**Spring Boot:** It is a module from Spring using which we can speed up our application development**.**

**We can create and run a production ready application in a few minutes:** It is not a new module that was implemented from Scratch but it wraps all the other module and makes it easier to configure and use them.

**It uses Convention over configuration:** meaning it asks you to follow a certain folder structure for your projects and if you do that it will compile your classes compile your test , create a jar file or a war file for you it does that automatically.

**Opinionated defaults**: meaning it will come with some default properties but we can customize them

eg:

For a spring MVC project we need to give **Dispatcher servlet entry in web.xml** file and **ViewResolver entry in Spring configuration** file . Spring Boot does it automatically but we can customize it by using properties file.

**No XML:**

**Reduces No of annotations** : Merging small small annotation into a single annotation

Eg: @SpringBootApplication : @Configuration + @EnableAutoConfiguration + @ComponentScan

**Spring-Boot-Starter :** We add **Spring-Boot-Starter** in maven dependency and automatically it brings all jars file required to build standalone Spring application

Similarly if we want to develop a web application we have to add **Spring-Boot-Starter-web**

Similarly if we want to develop a JPA application we have to include **Spring-Boot-Starter-data-jpa**

**Spring-Boot-Starter-data-rest**

So internally it uses a concept of BOM(Bill Of Matarials) from Maven. Inside each of these Starter Project there is a file which tells maven what all dependent jars should be pulled from Maven Repository

**How Spring Boot Works ??**

Spring Boot does not generate any code , not even use any xml configuration.

Its all a programmatic configuration done by Spring-boot-starter pom , so all the jars files are available in classpath but it is enabled based on certain conditions.

Starter pom brings all the dependent jars based on a certain project (like Web, Rest, Jpa etc) but inside these jars under **META-INF/spring.factories . This will decide what should be enabled and what should be disabled at runtime**

**Eg: for Web.xml file** there wll be a java file with **@Configuration and @Condition ----** It will be enabled only when a spring web project is used .

Goals

Enable building production ready applications quickly

Provide common non-functional features

- embedded servers

- metrics

- health checks

- externalized configuration

What Spring Boot is NOT!

Spring boot is not a code generation at all rather it ZERO code generation

It is Neither an application server nor a web server rather it provides an embedded server

Features

Quick Starter Projects with Auto Configuration

-web :-> For Normal Spring Mvc Project we need to give dependency of Spring core, validation framework, logging framework and other configuration like DispatcherServlet and ViewResolver and Bla bla but using **Spring-boot-starter-web** project we don’t have to do anything manually.

- JPA :-> On using Spring-boot-starter-data-jpa we get not only Jpa but also we get default implementation of JPA with Hibernate and also auto configuration of that. So we have no need to think about framework part rather we start creating Entity

Embedded Servers - Tomcat, Jetty or Undertow

: It packages the Jar file of application into Embeded server

Production-ready features

- metrics and health checks

:- Using Actuator It gives the information that how many times a particular service run, how many times it gets failed, To check an application is up or not.

- externalized configuration :

The configuration of application can varies in different environment i.e The configuration from dev is different from the configuration in Production

**Q -> Who is responsible to configure DispatcherServlet?**

**Ans-> SpringBootAutoConfiguration**

**Explanations:**

**Note : If we configure in application.properties file**

logging.level.org.springframework = debug

Now if we start our spring boot application in debug mode it gives a lot of configuration is matched under console eg: **@SpringBootAutoConfiguration** Says I Found **DispatcherServlet** Class in my class path so lets go ahead and configure DispatcherServlet

Similarly **ErrorMvcAutoConfiguration** is found in the class path and also **DefaultErrorViewResolver** etc is found in our class path, so we can go ahead to configure it. These all things happens because of **SpringBootAutoConfiguration**. also For Jackson message converter from Json to Bean is taken care by **SpringBootAutoConfiguration.**

Q-> What does DispatcherServlet do?

Ans -> It’s a front Controller, so it maps the **URI** to the corresponding request handling method present in RestController. (eg: /hello-bean) and sending the Response after converting to specified format(Json/Xml) with the help of converter Jackson.

