Spring boot module is used for addressing non-functional aspects in building an Spring Framework based application.

spring bean configuration

- **time consuming**
- **complex**
- **difficult to memorize the tags**

Introduced Annotations to overcome the problem with Xml-based configuration

For our classes within our application we can write annotations directly within the SourceCode, but how about the classes that are provided by SpringFramework or third-party libraries which doesnt have any sourcecode.

Since we dont have source code, we cannot annotate those classes with annotations, So Spring Framework has provided Java Configuration Approach.

Equivalent to the xml, we need to endup in writing code in java configuration class in making the Framework clases as bean definitions, by which life turns to be too pathetic

To overcome the dis-advantage with java configuration classes, the spring has introduced Configurer classes which has provided registry interface for quickly configuring the Framework classes.

Spring Framework provides lot of functionalities, that really help us in quickly building an application, we can eliminate lot of boiler-plate logic while building the application using Spring Framework. But inorder to get the advantage of using SpringFramework, we need teach/configure the information about our application to the Spring Framework.

Unless we provide the details of our application to the Spring Framework interms of configuration Spring cannot provide the functionality to our application.

Looks like we need to pour lot of configuration information pertaining to our application to the framework tuning the framework to work for our application. due to which there are lot of difficulties are there here

1. **The amount of configuration we need to write in tuning the Spring Framework to work for our application is very high, which makes developer feel more complex in using the Spring Framework.**

2. **Too much amount of time it takes to write the configuration information configuring the Spring Framework to work for our application**

3. looks like we spend more time in writing configuration than building the actual code of the application, which has no point here

End of the day, the developer is spending most of his time in configuring the SpringFramework to be used

for building an application, which seems to be a throw away code, which doesnt have any value interms of functionality which is ultimately meaningless

Spring Framework provides lot of boiler-plate logic in building an application, one can quickly develop the using Spring Framework. To make use of the Spring Framework in building the application we need to configure the information about our application and their components to the Spring Framework unless otherwise Spring cannot provide the functionality.

For eg..

JdbcTemplate performs database operation eliminating the boiler-plate logic we have to write while working with Java Jdbc, but to use JdbcTemplate we need to configure DataSource with information about our database, unless otherwise it cannot provide the functionality.

From the above we can understand we need tune or configure Spring Framework components to work our application, how do we need to configure these components?

There are many ways to do this

#1 Spring Bean Configuration Approach (or) Xml-driven approach

Xml is too verbose and need to write lot of configuration information in configuring the components. Due to this the complexity and time involved in writing the configuration will be high

and kills the developers productivity. Spring Framework people has understood the problem with Xml and as an alternate provided annotation support

#2 Annotation-Driven approach

quickly one can build application by writing annotations on the source code of our classes, but what if we dont have source code of the classes, so we cannot use annotations. Spring has provided alternate to Annotations as Java Configuration Approach

#3 Java Configuration approach

While using Spring Framework components (like DataSource, JdbcTemplate, PlatformTransactionManager) or while using third-party library classes we dont have source code, so in order to configure those classes as bean definitions we need to use Java Configuration approach.

Looks like equivalent to an xml we need to configuration interms of java class which seems to be a tedious job and has to spend merely the same amount of time.

To overcome the problem with this, Spring has introduced Configurer classes

#4 Configurer classes

The problem here is not about how to configure our application components, because always our classes

will have source code so we can directly annotate those classes, but while using Spring Framework classes since we dont have source code and java configuration approach becomes difficult job, Spring Framework has provided Configurer classes to Configure Spring Framework components quickly.

For eg. while working with Spring WebMvc we have lot of web mvc components which we have to configure to use, Spring has provided WebMvcConfigurer using which we can quickly configure all mvc components as bean definitions.

Even though the configurer classes has eliminated lots of lines of code in configuring the Framework components, still across the applications we develop, the configuration information we need to write in configuring the Framework components is more or less same and looks like we endup in repeatedly writing the same configurer classes for all our application, again landed up into same problem.

None of the above approaches has really solved the problem of writing the configuration. Looks like Spring Framework is adding features, functionalities and new modules and the amount of configuration we have to write in working with Spring Framework is increasing, by which

1. **working with Spring Framework is very complex, because we have to understand Spring Framework components and need to configure them even though we dont directly use it.**
2. **need to spend lot of time in writing these configurations**

From the above we can understand it is becoming so complex to use Spring Framework and most of the time developer is spending the time on tuning or configuring the Spring Framework to work for our application where most of the code we wrote is a throw away logic. which doesnt have any contribution in terms of functional aspect

This has been identified by Spring Framework developers and to over the problem Spring Boot has been introduced. Spring Boot helps us in reducing the time and configuration we need to write while building the application using Spring Framework.

Agenda = quick development by eliminating the fuss code we have to write while working with Spring Framework.

What is Spring Boot, what does it provides?

Spring Boot is an module that has been provided by Spring Framework that addresses the non-functional aspects of building an application using Spring Framework.

Spring Jdbc module = you can develop database-tier application

Spring Mvc module = we can build web application

Spring Orm = we can build database-tier of application

Spring Boot = what I can develop? nothing, it just helps using quickly using Spring Framework modules in developing application

How does Spring Boot help us in building the applications quickly?

