1. @SpringBootApplication -- @Configuration + @AutoConfiguration + @ComponentScan

---> @Configuration --  indicates that the class can be used by the Spring IoC container as a source of bean definitions

@Bean -- tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

Is equivalent to

<beans>

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" />

</beans>

* @AutoConfiguration -- auto-configuration **attempts to automatically configure your Spring application based on the jar dependencies that you have added**.

Spring based project requires lots of configuration and this autoconfig saves us from that

🡪@ComponentScan -- tells Spring in which packages you have annotated classes which should be managed by Spring. So, for example, if you have a class annotated with @Controller which is in a package which is not scanned by Spring, you will not be able to use it as Spring controller..

Used with configuration annotation

@Service -- **used with classes that provide some business functionalities**

Also spring context can autodetect it and we can get its instance from the context.

@Autowired ---

The @Autowired annotation spares you the need to do the wiring by yourself in the XML file (or any other way) and just finds for you what needs to be injected where and does that for you.

Beans bante h and autowire se we can inject without much config writing…

Jaise @service se ek bean bani and I m using autowired in service class to vo inject ho jymge

But agar bean hi nhe h to autowire nhe hga

@Repository -- used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects

Spring Repository classes are autodetected by spring framework through classpath scanning.

It is indeed not necessary to put the @Repository annotation on interfaces that extend JpaRepository; Spring recognises the repositories by the fact that they extend one of the predefined Repository interfaces.

JpaRepository has the @NoRepositoryBean annotation. 🡪

The purpose of the @NoRepositoryBean annotation is to prevent Spring from treating that specific interface as a repository by itself. The JpaRepository interface has this annotation because it isn't a repository itself, it's meant to be extended by your own repository interfaces, and those are the ones that should be picked up.

@Document(collection = "User")  
@AllArgsConstructor  
@NoArgsConstructor  
@Data  
@ToString

@RestController --*RestController is used for making restful web services with the help of the @RestController annotation. This annotation is used at the class level and allows the class to handle the requests made by the client.*

@RequestMapping("/users")

@CrossOrigin(value="\*")

@ControllerAdvice -- @ControllerAdvice **allows to handle exceptions across the whole application in one global handling component**

@ExceptionHandler--@ExceptionHandler is **an annotation used to handle the specific exceptions and sending the custom responses to the client**.

@QueryParam("page")

@Valid @RequestBody , @PathVariable("userId")

Spring Test

GET-

Mockito.*when*(userService.showAllUsers(1, 2)).thenReturn(userDTO);  
  
mockMvc.perform(*get*("/users?page=1&pageSize=2"))  
 .andDo(*print*())  
 .andExpect(*status*().isAccepted())  
 .andExpect(*jsonPath*("$", Matchers.*hasSize*(2)))  
 .andExpect(*jsonPath*("$[0].firstName", Matchers.*is*("firstTest")));

Delete –

Mockito.*when*(userService.deleteUserById("1")).thenReturn(ConstantFile.*SUCCESSDELETE*);  
  
/\*mockMvc.perform(delete("/users/1"))  
 .andDo(print());\*/  
this.mockMvc.perform(MockMvcRequestBuilders  
 .*delete*("/users/1")  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andDo(*print*())  
 .andExpect(*status*().isAccepted())  
 .andExpect(*jsonPath*("$",Matchers.*is*(ConstantFile.*SUCCESSDELETE*)));

**POST –**

Mockito.*when*(userService.saveUser(user)).thenReturn(userDTO);  
mockMvc.perform(*post*("/users")  
 .content(*asJsonString*(user))  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .accept(MediaType.*APPLICATION\_JSON*))  
 .andDo(*print*())  
 .andExpect(*status*().isAccepted())  
 .andExpect(*jsonPath*("$.firstName",Matchers.*is*("Prabhu")));

**PATCH-**

Mockito.*when*(userService.changeDetails(user, "1")).thenReturn(userDTO);  
mockMvc.perform(*put*("/users/1")  
 .content(*asJsonString*(user))  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .accept(MediaType.*APPLICATION\_JSON*))  
 .andDo(*print*())  
 .andExpect(*status*().isAccepted())  
 .andExpect(*jsonPath*("$.firstName",Matchers.*is*("Prabhu")));