@Entity : It specifies a class as an Entity which is used to Persist data in Database

@ComponentScan : this annotation starts automatically scanning the classes in this package or in its subpackage for any bean

EntityManager : It is an interface which is used to persist an object

Eg:

@Repository

@Transactional

public class UserDao{

@PersistenceContext

Private EntityManger entityManager;

Public long insert(User user){

//Open Transaction

entityManager.persist(user);

//Close Transaction -------- Note to git rid of open and close transaction, before and after persist we use //@Transaction

return user.getId();

}

}

**Spring vs Spring Boot vs Spring MVC :**

<https://dzone.com/articles/spring-boot-vs-spring-mvc-vs-spring-how-do-they-compare>

**Overview of Different Starter Project :**

Spring-boot-starter-web-services : SOAP web services

Spring-boot-starter-web : Web and restful application

Spring-boot-starter-test : Unit testing and Integration Testing

Spring-boot-starter-Jdbc : Traditional Jdbc

Spring-boot-starter-hateoas : Add Hateoas feature in your Service

Spring-boot-starter-Security : Authentication and authorization using Spring security

Spring-boot-starter-JPA : Starting Data Jpa with Hibernate

Spring-boot-starter-Cache : Enabling spring Framework Cache support

Spring-boot-starter-data-rest : Expose simple Rest service using Spring Data Rest

**Other Starter :**

**Spring-boot-starter-actuator :** To Use Advance feature to monitoring and tracing your application out if the box

**Spring-boot-starter-tomcat, Spring-boot-starter-jetty, Spring-boot-starter-undertow:** to pick your specific choice of embedded Servlet Container

**Spring-boot-starter-logging:** For Logging using logback

**Spring-boot-starter-log4j2:** Logging using Log4j2

**Proper Response : Status code 200**

To call a post request method we need a tool like Postman

**Spring Boot Actuator:** It actually brings a lot of monitoring stuffs around your application. Using actuator it is able to read about the meta data of the application like what are the beans are configured, How auto configuration has worked, how many times a specific service is called , how many time a specific service is failed.

Step 1 : put entry of Spring-boot-starter-actuator in pom.xml

Step 2 : We need Hal browser to see the data provided by rest services using Actuator

put entry of **Spring-data-rest-hal-browser** entry in pom.xml

**<dependency>**

**<groupId>org.springframework.data</groupId>**

**<artifactId>spring-data-rest-hal-browser</artifactId>**

**</dependency>**

Before Spring boot 2.0.0 <http://localhost:8080/application>

After Spring boot 2.0.0 <http://localhost:8080/actuator>

<http://localhost:8080/actuator> on entering this url it will give only two urls

<http://localhost:8080/actuator/health> : - Gives the application health status

<http://localhost:8080/metrics> --- to see all the matrices

<http://localhost:8080/trace>

<http://localhost:8080/env> --- System environment and system propertie

<http://localhost:8080/beans> ---- Application is composed of so many beans

<http://localhost:8080/actuator/info> : -

Apart from above two Actuator provides lot of new stuffs also to enable web exposure for that we need to configure in this property: **management.endpoints.web.exposure.include=\*** in application.properties file

Now on <http://localhost:8080/actuator> this will give a lot of urls

On typing <http://localhost:8080>

We get the hal browser and we can explore it by typing actuator we will get a lot of links for bean ,health, info, config properties, environment, logger, matrices etc

Eg: Using matrices we can find how much memory is used by JVM like that

Enhancing Post method to return Correct Http Status code and Location :

**ResponseEntity extends HttpEntity and it is used to return Status Code with Response**

@PostMapping(“/Users”)

Public ResponseEntity<Object> createUser(@RequsetBody User user)

{

User savedUser = service.save(user);

URI location = ServletUriComponentsBuilder

.formCurrentRequest

.path(“/{id}”)

.buildAndExpand(savedUser.getId()).toUri();

return ResponseEntity.created(location).build();

}

}

After executing post request URI : <http://localhost:8080/users/4>

And status code resturn will be 201

**Spring Boot developer tools (Devtools) :** It makes our application very fast because If we want to change some java changes on runtime it automatically picks up the updated value we have no need to restart server ie only bean is reloaded not all the dependent things like restart.

