this is consistent with the earlier URI that we decided for the list of Todos.
- **Creating a Todo for a user**: For the create operation, the suggested HTTP Request method is POST. To create a new Todo, we will post to URI /users/{name}/Todos

Todo Coding

Todo Bean

We have a created a simple Todo bean with the ID, the description of the Todo, the Todo target date, and an indicator for the completion status. We added a constructor and getters for all fields.

```
public class Todo {

    private int id;

    private String description;
```

```
    private Date targetDate;

    private boolean done;

    public Todo() {
        super();
    }

    public Todo(int id, String description, Date targetDate, boolean done) {
        super();
        this.id = id;
        this.description = description;
        this.targetDate = targetDate;
        this.done = done;
    }
    )
```

Todo Service for retriving the list

We have a created a simple Todo bean with the ID, the description of the Todo, the Todo target date, and an indicator for the completion status. We added a constructor and getters for all fields.

```
@Service
public class TodoServiceInMemoryImpl implements TodosService {
    private static List<Todo> todos =new ArrayList<>();
    private static int todosCount =3;
    static {
        todos.add(new Todo(1,"Learn Spring Boot", new Date(),false));
        todos.add(new Todo(2,"Learn Spring Boot", new Date(),false));
        todos.add(new Todo(3,"Learn Spring Boot", new Date(),false));

    }

public List<Todo> getAllTodos() {
    // TODO Auto-generated method stub
    return todos;
    }

}
```

@Service

The @Service annotation is also a specialization of the component annotation. It doesn't currently provide any additional behavior over the @Component annotation, but it's a good idea to use @Service over @Component in service-layer classes because it specifies intent better. Additionally, tool support and additional behavior might rely on it in the future.

Retriving the Todo List

We will create a new RestController annotation called TodoController.

```
@RestController
@RequestMapping("/api/v1/todos")
public class TodosController {


    @Autowired
    private TodosService todoService;

    @GetMapping()
    public List<Todo> getAllTodos(){
        List<Todo> allTodos = todoService.getAllTodos();
        return allTodos;
    }
}
```

- We are autowiring the Todo service using the @Autowired annotation
- We use the @GetMapping annotation to map the Get request for the URI to the retrieveTodos method

Output


```
[
    {
        "id": 1,
        "description": "Learn Spring Boot",
        "targetDate": "2023-04-02T09:44:30.775+00:00",
        "done": false
    },
    {
        "id": 2,
        "description": "Learn Spring Boot",
        "targetDate": "2023-04-02T09:44:30.775+00:00",
        "done": false
    },
    {
        "id": 3,
        "description": "Learn Spring Boot",
        "targetDate": "2023-04-02T09:44:30.775+00:00",
        "done": false
    }
]
```


Todo Service for retriving the list by Id

```
@Override
public Todo getTodoById(int id) {
```

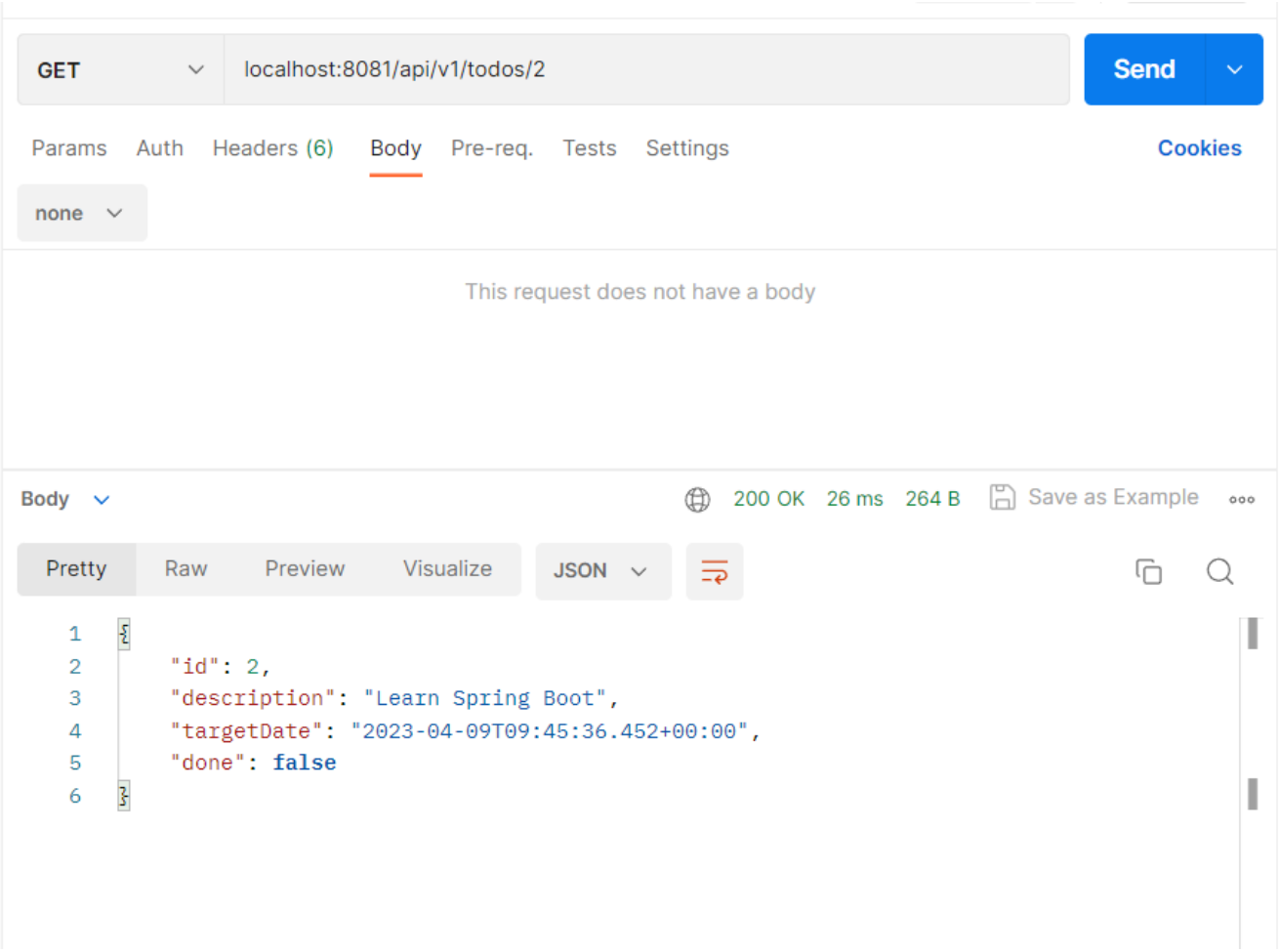
```
        System.out.println("Loading from Static ArrayList");
        for(Todo todo: todos) {
            if(todo.getId()==id) {
                return todo;
            }
        }
        throw new ResourceNotFoundException();
    }
```