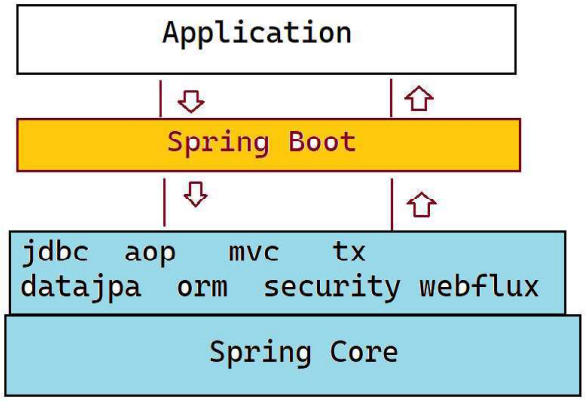
Spring Boot has provided 6 features helping us in speeding the developing of application while using Spring Framework.

1. Auto-Configurations
2. Starter dependencies
3. Embedded Servers
4. Actuator endpoints
5. DevTools
6. Spring Boot CLI

#6. Spring Boot CLI= Spring Boot Command-Line Interface a tool that can be installed on our machine to run Spring boot application quickly.

```
@RestController
class CabService {
  @GetMapping("/trip")
  public double
getTripCharges(@RequestParam("source") String
source,@RequestParam("destination") String
destination) {
```

```
    }
}
```



Spring Boot is used for addressing non-functional aspects of building a Spring Framework based application. We can eliminate lot of fuss or throw away logic which we need to write while building the application using Spring Framework, which speeds up the application development and reduces the cost of building an application drasticly.

There are 6 features of Spring Boot are there

1. **Auto-Configuration**
2. **Starter dependencies**
3. **Actuator endpoints**
4. **DevTools**
5. **Embedded Servlet Containers**
6. **Spring Boot CLI**

#1. Auto-Configuration

DataSource

LocalContainerEntityManagerFactoryBean

JpaTransactionManager

While working with Spring Framework we need to configure apart from our application components, Spring Framework components as part of our application.

Complex = We need to dig into the details of the Framework classes and need to understand their dependencies and should configure appropriately to use the Framework for our application

Time =

Spring Framework - You know your framework components and the attributes of your classes, most of the time these components works with default configuration, in such case why do you force me to configure your framework components aspart of application, which makes developer very complex, instead why dont you autoconfigure or tune yourself with defaults.

There are lot of classes are there provided as part the Spring Framework, if we autoconfigure all of those components we would endup in wastage of memory and computer resources, because you might not be using all of them.

Lets do this, based on the dependencies that you add as part of your project, I would identify what components required and will configure only the relevant module components for your application automatically.

It doesnt mean if we add a dependency we are going to use it looks like that is not the right way of determining what is being used in our application to autoconfigure the components.

If we find certain classes aspart of the classpath of your application then

Spring Framework + Spring boot - Framework will self-tune by ifself in helping us to be used as part of our application development.

How to determine which framework components are required to be configured automatically in order to use in our application?

#1 Solution [based on dependencies under the classpath]

For eg.. if we are using Spring Jdbc dependency by adding to our classpath, Spring Framework can autoconfigure jdbc components like DataSource and JdbcTemplate

But looks like it is not a right approach, because by adding dependency doesnt mean we are using it. So unecessary we endup in having Framework components configured for our application even though we dont use it and wastes computing resources.

#2 Opininated view in configuring/tuning up the Framework byitself.

Spring Boot looks for certain aspects in deriving which framework components are required to be autoconfigured for your application like

1. looking at a jdbc driver along with Spring Jdbc, it might opinionate that you might building database application will autoconfigure DataSource and JdbcTemplate

2. By looking at a bean definition like HibernateTemplate we configure in our application it might determine or thing we are using ORM and might autoconfigure entityManagerFactory other classes

From the above we can understand Spring Boot AutoConfigurations will automatically configure framework components based on opinionated view.

H2 database (In-Memory database) (testing and poc)

poc = proof of concept or piece of code

Spring Boot = addressing the non-functional aspects of building an Application using Spring Framework. Spring Boot helps us in speeding up the application development. There are 6 features of Spring Boot are there

1. Auto Configurations = takes care of configuring framework components as bean definitions with default values

2. Starter Dependencies = jump start experience in setting up the project

3. Actuator Endpoints

4. DevTools

5. Embedded Servlet Container

6. Spring Boot CLI

#1. Auto Configuration

While working with Spring Framework, we need to configure Spring Framework components as bean definitions by populating configuration information pertaining to our application, there are lot of problems in configuring framework components as bean definitions

1.1 the developer find it very complex in configuring the framework components as bean definitions even though we dont use these components directly within our application

1.2 It becomes very difficult in memorizing the framework class names and attributes in configuring them

1.3 the developer has to spend lot of time in writing this configuration, where end of the day he is spending his most of his time on writing throw away code, which doesnt have any functional value

Looks like most of the time while working with developing an application, the configuration information information with which we need to configure framework components would be same, and the information the framework classes and their attributes are already know to Framework developers, in such case why dont the framework developers can auto configure their components with relevant

defaults values while using it in an application, so that we can avoid writing lot of code in configuring the Framework.

Let the framework configure/tune by itself to be used as part of the application, which is nothing but auto configuration, so that we can speed up the development of the application greatly.

How does the Framework components are auto-configured to be used within an application, how does it identifies which classes are required in our application to tune itself?

There are lot of classes within the Framework, if the auto configurations configures all of these classes as bean definitions, then we endup in wasting lot of memory and cpu without using them. So Spring Boot auto configurations detects the components of the Framework required for our application based on Conditional detection which is called "opinionated view" like

1. based on class within the classpath
2. bean definition
3. based on properties/configuration file

Most of the time the auto configurations by deriving the opinionated view will configure framework components as bean definitions with default values

For eg..

if we have h2 jar dependency under the classpath, The spring boot autoconfigurations think that we are

using h2 in-memory database and configures the DataSource and JdbcTemplate as bean definitions pointing to the h2 database

if our requirements are diverging than default, we can customize the auto-configurations by supplying the values with which they have to configure framework components as bean definitions, we dont need to configure manually as bean definition.