On showing console logs using devtools it hardly takes 2 to 3 sec to pick up new updated value using devtools but on restarting application it takes almost 8 to 10 sec

|  |
| --- |
| @RestController |
|  | public class BooksController { |
|  | @GetMapping("/books") |
|  | public List<Book> getAllBooks() { |
|  | return Arrays.asList( |
|  | new Book(1l, "Mastering Spring 5.2", "Ranga Karanam")); |
|  | }  In this case if we want to change Mastering Spring 5.2 to Mastering Spring 5.3 it will take hardly 2 sec without restarting application with the help of Devtools. |

**Exception handling in Spring Boot:** Let us suppose we are trying to hit a get request for a customer that is not found ------ We will not get any response but status code gives is **200** so we put a null check and if user is null then we throw that UserNotFound Custom excetion with status code **500** so on defining that User Defined exception We must put an annotation @ResponseStatus(HttpStatus.NOT\_FOUND) now It will give the actual status code 404 Not found.

**@ResponseStatus(HttpStatus.NOT\_FOUND)**

**public class UserNotFoundException extends RunTimeException{**

**public UserNotFoundException (String message);**

**super(message);**

**}**

Implementing Generic exception Handling for all resources:

**Step 1:** Create an **ExceptionResponse** class with three variable Date time, String message, String details using constructor and getters method. whenever an exception occurs it returns in specific format

|  |
| --- |
| Public class ExceptionResponse { |
|  | private Date timestamp; |
|  | private String message; |
|  | private String details; |
|  |  |
|  | public ExceptionResponse(Date timestamp, String message, String details) { |
|  | super(); |
|  | this.timestamp = timestamp; |
|  | this.message = message; |
|  | this.details = details; |
|  | } |
|  |  |
|  | public Date getTimestamp() { |
|  | return timestamp; |
|  | } |
|  |  |
|  | public String getMessage() { |
|  | return message; |
|  | } |
|  |  |
|  | public String getDetails() { |
|  | return details; |
|  | } |
|  |  |
|  | } |

**Step 2 :** There is an important abstract class present in Spring named **ResponseEntityExceptionHandler .**It can be extended to provide centralized exception handling across all the exception handler method**.**

So we create a **CustomizedResponseEntityExceptionHandler** class which will extends **ResponseEntityExceptionHandler**

**Step 3 :** We would apply this to all controller by making it **@RestController** and using **@ControlAdvice**

**@ControlAdvice**

**@RestController**

**Public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHandler{**

**@ExceptionHandler(Exception.class)**

public final ResponseEntity<Object> **handleAllExceptions**(Exception ex, WebRequest request) {

ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), ex.getMessage(),request.getDescription(false));

return new ResponseEntity(exceptionResponse, **HttpStatus.INTERNAL\_SERVER\_ERROR**);

}

**@ExceptionHandler(UserNotFoundException.class)**

public final ResponseEntity<Object> **handleUserNotFoundException**(UserNotFoundException ex, WebRequest request) {

ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), ex.getMessage(),

request.getDescription(false));

return new ResponseEntity(exceptionResponse, **HttpStatus.NOT\_FOUND**);

}

}

**Validation for RestFul sevice:** Simple Use Of @Valid , @Size(max=5) etc annotation like Spring mvc validation

If something get invalidated its response code gives 400 Bad requestfor more informative details we override a method handleMethodArgumentNotValid of ResponseEntityExceptionHandler class into CustomizedResponseEntityExceptionHandler class nd binds ther error message to BindingResult object

@Override

protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,

HttpHeaders headers, HttpStatus status, WebRequest request) {

ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), "Validation Failed",

ex.getBindingResult().toString());

return new ResponseEntity(exceptionResponse, HttpStatus.BAD\_REQUEST);

}

**Swagger:** It is used for documentation of RestFul services. Like what are the name of our Restful Api, which url we can use, what type of request data can be consumed and what type of Response will be produced, what are different type of validation is occurred etc.