GetMapping (getTodoById)

- It is basically used to read the data
- Annotation Interface GetMapping. Annotation for mapping HTTP GET requests onto specific handler methods.
- Specifically, @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. GET) .

```
@GetMapping("/{id}")
    public Todo getTodoById(@PathVariable int id) {
        return todoService.getTodoById(id);
    }
```

output



Service for saving the todo list

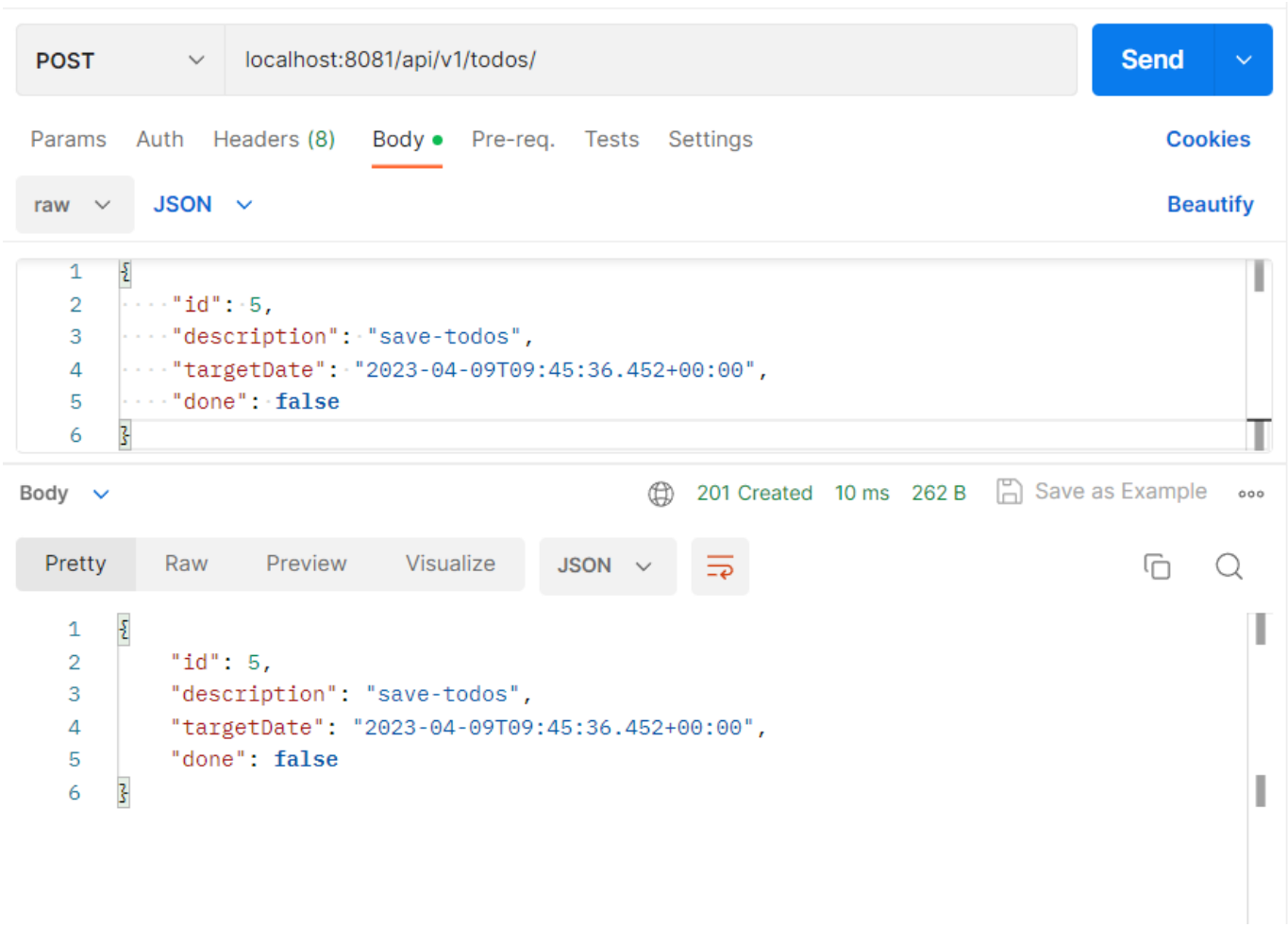
```
@Override
public Todo saveTodo(Todo todo) {
    todo.setId(++todosCount);
    todos.add(todo);
    return todo;
    // TODO Auto-generated method stub

}
```

PostMapping (saveTodo)