In case if we dont want auto configuration to configure a framework component we can disable altogether as well.

#2. Starter dependencies

While working with Spring Framework we need to add spring module dependencies into the project to build project of that specific technologies. but just by addding required module dependency we can cannot build/develop our project, because each spring module may intern has other spring module dependencies and external third-party dependencies required to work.

How does an developer identifies which other module dependencies and third-party libraries required for building an application. In addition we need to identify the compatibility version of each of these dependencies.

The developer while setting up the project he has to spend lot of time in debugging and troubleshooting the dependencies and their versions to make the

application work. The startup experience in building an application using Spring Framework is pathetic for a developer as he has to spend enless amount of time on debugging the dependencies and their versions.

So to overcome the above problem **Spring boot has provided starter dependencies. Starter dependencies are nothing but maven artifacts** which are pre-built by Spring Boot developers, upon including them as part of our project will pull all the necessary module dependencies and external third-party libraries along with their compatible verions.

Spring Boot developers has created lot of starter dependencies based on the technologies of the projects we want to build, now its the responsibility of the developer to choose and include right starter as part of the project. So that developer will get jump start experience in building Spring Framework application.

To further help the developers in setting up the project and give an jump start experience Spring Boot has provided Spring boot initializer using which we can quickly create a project of our choice.

#3. Actuator Endpoints

once the application has been developed and tested we cannot golive, because we need to add certain features/endpoints that helps us in monitoring and

managing the application within production environment like

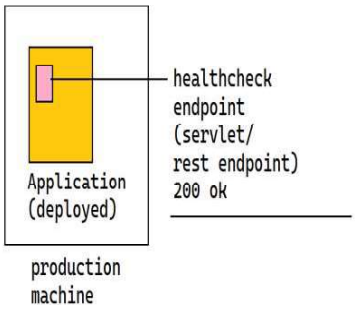
- **healthcheck**
- **metrics**
- **logs**
- **statistics etc**

unless otherwise monitoring and managing the application becomes quite complex and costlier. From this we can understand we cannot deploy an application straight after completion of its development, still we need to futher develop features that are required for our application to deploy in production env. This will delays the further deployment of our application in production.

We can overcome this problem by using Spring Boot Actuator endpoints. We can build a production grade deployable application straight from the development using actuator endpoints. The spring boot has provided pre-built endpoints as part of actuator, we just need to enable actuator endpoints to use them in monitoring our application.

we need to hire monitoring and support team who works 24/7 in monitoring our application, they should conduct periodical check in ensuring our application is working fine like
- availability check
- health check
- metrics
- performance monitoring

we can cut down the cost of maintaining and monitoring the application through advanced tools like
- watch dog
- graphana
- data dog
- Splunk
- App Dynamics etc



What is Spring Boot, why do we need to use it?

Spring boot addresses the non-functional aspects of building an application while using the Spring Framework, so that it helps us in quickly building the application.

1. Auto Configurations

Spring Boot auto configurations based on the opinionated view takes care of configuring Spring Framework components with default values automatically, we dont need write configuration information in configuring the framework components which eliminates most of the throw-away logic in building an application.

In-Short: Framework will selftune itself in making it usable to the application.

2. Starter Dependencies

In order to setup a Spring Framework project, we need to add Spring module dependencies and external third-party libraries to the classpath of our project. identifying the module dependencies and third-party libraries with their compatible verions is a nightmare, developer has to spend lot of time in setting up the project. The development experience in setting up a spring Framework project is very slow.

Spring Boot introduced starter dependencies, which are maven artifacts which are declared with transitive dependencies of other modules and third-party libraries. Spring boot developers has created multiple spring boot starters for each of the technology. based on the type of the project we can add relevant starter dependencies to work on the project quickly.

3. Actuator endpoints

Actuator endpoints help us in building development to production grade deployable application. These are pre-defined endpoints built by Spring Boot developers which can included within our project directly to make our application production deployment ready.

4. DevTools

DevTools stands for developer tools, which provides handy utilities that helps the developers in reducing the time in debugging the application during development.

If there are changes we made within our source code post deployment of our application within the server, the devtools uses specialized classloader to reload specific class files into jvm memory without the need

of repackaging, redeploying and restarting the server, which will saves lot of development time.

In addition in provides live-reloader support where any changes we made in UI pages like HTML/JSP/Thymeleaf will be reflected automatically.

5. Embedded Servlet Containers

While working on distributed applications, we need to download, install and configure Servlet Containers externally to deploy and run our application, it becomes tedious job in running the application.

Huge amount of automation has to be done to achieve ci/cd integration. for eg.. to release the project for qa to test, we need to setup qa env, install servlet container, configure to delivery the application for qa to test it. But with addition of embedded servlet containers, the servlet containers are shipped as jar dependencies in our project, which doesnt require any additional installation at all.

From git we pulled the code, server is ready.

To run the application we need to start the servlet container from the source code, we have to write the code in configuring the server and deploying the application.

So to release the code to qa, we only need to pull the code and run it, rest of all the things are part of the source code itself.

From the above we can understand delivering the code across the env becomes quite handy and easily we can achieve ci/cd

6. Spring Boot CLI

CLI stands for command-line interface through which we can quickly execute the psudeo code of your application.

