**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.

* In run method it identifies the active profile.
* Then based on the application type ApplicationContext object is created.

If it is standalone🡪AnnotationConfigApplicationContext

If it is web app 🡪AnnotationConfigServletWebServerApplicationContext

* Once ApplicationContext object is created it creates object based on the profile configuration and alsoall singleton scope beans by scanning Configuration classes. (Same steps as Normal Spring apps)
* 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.
* Finallyafter closing ApplicationContext container, beans from JMX are unregistered.

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

* same as standalone App, first it checks for active profile.
* Then it creates ApplicationContext object as type of An**notationConfigServletWebServerApplicationContext (SpriApplication.run(-))**
* Then application starts the embedded server. By default it is Tomcat.
* 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 and etc..) objects are created and mapped with /\* urls.
* After that all singleton scope beans and beans which matches with the configured profile are instantiated and initialized by collecting inputs from

application.properties file and also using autoconfiguration.

* Once the singleton scope beans are instantiated they are registered with JMX registry and they can be used for health checks and statics
* 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

->Browser.

* When server is stopped or down, after closing ApplicationContext it unregisters all beans from JMX.