- It is basically used to store the data.
- Annotation for mapping HTTP POST requests onto specific handler methods.
- Specifically, @PostMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. POST) .

```
@PostMapping()
    public ResponseEntity<Todo> saveTodo(@RequestBody Todo todo){
        Todo newTodo=todoService.saveTodo(todo);
        return new ResponseEntity<Todo>(newTodo,HttpStatus.CREATED);
    }
```

output



Service for updating the todo by Id

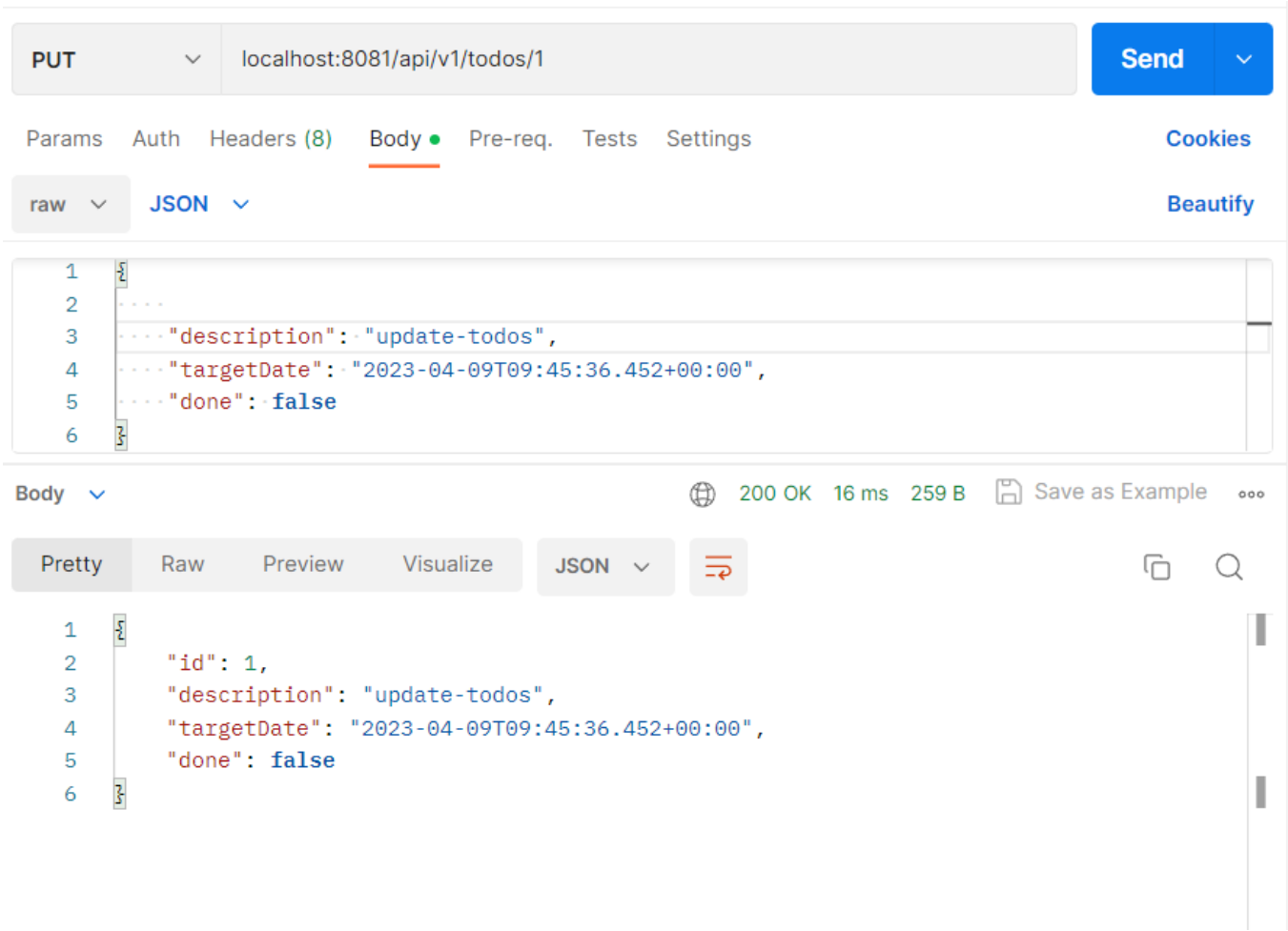
```
@Override
public Todo updateTodo(int id, Todo todo) {
    Todo existingTodo =this.getTodoById(id);
    if(existingTodo!=null) {
        existingTodo.setDescription(todo.getDescription());
        existingTodo.setTargetDate(todo.getTargetDate());
        existingTodo.setDone(todo.isDone());
    }
    return existingTodo;

}
```

PutMapping (UpdateTodo)

- It is basically used for updating the data
- Specifically, @PutMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. PUT).

```
@PutMapping("/{id}")
    public ResponseEntity<Todo> updateTodo(@PathVariable int id,@RequestBody Todo todo){
        Todo updatedTodo =todoService.updateTodo(id, todo);
        return new ResponseEntity<Todo>(updatedTodo,HttpStatus.OK);

    }
```

output



Service for deleting the todo by Id

```
@Override
public boolean deleteTodo(int id) {
```