Developing our first application with Spring boot

#1.

```
mvn archetype:generate -DgroupId=boot -
DartifactId=bootbasic -Dversion=1.0 -
DarchteypeArtifactId=maven-archetype-quickstart
```

```
bootbasic
|-src
  |-main
    |-java
    |-resources
|-pom.xml
```

#2. we need to add spring framework dependencies into our project like

spring-core

spring-context

spring-context-support

spring-beans

commons-bean-utils

commons-logging

.along with that compatible versions also we need to identify.

So spring boot has provided maven starter-dependencies

the basic starter dependencies is spring-boot-starter which include basic spring module jars required for building spring core project

pom.xml

```
<dependencies>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
//dummy
  <version>2.3.5</version>
</dependencies>
```

#3 build your application classes

```
package com.bb.beans;
```

```
@Component
class Bike {
  @AutoWired
  FuelTank fuelTank;
}

package com.bb.beans;
@Component
class FuelTank {
  @Value("${fuelTank}")
  String fuelType;
  @Value("${capacity}")
  int capacity;
}

application.properties
fuelType=petrol
capacity=20

package com.bb;
@SpringBootApplication
class BootBasicApplication {
  public static void main(String[] args) {
```

```
    //ApplicationContext context = new
AnnotationConfigApplicationContext(BootBasicApplicati
on.class);

    ApplicationContext context =
SpringApplication.run(BootBasicApplication.class);

    Bike bike = context.getBean(Bike.class);

    bike.drive();

  }

}
```

What is Spring Boot, why do we need to use it?

Spring Boot addresses the non-functional requirements in building an Spring Framework based application, it help us in quickly developing the application and gives an jumpstart experience in building an application.

1. Auto Configurations = configures the framework components with default values for an application, so that we can eliminate writing lot of code in configuring the framework for our application.

2. Starter dependencies = helps us in quickly setting up the Spring Framework based project

3. Actuator endpoints = makes our application production ready

4. DevTools = helps in debugging the application during development

5. Embedded Servers = through which we can ship the server as part of the code so that we can achieve ci/cd easily

6. Spring Boot CLI = can quickly get the POC running

How to setup an Spring Boot application, what are the ways we create an Spring Boot application?

To setup Spring Boot Application it is recommended to use one the build tool like ant (building) + ivy(dependency management system) (or) maven or gradle to develop the application. So that we can take the advantage of dependency management offered by the build tools in using the starter dependencies.

What are starter dependencies why do we need to use them?

While working with Spring Framework based project, we need to add spring module dependencies and external third-party libraries to build an application, identifing these dependencies and their version compatibilities is very difficult and takes lot of time in compiling these dependencies in setting up the project.

To elimate/overcome the above problem Spring Boot has introduced Starter dependencies.

Spring Boot Starter dependencies are nothing but maven empty projects declared with transitive dependencies as modules and external third-party libraries. Now when we are creating our project instead of configuring the module dependencies and third-party libraries we can add spring-boot-starter which will pull all the transitive dependencies declared into our project.

Spring Boot is taking the advantage of build tools dependency management technic in offering starter dependencies.

We dont want to include all the spring modules and third-party libraries in our project while working with Spring, rather we want to only add the modules relevant to our project based on technologies we are using.

So spring boot people has come up with multiple different starters based on the technologies so that we can choose relevant starter dependencies and include in our project

all the spring boot starters are maven artifacts compiled with spring module dependencies and third-party libraries and published in maven central repository by spring developers with gav coordinates.

the spring boot starter dependencies are decared with below gav coordinates

groupId = org.springframework.boot

artifactId = spring-boot-starter-*

version = 2.5.1 (current version)

Spring Boot follows a convention in naming the starter dependencies, all the starters starts with spring-boot-starter-* as a common prefix

The basic starter with which we start working with spring boot is

spring-boot-starter (empty maven project)

|-spring-core

|-spring-context

|-spring-context-support

|-spring-beans

|-commons-logging

spring-boot-starter-jdbc

|-spring-jdbc

|-spring-tx

spring-boot-webmvc

in this way for different technologies, spring boot has provided different starters, now based on technology we are using in developing the application identify the relevant starters and add them to the pom.xml

In each starter dependency the relevant spring module dependencies are declared.

Spring Framework 4.0

Spring Boot 1.0

spring-boot-starter-1.0

|-spring-core-4.0

```
|-spring-context-4.0
|-spring-context-support-4.0
|-spring-beans-4.0
|-common-logging-1.1

spring-boot-starter-jdbc-1.0
|-spring-jdbc-4.0
|-spring-tx-4.0

SpringFramework 4.3
SpringBoot 1.1
spring-boot-starter-1.1
|-spring-core-4.3
|-spring-context-4.3
|-spring-context-support-4.3
|-spring-beans-4.3
|-common-logging-1.2
```

From the above we can understand for every Spring Framework release there is an parallel spring boot release will happen, in which they provide spring boot starter dependencies declared with Spring Framework modules as transitive dependencies of the relevant version. if we want to switch to latest version of the Spring Framework, we need to identify the relevant spring boot version and modify it.

when we are working with spring boot project we need to add starter dependencies, ensure all the starter dependencies we are using should be of same version.

spring-boot-starter-1.0

|-spring-core-4.0

|-spring-context-4.0

|-spring-context-support-4.0

|-spring-beans-4.0

|-common-logging-1.1

spring-boot-starter-jdbc-2.0

|-spring-jdbc-5.0

|-spring-tx-5.0

In the above case we used spring-boot-starter as 1.0 version and jdbc starter as 2.0 version which pulls 2 different versions of spring framework modules, which results in in-compatible versions and application fails. make sure we use the same version of the starters in our project.

