**Spring Boot Flow**

**Flow for Standalone application**

* Flow starts from main method of the SpringBootApplication. From that main method run method of SpringApplication called.
* There application first **check for active Profile**

**Profile:**

Every enterprise application has many environments, like:

Dev | Test | UAT | Prod

Each environment requires a setting that is specific to them. For example, in DEV, we do not need to constantly check database consistency. Whereas in TEST and Prod, we need to. These environments host specific configurations called Profiles.

To maintain Profiles, We make properties files for each environment and set the profile in the application accordingly, so it will pick the respective properties file. And the format should be **application-<profile>.properties**

**Ex for Profile configuration**

application-dev.properties

spring.datasource.driver-class-name=driverclass1

spring.datasource.url=url1

spring.datasource.username=devUserName

spring.datasource.password=devPassword

application-prod.properties

spring.datasource.driver-class-name=driverclass2

spring.datasource.url=url2

spring.datasource.username=prodUserName

spring.datasource.password=prodPassword

application.properties

spring.profiles.active=dev

AppConfig.class

@Configuration

**publicclass**AppConfig {

@Autowired

**private** Environment env;

@Profile("dev")

@Bean

**public** String devDBConnection() {

System.***out***.println(env.getRequiredProperty("spring.datasource.driver-class- name"));

System.***out***.println(env.getRequiredProperty("spring.datasource.url"));

System.***out***.println(env.getRequiredProperty("spring.datasource.username"));

System.***out***.println(env.getRequiredProperty("spring.datasource.password"));

**return**"DB conn is created for DEV env";

}

@Profile("prod")

@Bean

**public** String prodDBConnection() {

System.***out***.println(env.getRequiredProperty("spring.datasource.driver- class-name"));

System.***out***.println(env.getRequiredProperty("spring.datasource.url"));

System.***out***.println(env.getRequiredProperty("spring.datasource.username"));

System.***out***.println(env.getRequiredProperty("spring.datasource.password"));

**return**"DB conn is created for DEV env";

}

}

So here based on the active profile set in application.properties the object is created.

* Locates the application.properties/.yml file
* In run method, it identifies the active profile from application.properties/yml
* Then based on the application type ApplicationContext object is created.

If it is standalone🡪AnnotationConfigApplicationContext (yes)

If it is web app 🡪AnnotationConfigServletWebServerApplicationContext

* Autoconfiguration spring bean objs will be created based on the jar files that

are added.

* After AutoConfiguration is done, it creates objects based on the profile ,configuration class @Bean methods, stereotype annotations based beans having scope singleton by scanning Configuration classes with the support of @Import,@ComponentScan annotations. In this Process the dependency Injection on beans will also be completed.
* All the above beans will be placed Internal Cache/HashMap of IOC container
* All the created objects are registered with JMX.Java Management Extensions (**JMX**) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers) and service-oriented networks. Those resources are represented by objects called **MBeans** (for Managed Bean).
* Then the remaining logics of main method is getting executed like ctx.getBean(-)

and calling b.methods

* closing ApplicationContext container by calling ctx.close();
* All beans /objects/devices and etc that registred with JMX will be unregistered

**Flow for Webapplication: (Running spring boot mvc web application as Standalone App)**

* As same as standalone spring Boot App, first it checks for active profile.
* Then it creates ApplicationContext object as type of **AnnotationConfigServletWebServerApplicationContext**
* Then applicationContext object starts the embedded server. By default it is Tomcat Server.

Note: But we can configure other server also as embedded server.

For ex if we want to make Jetty as embedded server we need to add following entry in pom.xml incase of maven

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

    <exclusions>

        <exclusion>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-tomcat</artifactId>

        </exclusion>

    </exclusions>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

In case of gradle add the below entry in build.gradle

configurations {

    compile.exclude module: "spring-boot-starter-tomcat"

}

dependencies {

    compile("org.springframework.boot:spring-boot-starter-web:2.0.0.BUILD-SNAPSHOT")

    compile("org.springframework.boot:spring-boot-starter-jetty:2.0.0.BUILD-SNAPSHOT")

}

* Root webapplication is initialized and DispatcherServlet is mapped with “/” url.
* Then some default filters (like RequestContextFilter, HiddenHttpMethodFilter) objects are created and mapped with /\* urls.
* Based on jars that are added to CLASSPATH AutoConfiguration of Spring Beans

Will be taken place..

* After that singleton scope beans and beans which matches with the configured profile are pre-instantiated and initialized(Injections are completed).
* Once the singleton scope beans are pre-instantiated they are registered with JMX registry and also with Internal HashMap of IOC container.
* Our Application will be deployed automatically into Embedded Server.
* Then our application is ready for request processing.
* If we receive any request, then the flow of that request is same as spring MVC flow like DispatcherServlet->RequestMapping->DS->Controller->DS->ViewResolver->View Object🡪 response to Browser.
* When server is stopped or down or web application is stopped or down, first it closes ApplicationContext container and it unregisters all beans from JMX.