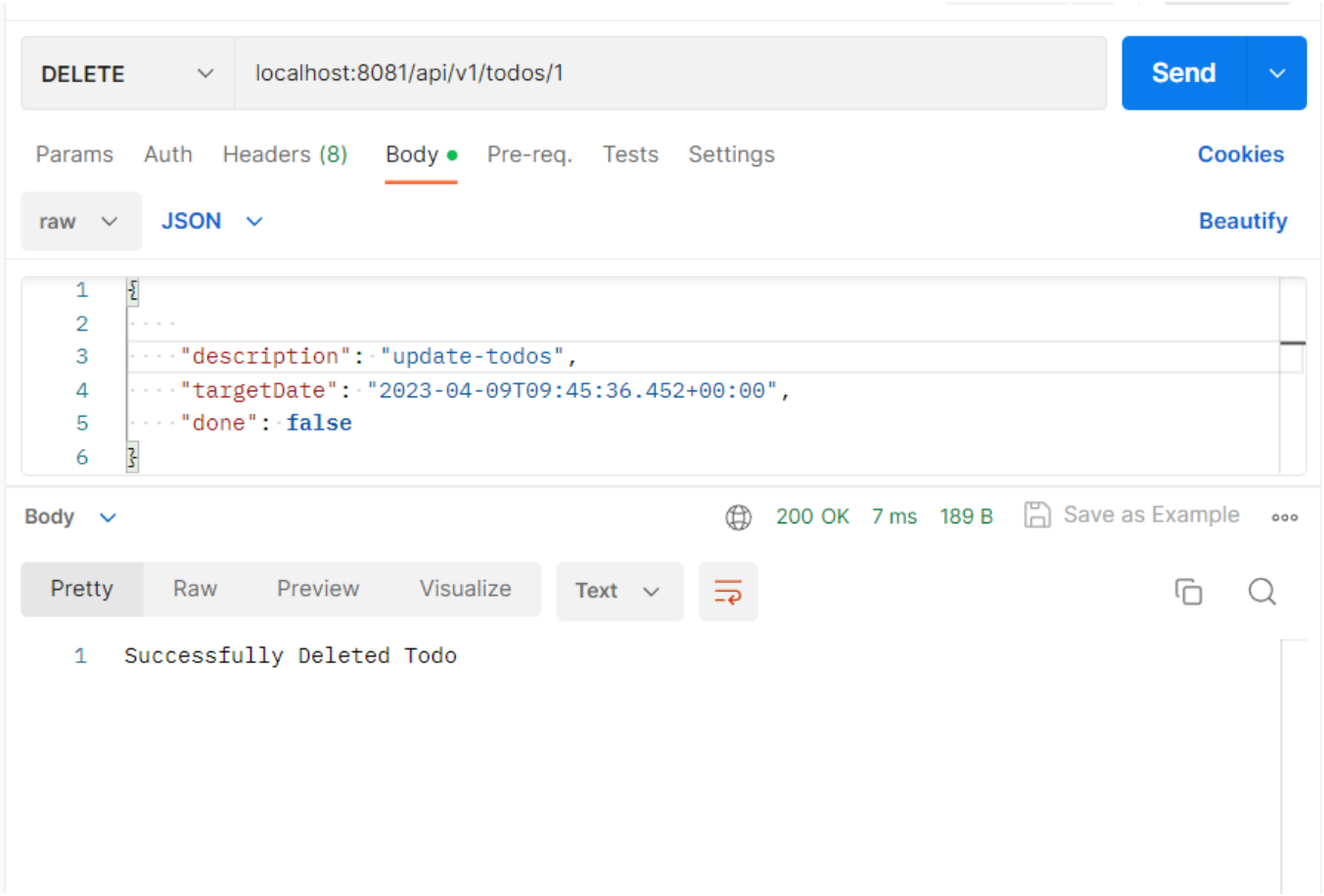
```
    for(int i=0;i< todos.size();i++) {
        if(todos.get(i).getId()==id) {
            todos.remove(i);
            return true;
        }
    }
    return false;
}
```

DeleteMapping(deleteTodoById)

- The DELETE HTTP method is used to delete the resource
- @DeleteMapping annotation for mapping HTTP DELETE requests onto specific handler methods.
- Specifically, @DeleteMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.DELETE)

```
@DeleteMapping("/{id}")
    public ResponseEntity<String> deleteTodo(@PathVariable int id){
    boolean result =todoService.deleteTodo(id);
    if(!result) {
        throw new ResourceNotFoundException();
    }
    return new ResponseEntity<String>("Successfully Deleted Todo",HttpStatus.OK);
    }
```

Output



Developer Tools

- Spring Boot provides tools that can improve the experience of developing Spring Boot applications.
- Spring Boot developer tools, by default, disables the caching of view templates and static files. This enables a developer to see the changes as soon as they make them.
- Another important feature is the automatic restart when any file in the classpath changes. So, the application automatically restarts in the following scenarios:

1. When we make a change to a controller or a service class
2. When we make a change to the property file

Dev Tool dependency and advantage

The advantages of Spring Boot developer tools are as follows:

- The developer does not need to stop and start the application each time. The application is automatically restarted as soon as there is a change.
- The restart feature in Spring Boot developer tools is intelligent. It only reloads the actively developed classes. It does not reload the third-party JARs (using two different class-loaders).
- Thereby, the restart when something in the application changes is much faster compared to cold-starting an application.

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

Exception Handling

Exception handling is one of the important parts of developing web services.

When something goes wrong, we would want to return a good description of what went wrong to the service consumer.

- Spring Boot provides good default exception handling. We will start with looking at the default exception handling features provided by Spring Boot before moving on to customizing them.

It's a very simple piece of code that defines ResourceNotFoundException.

- Now let's enhance our TodoController class to throw ResourceNotFoundException when a Todo with a given ID is not found:

```
@ResponseStatus(value =HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException{


    private static final long serialVersionUID = -1708853263336663516L;
}
```

Throwing a Custom Exception

- It's a very simple piece of code that defines ResourceNotFoundException.
- Now let's enhance our TodoController class to throw ResourceNotFoundException when a Todo with a given ID is not found:

```
@ResponseStatus(value =HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException{


    private static final long serialVersionUID = -1708853263336663516L;
}
```

```
@PostMapping()
    public ResponseEntity<Todo> saveTodo(@RequestBody Todo todo){
        Todo newTodo=todoService.saveTodo(todo);
        return new ResponseEntity<Todo>(newTodo,HttpStatus.CREATED);
    }
```