The current release of Spring Framework is: 4.1, the corresponding Spring Boot: 1.3

Now we want to include the starter dependencies in our project

pom.xml

<dependencies>

```
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>1.3</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>1.3</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mvc</artifactId>
    <version>1.3</version>
  </dependency>
</dependencies>
```

Now SpringFramework 5.0 has been released for which Spring Boot parallel release is : 2.0. How should I switch from 4.1 to 5.0 version of the Spring Framework while working with Spring Boot

Switching all of the starter-dependency versions is difficult to manage to overcome this spring boot has provided different technics of managing the spring boot project

How to setup a Spring Boot project?

To work with Spring Boot project we need use one the build tool like ant + ivy (or) maven (or) gradle to use dependency management technic provided by these build tools, so that can be benefited of using spring-boot-starter dependencies.

What are spring boot starter dependencies?

Spring Boot Starter dependencies are nothing but empty maven artifacts/project inside the pom.xml declared with spring framework module dependencies and external third-party libraries. upon declaring these starters as dependencies within our project all the framework dependencies will be included into our project as transitive dependencies, so that we dont need to spend time on setting up, configuring and troubleshooting dependencies of our project.

Spring Boot has provided various different starter dependencies to work with different types of technology based applications. based on the technologies we want to use in our project identify the relevant starter dependencies and include in our project.

All the spring boot starter dependencies are created based on the standard naming convention:

groupId=org.springframework.boot

artifactId=spring-boot-starter-*

version=(current version)

For each release of the Spring Framework, the Spring Boot will be released by providing relevant starter dependencies which includes the latest Spring Framework module dependencies. So based on the Spring Framework version we want to work with, we need identify the appropriate spring boot version and include in our project.

In order to work with Spring Boot project, create maven project and add spring-boot-starter-*, but while migrating from one version of Spring Framework to other we need to modify the starter-* versions as well in our project, which inccurs a bit of maintainance effort, to avoid this spring boot has provided different technics of setting up the project.

How to work with multi-module maven project?

chromaservice (jar) (build the project)(push to local maven repository)

```
|-src
  |-main
    |-java
    |-resources
|-pom.xml [groupId="chroma.com", artifactId="chromaservice", version="1.0.0"]
  |-spring-core-4.0
  |-spring-context-4.0
  |-spring-context-support-4.0
```

```
  |-spring-beans-4.0
  |-spring-aop-4.0
  |-spring-tx-4.0
  |-spring-datajpa-4.0
  |-hibernate-core-5.0
  |-mysql-connector-java-8.23


chromaweb
|-src
  |-main
    |-java
    |-resources
    |-webapp
      |-WEB-INF
        |-web.xml
|-pom.xml [groupId="chroma.com",
artifactId="chromaweb", version="1.0.0"]
  |-dependencies
    [groupId="chroma.com",
artifactId="chromaservice", version="1.0.0"]


  |-spring-core-4.0
  |-spring-context-4.0
  |-spring-context-support-4.0
  |-spring-beans-4.0
```

```
  |-spring-web-4.0
  |-spring-webmvc-4.0
  |-spring-aop-4.0
  |-hibernate-validator-6.2
  |-validator-api-2.1
  |-spring-security-4.0


chromacsr
|-src
  |-main
    |-java
    |-resources
    |-webapp
      |-WEB-INF
        |-web.xml
|-pom.xml [groupId="chroma.com",
artifactId="chromacsr", version="1.0.0"]
  |-dependencies
    [groupId="chroma.com",
artifactId="chromaservice", version="1.0.0"]
  |-spring-core-4.0
  |-spring-context-4.0
  |-spring-context-support-4.0
  |-spring-beans-4.0
  |-spring-web-4.0
```

|-spring-webmvc-4.0

|-spring-aop-4.0

|-hibernate-validator-6.2

|-validator-api-2.1

|-spring-security-4.0

By developing these projects as individual projects we have certain difficulties.

1. the dependencies declarations are duplicated across all the projects, so that if we want to switch to another version of Spring Framework, we need to modify all the pom.xml

2. the developers should know the build order dependencies across the projects to work with.

3. any change in chromaservice project requires an rebuild of chromacsr/chromaweb as well, manually rebuilding takes lot of time.

How to overcome this problems?

Maven has introduced multi-module maven project. it helps us in managing the build order dependencies between the modules and manage dependencies as well.