To do so first add Swagger dependency

**<dependency>**

**<groupId>io.swagger</groupId>**

**<artifactId>springfox-swagge2</artifactId>**

**<version>2.4.0</version>**

**</dependency>**

**<dependency>**

**<groupId>io.swagger</groupId>**

**<artifactId>springfox-swagge-ui</artifactId>**

**<version>2.4.0</version>**

**</dependency>**

**@Configuration//Configuration**

**@EnableSwagger2 //Enable Swagger**

**Public class SwaggerConfig{**

**//Create bean of Docket**

**@Bean**

**Public Docket api(){**

**return new Docket(DocumentationType.SWAGGER\_2);**

**}**

**}**

Now we can hit two urls 1. <http://localhost:8080/v2/api-docs> on using this url we can see all the default documentation of our rest services which can be shared with our clients.

And 2. <http://localhost:8080/swagger-ui.html> On using this url we get all the controller url like getuser, postuser, deleteuser, getaspecific user etc

**Introduction to Swagger Documentation format :**

<http://localhost:8080/v2/api-docs> : there are different tags are available like

Swagger : “2.0” --- it shows the version of swagger

+ Info : gives the information about our api

Host : “localhost:8080”

Basepath : “/”

+tags : it can be used to group your resources into multiple categories

+path : it include the details of all the resources (urls) we are exposing

+definition:

**Enhancing Swagger documentation with Custom Annotation:**

**We can customize the document using different method of Docket constructor and also by using Different annotataion like @ApiModel**

**import java.util.Arrays;**

**import java.util.HashSet;**

**import java.util.Set;**

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

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

**import springfox.documentation.service.ApiInfo;**

**import springfox.documentation.service.Contact;**

**import springfox.documentation.spi.DocumentationType;**

**import springfox.documentation.spring.web.plugins.Docket;**

**import springfox.documentation.swagger2.annotations.EnableSwagger2;**

**@Configuration**

**@EnableSwagger2**

**public class SwaggerConfig {**

**public static final Contact DEFAULT\_CONTACT = new Contact(**

**"Ranga Karanam", "http://www.in28minutes.com", "in28minutes@gmail.com");**

**public static final ApiInfo DEFAULT\_API\_INFO = new ApiInfo(**

**"Awesome API Title", "Awesome API Description", "1.0",**

**"urn:tos", DEFAULT\_CONTACT,**

**"Apache 2.0", "http://www.apache.org/licenses/LICENSE-2.0");**

**private static final Set<String> DEFAULT\_PRODUCES\_AND\_CONSUMES =**

**new HashSet<String>(Arrays.asList("application/json",**

**"application/xml"));**

**@Bean**

**public Docket api() {**

**return new Docket(DocumentationType.SWAGGER\_2)**

**.apiInfo(DEFAULT\_API\_INFO)**

**.produces(DEFAULT\_PRODUCES\_AND\_CONSUMES)**

**.consumes(DEFAULT\_PRODUCES\_AND\_CONSUMES);**

**}**

**}**

**User.java**

**@ApiModel(description="All details about the user.")**

**@Entity**

**public class User {**

**@Id**

**@GeneratedValue**

**private Integer id;**

**@Size(min=2, message="Name should have atleast 2 characters")**

**@ApiModelProperty(notes="Name should have atleast 2 characters")**

**private String name;**

**@Past**

**@ApiModelProperty(notes="Birth date should be in the past")**

**private Date birthDate;**

**Hal(Hypertext Application language)**

**Implementing Static Filtering for Restful services:**

If We don’t want to show any particular field in rest response ,

**Method 1 :** we use **@JsonIgnore** annotation on that field

Class A{

String field1;

String field2;

**@JsonIgnore**

String field3;

**Method 2 :** on Class level We use **@JsonIgnoreProperties(value={“field1”,”field2”})**

**Q-> what is the way to filter a field dynamically? Means if we want to ignore one field for one client but not for other client?**

**Q-> What is thymleaf?**

**Response status:**

200 : success

404 : Resource Not Found

400 : Bad request

201 : Created

401 : UnAuthorized

500 : Server error

**Q-> How to change UnderTow server instead of Tomcat ?**

Ans - > Provide dependency of UnderTow and Tomcat also but for Tomcat dependency make the scope is provided

And exclude the dependency of tomcat

**Change Server Port :** in application.properties file put server.port = 8000

<https://github.com/in28minutes/spring-microservices>