Unit Testing

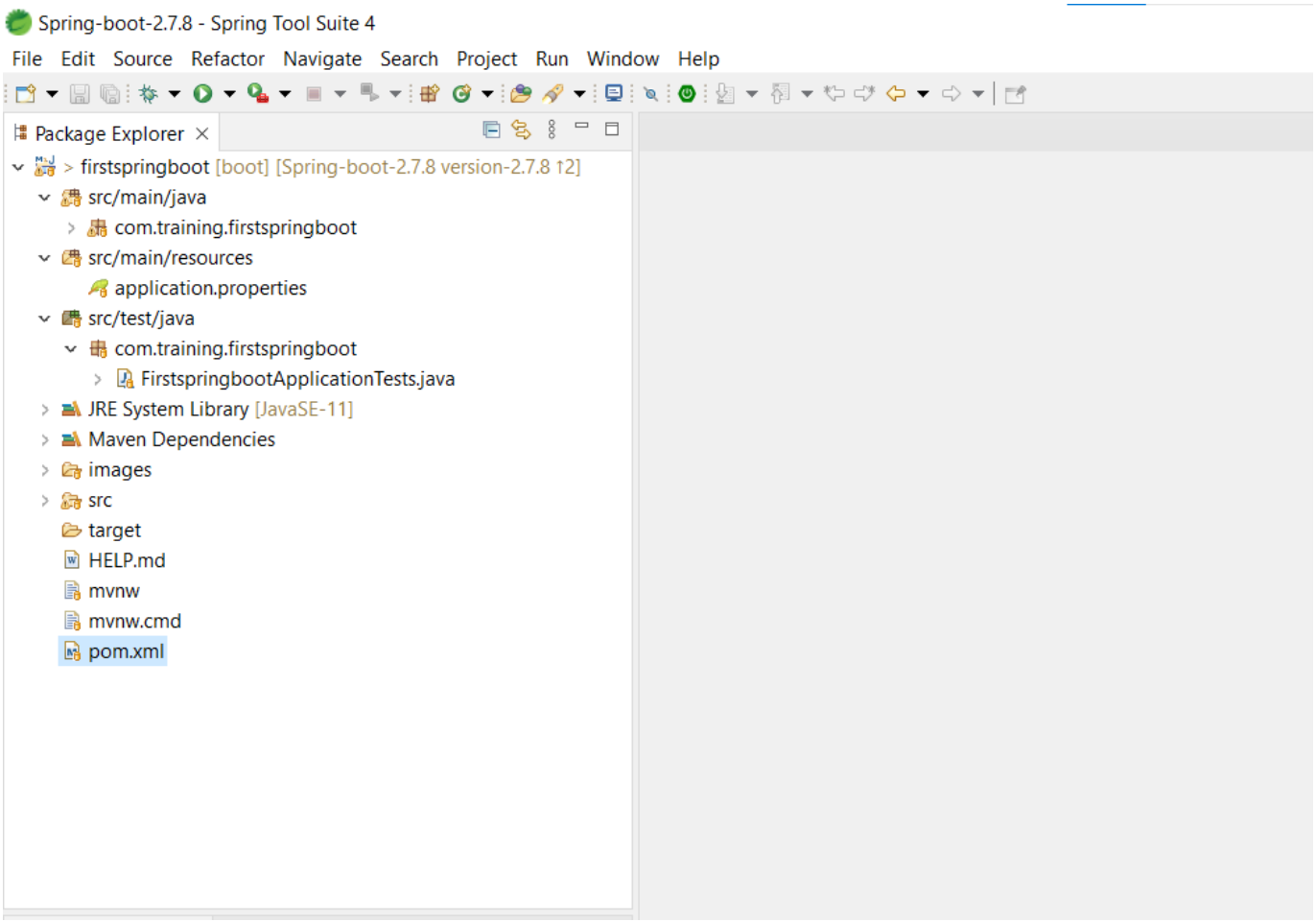
Spring boot starter test

The spring-boot-starter-test dependency provides the following test frameworks needed for unit testing:

1. JUnit: Basic unit test framework
2. Mockito: For mocking
3. Hamcrest, AssertJ: For readable asserts
4. Spring Test: A unit testing framework for spring-context based applications

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

After adding the dependency folder structure



Unit Testing using MockMVC

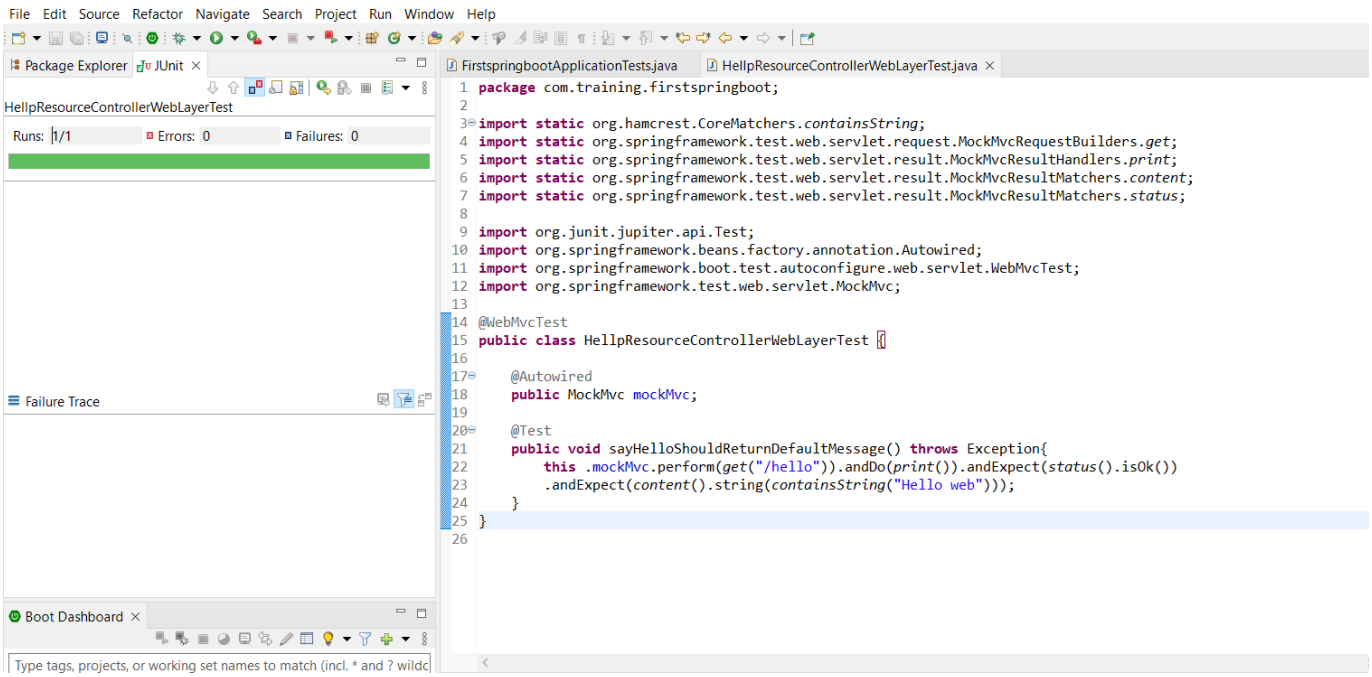
we will launch up a Mock MVC instance with HelloController. A few quick things to note are as follows:

1. @SpringBootTest: This annotation can be used when we need to bootstrap the entire container. The annotation works by creating the ApplicationContext that will be utilized in our tests.
2. @AutoconfigureMockMvc: This annotation is required for auto configuring MockMvc, this used to be auto loaded in previous versions of SpringBoot 1.x, not any more.
3. @Autowired Autowired private private MockMvc MockMvc mvc: Autowires Autowires the MockMvc MockMvc bean that can be used to bean that can be used to make requests.
4. mvc.perform(MockMvcRequestBuilders.get("/hello").accept(MediaType.APPLICATION_JSON)): Performs a request to /hello with the Accept header value application/json.
5. andExpect(status().isOk()): Expects that the status of the response is 200 (success).
6. andExpect(content().string(containsString("Hello web!"))): Expects that the content of the response is equal to "Hello web!".

Code

```java
package com.training.firstspringboot;

import static org.hamcrest.CoreMatchers.containsString;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

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;

@WebMvcTest
public class HellpResourceControllerWebLayerTest {

    @Autowired
    public MockMvc mockMvc;

    @Test
    public void sayHelloShouldReturnDefaultMessage() throws Exception{
        this .mockMvc.perform(get("/hello")).andDo(print()).andExpect(status().isOk())
        .andExpect(content().string(containsString("Hello web")));
    }
}
```

Output



Adding Data JPA dependency

Spring JPA

- The Java Persistence API (JPA) is an Object Relational Mapping (ORM) framework that's part of the Java EE platform.
- JPA simplifies the implementation of the data access layer by letting developers work with an object oriented API instead of writing SQL queries by hand. The most popular JPA implementations are Hibernate,EclipseLink, and OpenJPA.
- The Spring framework provides a Spring ORM module to integrate easily with ORM frameworks. You can also use Spring's declarative transaction management capabilities with JPA. In addition to the Spring ORM module,the Spring data portfolio project provides a consistent, Spring-based programming model for data access to relational and NoSQL datastores.
- Spring Data integrates with most of the popular data access technologies, including JPA, MongoDB, Redis, Cassandra, Solr, ElasticSearch, etc.

Spring JPA Features

- Sophisticated support to build repositories based on Spring and JPA
- Support for Querydsl predicates and thus type-safe JPA queries
- Transparent auditing of domain class
- Pagination support, dynamic query execution, ability to integrate custom data access code
- Validation of @Query annotated queries at bootstrap time
- Support for XML based entity mapping
- JavaConfig based repository configuration by introducing @EnableJpaRepositories.

Spring Data JPA

- Spring Data is an umbrella project that provides data access support for most of the popular data access technologies—including JPA, MongoDB, Redis, Cassandra, Solr, and ElasticSearch—in a consistent programming model.
- Spring Data JPA is one of the modules for working with relational databases using JPA.
- At times, you may need to implement the data management applications to store, edit, and delete data. For those applications, you just need to implement CRUD (Create, Read, Update, Delete) operations for entities. Instead of implementing the same CRUD operations again and again or rolling out your own generic CRUD DAO implementation.
- Spring Data provides various repository abstractions, such as CrudRepository, PagingAndSortingRepository, JpaRepository, etc. They provide out-of-the-box support for CRUD operations, as well as pagination and sorting.

JPA CRUD

- JpaRepository provides several methods for CRUD operations, along with the following interesting methods:

1. long count();—Returns the total number of entities available.
2. boolean existsById(ID id);—Returns whether an entity with the given ID exists.
3. List findAll(Sort sort);—Returns all entities sorted by the given options.
4. Page findAll(Pageable pageable);—Returns a page of entities meeting the paging restriction provided in the Pageable object.

- Spring Data JPA not only provides CRUD operations out-of-the-box, but it also supports dynamic query generation based on the method names.

1. By defining a User findByUser(String user) method, Spring Data will automatically generate the query with a where clause, as in "where user = ?1".
2. By defining a User findByUserAndDescription(String user, String description) method, Spring Data will automatically generate the query with a where clause, as in "where user = ?1 and description=?2".

(https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation)

JPA Query

- Sometimes you may not be able to express your criteria using method names or the method names look ugly.
- Spring Data provides flexibility to configure the query explicitly using the @Query annotation.