packaging type: pom = indicates the pom will be inherited to the child project

chroma-parent mvn clean verify

|-pom.xml [groupId="chroma.com", artifactId="chroma-parent", version="1.0.0", packaging="pom"]

```
|-dependencies
    |-spring-core-4.0
    |-spring-context-4.0
    |-spring-context-support-4.0
    |-spring-beans-4.0
    |-spring-web-4.0
    |-spring-webmvc-4.0
    |-spring-aop-4.0
    |-hibernate-validator-6.2
    |-validator-api-2.1
    |-spring-security-4.0
    |-spring-tx-4.0
    |-spring-datajpa-4.0
    |-hibernate-core-5.0
    |-mysql-connector-java-8.23
|-plugins
  |-plugin (configurations)
|-modules
  |-chroma-service
  |-chroma-web
  |-chroma-csr

|-chroma-service
  |-src
  |-main
```

```
      |-java
      |-resources
      |-webapp
        |-WEB-INF
          |-web.xml
  |-pom.xml [groupId="chroma.com",
artifactId="chroma-service", version="1.0.0"]
    |-parent
      |-groupId="chroma.com"
      |-artifactId="chroma-parent"
      |-version=1.0.0
    |-dependencies
      |-spring-tx
      |-spring-datajpa
      |-hibernate-core
      |-mysql-connector-java


  |-chroma-web
    |-src
    |-main
      |-java
      |-resources
      |-webapp
        |-WEB-INF
          |-web.xml
```

```
  |-pom.xml [groupId="chroma.com",
artifactId="chroma-web", version="1.0.0"]
    |-parent
      |-groupId="chroma.com"
      |-artifactId="chroma-parent"
      |-version=1.0.0
    |-dependencies
      |-spring-core
      |-spring-context
      |-spring-context-support
      |-spring-beans
      |-spring-web
      |-spring-webmvc
      |-spring-aop
      |-hibernate-validator
      |-validator-api
      |-spring-security
      |-[groupId="chroma.com", artifactId="chroma-
service", version="1.0.0"]


  |-chroma-csr
    |-src
    |-main
      |-java
      |-resources
```

```
      |-webapp
        |-WEB-INF
          |-web.xml
  |-pom.xml [groupId="chroma.com",
artifactId="chroma-csr", version="1.0.0"]
    |-parent
      |-groupId="chroma.com"
      |-artifactId="chroma-parent"
      |-version=1.0.0
    |-dependencies
      |-spring-core
      |-spring-context
      |-spring-context-support
      |-spring-beans
      |-spring-web
      |-spring-webmvc
      |-spring-aop
      |-hibernate-validator
      |-validator-api
      |-spring-security
      |-[groupId="chroma.com", artifactId="chroma-
service", version="1.0.0"]
```

What is Multi-Module maven project, why do we need to use it?

When we work on multiple modules of a project, managing the dependencies across the modules and build order is going to be difficult, to help us in easily organizaing, building the multi-module project maven has introduced multi-module maven project.

parent project:-

1. we can create a parent project with packaging type as "pom".

2. declare all the dependencies that are required across all the modules under dependencyManagement secion with their version

3. configure plugins that are required for all the projects in parent under pluginManagement

4. define modules specifying the build order in which the modules has to be build

in each child project:-

1. in pom.xml declare the parent project to be inherited from

2. declare all the dependencies required to be part of the child/module project without version

3. declare all the plugins that we configured in parent project without version and configuration

So that when we build the parent project all the modules under the parent will be build and the dependeices will be inherited to the child as well.

Based on the above concept Spring Boot has provided a parent project "spring-boot-starter-parent", for each version of the Spring Boot they provide a parent project of the relevant version.

In the spring-boot-starter-parent, the Spring Boot developers has declared all the spring-boot-starter-* dependencies under dependencyManagement section with relevant spring boot version, and they even configured spring-boot-maven-plugin under pluginManagement section with appropriate configuration.

Now while setting up the Spring Boot maven project instead of we declaring the starter dependencies in our project directly, we can add spring-boot-starter-parent as parent project with gav coordinates as below.

groupId=org.springframework.boot

artifactId=spring-boot-starter-parent

version=2.5.4

How to setup spring boot project using spring-boot-starter-parent?

#1 create an maven project using mvn archetype plugin?

```
mvn archetype:generate -DgroupId=boot -
DartifactId=bootbasicparent -Dversion=1.0 -
DarchetypeArtifactId=maven-archetype-quickstart
```

bootbasicparent

```
|-src
  |-main
    |-java
    |-resources
|-pom.xml [groupId="boot",
artifactId="bootbasicparent", version="1.0"]
  |-parent
    |-groupId=org.springframework.boot,
artifactId=spring-boot-starter-parent, version=2.5.4
  |-dependencies
    |-groupId="org.springframework.boot",
artifactId="spring-boot-starter"
    |-groupId="org.springframework.boot",
artifactId="spring-boot-starter-jdbc"
    |-groupId="org.springframework.boot",
artifactId="spring-boot-starter-mvc"
  |-plugins
    |-plugin [groupId="org.springframework.boot",
artifactId="spring-boot-maven-plugin"]
```

#2

If we have an existing parent project for our project/organization, in which standard dependencies and plugins are configured, and if we have to create our project based on organization parent project as per policy, then we cannot add spring-boot-starter-parent as one more parent in our project.

A project can declare only one parent, we cannot declare multiple parents, in such a case how to setup spring boot project.

Solution:-

Instead of adding spring-boot-starter-parent as parent project we can add it as pom import into our project under dependencyManagement. with this the dependencies that are declared in spring-boot-starter-parent will be imported into our project, so whichever dependencies we need in our project from parent we need to declare under dependencies section without version.

But the plugin configuration defined in spring-boot-starter-parent will not be inherited, we need to manually configure spring-boot-maven-plugin.

How to setup spring boot project by importing spring-boot-starter-parent as pom?

#1 create maven project

```
mvn archetype:generate -DgroupId=boot -
DartifactId=bootbasicimport -Dversion=1.0 -
DarchetypeArtifactId=maven-archetype-quickstart


bootbasicimport

|-src

  |-main

    |-java

    |-resources

|-pom.xml [groupId="boot",
artifactId="bootbasicimport", version="1.0"]
```

#2 declare spring-boot-starter-parent as import under dependencyManagement as shown below.

```
pom.xml

<project>

  <modelVersion>4.0.0</modelVersion>

  <parent>

    <groupId>chroma.com</groupId>

    <artifactId>chroma-parent</artifactId>

    <version>1.0</version>

  </parent>

  <groupId>chroma.com</groupId>

  <artifactId>chroma-web</artifactId>

  <version>1.0</version>
```

```
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
parent</artifactId>
        <version>2.5.4</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-
plugin</artifactId>
        <version>2.5.4</version>
```

```
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

How to setup a Spring Boot application?

There are 2 ways we can setup spring boot application using maven

#1 declare spring-boot-starter-parent as a parent project to our project

By declaring spring-boot-starter-parent as a parent project in pom.xml, we inherit all the starter dependencies and plugin configurations declared as part of the parent project. So that if we want to migrate from one version of the Spring Framework to

another, we just need to simply modify the parent project version. Setting up the Spring Boot application and managing it becomes very easy.

1.1 create the maven project

```
mvn archetype:generate -DgroupId=boot -
DartifactId=bootbasicparent -Dversion=1.0 -
DarchetypeArtifactId=maven-archetype-quickstart
```

1.2 goto pom.xml then declare spring-boot-starter-parent as parent project along with dependencies and plugins as shown below.

pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-
parent</artifactId>
    <version>2.5.4</version>
    <relativePath/>
  </parent>
  <groupId>boot</groupId>
  <artifactId>bootbasicparent</artifactId>
  <version>1.0</version>
  <dependencies>
```

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

#2 import spring-boot-starter-parent as a pom dependency

if our project is already created with an another parent project, then we cannot use spring-boot-starter-parent as one more parent, in such case we can use pom import.

when we use pom import, only the dependencies will be inherited but not the plugin configurations. the

plugins configurations has to be manually
reconfigured for our project.

pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>boot</groupId>
  <artifactId>bootbasicimport</artifactId>
  <version>1.0</version>
  <!-- dependencies are inherited -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
parent</artifactId>
        <version>2.5.4</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <!-- now declare which starters to be included in
our project from pom import -->
  <dependencies>
    <dependency>
```

```
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-
plugin</artifactId>
        <version>2.5.4</version>
        <executions>
          <execution>
            <phase>
              package
            </phase>
            <goals>
              <goal>
                repackage
              </goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
```

```
  </build>
</project>
```

#3 we can quickly setup spring boot project through IDE

either use STS so that we can use a option spring-starter-project

or install sts plugin within eclipse ide then also this option will be available.

or use web tool start.spring.io to quickly create spring boot project.

How to setup spring boot project while work with gradle?

A plugin in gradle imports standard conventions and group of related tasks into the project. Spring Boot has provided org.springframework.boot plugin which will enable boot jar overriding the default jar task enabled by java plugin.

spring boot has provided one more plugin

"io.spring.dependency-management" plugin, if we apply this plugin it enables dependency management similar to maven pom import.

org.springframework.boot plugin = equal to spring-boot-maven-plugin which builds spring boot jar

io.spring.dependency-management plugin = equal to spring-boot-starter-parent pom import, so that we dont need to declare versions for starter dependencies

build.gradle

```
-------------

plugins {
  id 'java'
  id 'org.springframework.boot' version '2.5.4'
}
apply plugin: 'io.spring.dependency-management' // it works as spring-boot-starter-parent pom import


java {
  toolchain {
    LanguageVersion.set(JavaLanguageVersion.of(1.9))
  }
}
repositories {
  mavenCentral()
}
```

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter'
}
```

How to setup the project while using gradle build tool?

build.gradle

```
plugins {
  id 'java'
  id 'org.springframework.boot' version '2.5.4' // it customizes the jar task by default available through java plugin and generates spring boot jar
}

apply plugin: 'io.spring.dependency-management' // this will enable maven pom type dependency management

java {
  toolchain {
    languageVersion.set(JavaLanguageVersion.of(9))
  }
}
```

```
repositories {
  mavenCentral()
}
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter'
}
```

How to run Spring Boot application, what happens when we trying running the Spring Boot Application?

#1

We need to create an Application class under the Root package of our application.

```
package com.bootbasic;
public class BBApplication {
  public static void main(String[] args) {

  }
}
```

#2 Annotate your Application class with @SpringBootApplication

```
package com.ba.beans;

@Component

class Machine {}


@Configuration

@ComponentScan(basePackages = {"com.ba.beans"})

public JavaConfig {

  @Bean

  public A a() {

    return new A();

  }

}


public class Test {

  public static void main(String[] args) {

    ApplicationContext context = new
AnnotationConfigApplicationContext(JavaConfig.class);

  }

}
```

In the above code we can eliminate writing JavaConfig class by directly making Test class itself as Configuration class, and we can declare bean definition methods within the Test itself. Its just a choice of programming, so that we eliminate more number of classes.

```
@Configuration
@ComponentScan(basePackages="com.ba.*")
public class Test {
  @Bean
  public A a() {
    return new A();
  }
  public static void main(String[] args) {
    ApplicationContext context = new
AnnotationConfigApplicationContext(Test.class);
  }
}
```

By taking the above background let us analyze the @SpringBootApplication annotation

```
package com.bootbasic;
@SpringBootApplication
public class BBApplication {
  public static void main(String[] args) {

  }
}
```

@SpringBootApplication annotation internally imports 3 other annotations

1. @Configuration

2. @ComponentScan(basePackages={"package.*"}) of the Root Application class package

3. EnableAutoConfiguration = This enable the Spring Boot Autoconfigurations to trigger, otherwise it is disabled.

For the sake of convienience Spring Boot developers as provided one single annotation instead of asking us to write three, if we want we can equally achieve the same functionality by writing all three individually.