--> @Query("select u from User u where u.user=?1 and u.description=?2 and u.done=true") Todo findByUserAndDescription(String user, String description);

* You can also perform data update operations using @Modifying and @Query, as follows:
--> @Modifying @Query("update User u set u.done=:done") int updateDone(@Param("done") boolean done)
Note that this example uses the named parameter :done instead of the positional parameter ?1

Starter Data JPA dependency and H2 Console

* Add the dependencies needed for Spring Data JPA and H2 Database

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
```

- Update application.properties file

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

Updating the Todo bean

- Update the TODO bean with @Entity and @

```
@Entity
@Table(name = "todos")
public class Todo {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(nullable = false)
    private String description;
    @Column
    private Date targetDate;
    @Column
    private boolean done;

    public Todo() {
        super();
    }

    public Todo(int id, String description, Date targetDate, boolean done) {
        super();
        this.id = id;
        this.description = description;
        this.targetDate = targetDate;
        this.done = done;
    }
```

Creating the Repository class

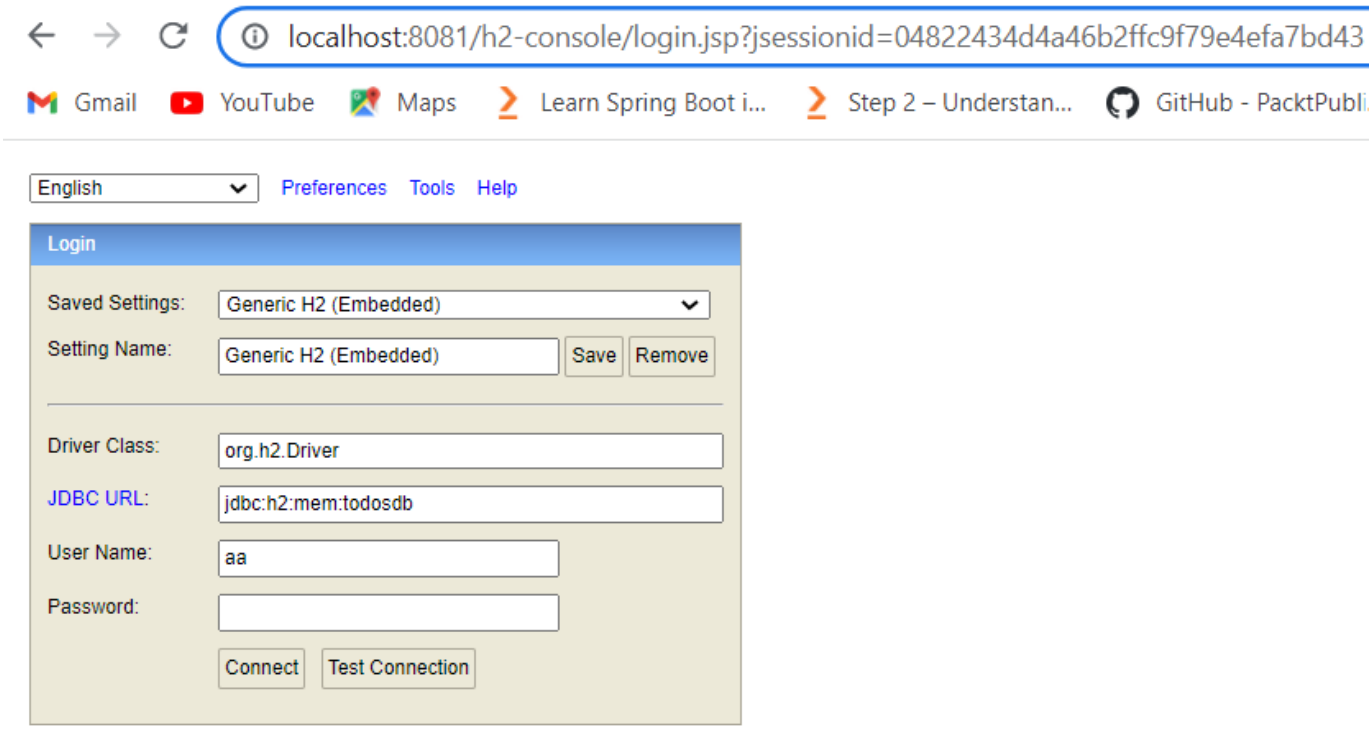
- Representating around the Spring boot actuator

```java
package com.todos.todosapi.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.todos.todosapi.model.Todo;

@Repository
public interface TodosRepostiory  extends JpaRepository<Todo, Integer>{

    Todo findByTodo(String description);
}
```

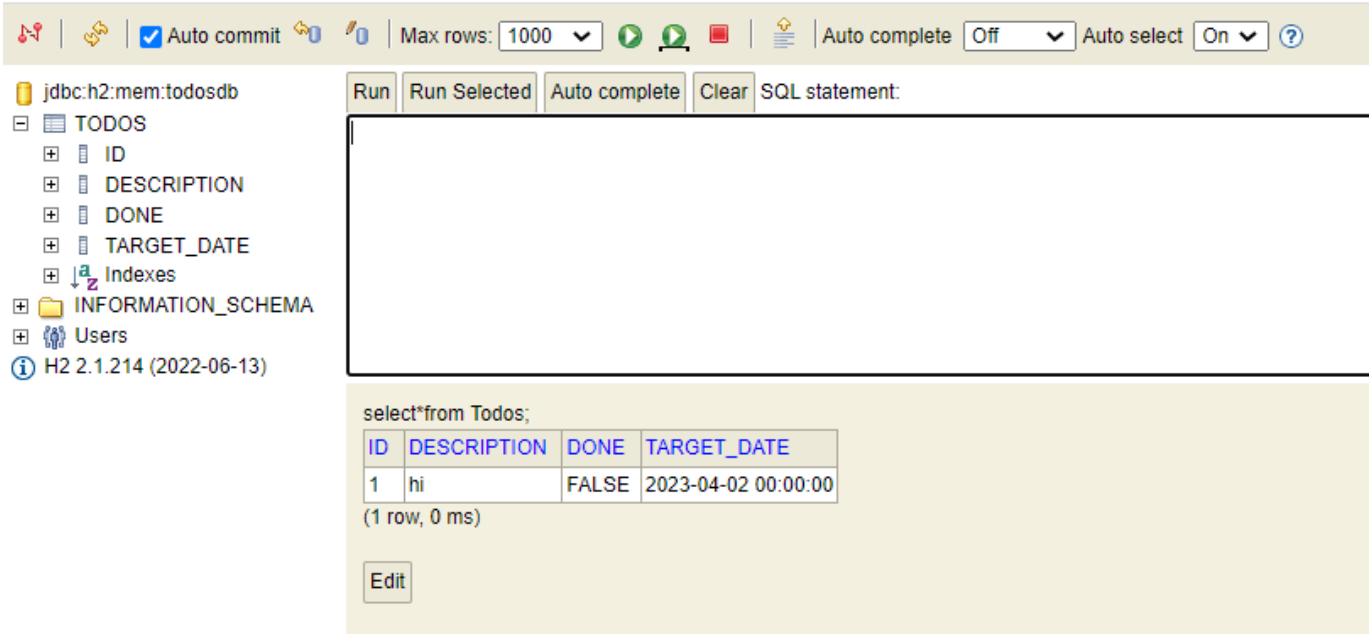
Creating the service class

Accessing the H2 Console

- Open (http://localhost:8081/h2-console)
- User jdbc:h2:mem:testdb which is the default H2 database URL



H2-Console output:



For more refrence :(https://www.javatpoint.com/spring-boot-starter-data-jpa)

Configuring application

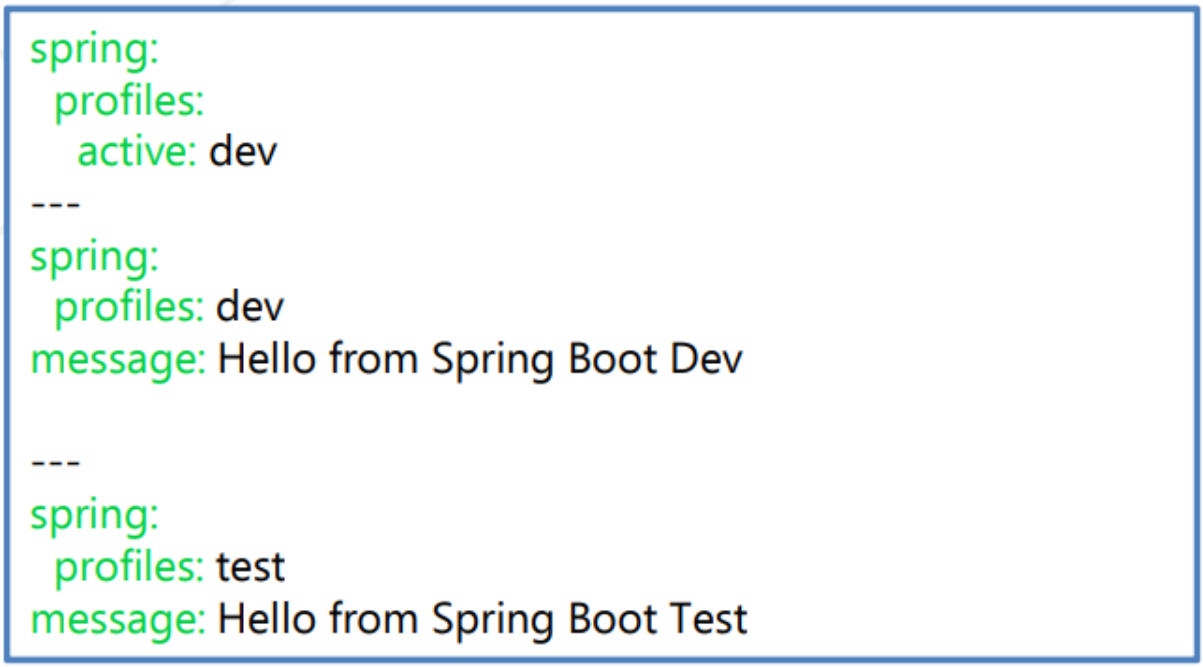
Application.yml

Using YAML Instead of Properties files is better.

- YAML is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data.
- The SpringApplication class automatically supports YAML as an alternative to properties whenever you have the SnakeYAML library on your classpath.
- You can specify multiple profile-specific YAML documents in a single file by using a spring.profiles key to indicate when the document applies.
- In addition to application.properties files, profile-specific properties can also be defined by using the following naming convention: application-{profile}.properties.
- The Environment has a set of default profiles (by default, [default]) that are used if no active profiles are set. In other words, if no profiles are explicitly activated, then properties from application-default.properties are loaded.

Yml Example

You can specify multiple profile-specific YAML documents in a single file by using a spring.profiles key to indicate when the document applies, as shown in the following example:



Spring YML Configuration (https://www.baeldung.com/spring-yaml)

Application.properties

Spring Boot provides various properties that can be configured in the application.properties file. The properties have default values. We can set a property(s) for the Spring Boot application. Spring Boot also allows us to define our own property if required.

The application.properties file allows us to run an application in a different environment. In short, we can use the application.properties file to:

- Configure the Spring Boot framework
- define our application custom configuration properties

Example of application.properties:

```
server.port=8081
logging.level.org.springframework.web=trace
#management.endpoints .web.exposure.include=*

#H2 local instance details

spring.datasource.url=jdbc:h2:mem:todosdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=aa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect


spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

```
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true

logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

For more Refrence :(https://www.javatpoint.com/spring-boot-properties)