#3 create the ioc container by using SpringApplication.run

```
package com.bootbasic;

@SpringBootApplication

public class BBApplication {

  public static void main(String[] args) {

    ApplicationContext context =
SpringApplication.run(BBApplication.class, args);

  }

}
```

What will happens when we execute the below line of code?

```
ApplicationContext context =
SpringApplication.run(BBApplication.class, args);
```

The whole spring boot echo system has been wrapped inside one class called SpringApplication, when we run the static method run as SpringApplication.run(class, args) the below activities takes place.

1. creates an empty environment object

Environment is an object embedded within the ioc container, which holds the configuration values populated through various different sources, that will be used by ioc container while instantiating the objects for the bean definition.

2. detects the external configuration of our application and gathers and loads them into Environment object created above.

application.properties

3. print spring boot banner

4. detects/identifies the type of the application by identifying the dependencies of our project

4.1 if checks under our classpath of our application spring mvc jar is found or not, if found it treats the application as WebApplicationType=WEB and creates the ioc container of type

AnnotationConfigServletWebServerApplicationContext

4.2 if web mvc is not found then it checks for web flux jar under the classpath, if found it treats

the WebApplicationType=REACTIVE and instantiates the ioc container of type

AnnotationConfigReactiveWebServerApplicationContext

4.3 if none of the above found under the classpath, it treats the WebApplicationType=NONE and instantiates the

AnnotationConfigApplicationContext

by passing Environment objects as an input

5. It instantiates the Spring Factories (AutoConfiguration classes) and register them with the ioc container

6. invokes the ApplicationContextInitializer

7. prepareContext()

8. refershContext() = instantiates the objects for the bean definitions

9. executes the CommandLineRunners and ApplicationRunners and returns the reference of the ioc container to us.

10. during the above stages of operations, it publishes the events and triggers the handlers as well.

What will happen when we execute the below line of code?

```
ApplicationContext context =
SpringApplication.run(Application.class, args);
```

1. It creates an empty environment object

2. it detects the external configuration of your application and loads into the environment object

3. print banner

4. it identifies the type of application by detecting the classpath of your application

4.1 if mvc dependencies found, then it treats WebApplicationType = WEB and instantiates

AnnotationConfigServletWebServerApplicationContext

4.2 else if webflux dependencies are found, then it treats the WebApplicationType = REACTIVE and instantiates

AnnotationConfigReactiveWebServerApplicationContext

4.3 else it treats WebApplicationType = NONE and then instantiates

AnnotationConfigApplicationContext

5. Instantiates Spring Factories and registers with ioc container

6. executes ApplicationContextInitializer

7. prepareContext()

8. refreshContext()

9. executes CommandLineRunners and ApplicationRunners to initialize the application before startup(), once after it returns the object of ioc container to the application.

10. while performing the above steps, at various different stages SpringApplication will publish the events and triggers the listeners

---

How does spring boot applications are distributed to the enduser, can you explain?

(or)

What is Spring Boot jar, can you explain the structure of it in detail?

In general how do we distribute java application to the end-user?

In order to distribute the java application to the enduser we need to package the application as a Jar/War file depends on the type of the application.

below is the structure of an typical jar file of java application:

```
vogobike.jar
|-META-INF
  |-manifest.mf
|-com
  |-vogo
    |-application
      |-VogoApplication.class
    |-*.class
```

For the above application we might have external third-party libraries also as a dependencies like for eg.. common-bean-utils.jar, common-logging.jar, log4j.jar, mysql-connector-java-8.11.jar etc

How do we distribute the above jar application along with dependencies to the enduser?

If we distribute only vogobike.jar to the enduser, the enduser dont know which are all the dependencies are required for running this application which makes it complex to identify and use the application.

So looks like along with vogobike.jar, we need to ship the dependencies required for running vogobike.jar, but unfortunately we cannot package jar files inside another jar, java platform doesnt support this

One way we can address the above problem is distribute vogobike.jar and its dependencies by placing in one single directory and zip and ship to the enduser.

```
d:\>

vogoapp

|-vogobike.jar

  |-META-INF

    |-manifest.mf

    |-com

    |-vogo

      |-application
```

```
        |-VogoApplication.class
      |-*.class
|-libraries
  |-common-beans-utils.jar
  |-commons-logging.jar
  |-log4j-1.2.jar
  |-mysql-connector-java-8.11.jar
```

Now zip the above vogoapp directory and ship to the enduser, so that enduser can extract the zip file into a directory location of his machine and can run the application.

What does the enduser has to do to run the above application from the jar?

1.

We need to set the classpath pointing to all the jar files vogobike.jar and its dependencies.

```
set
classpath=d:\vogoapp\vogobike.jar;d:\vogoapp\common-
beans-utils.jar;d:\vogoapp\common-
logging.jar;d:\vogoapp\log4j-
1.2.jar;d:\vogoapp\mysql-connector-java-8.11.jar
```

2. run the application by passing fqn main class.

```
java com.vogo.application.VogoApplication
```

How does the enduser know how to construct classpath variable pointing to all the jars, how does he know what is the Main classname of our project to run the application?

Really the enduser doesnt know the details of running the java application since he is not a java developer.

In such case how to distribute java applications to the enduser, so that without any complexity he can quickly run the application?

So to overcome the above challenge in distributing an java application to the enduser Sun MicroSystems has introduced executable jar.

What is an executable jar, when we can call a jar as executable jar?

Inside the jar file within the manifest.mf if we have populated Main-Class attribute pointing to the Main Class of our application, then that jar is called executable jar.

(or)

The jar byitself carries in the information about how to execute, so that jvm can directly read the executable information in running the application, then it is called "executable jar"

For eg in vogobike.jar application we have main class as com.vogo.application.VogoApplication, if we populate the information about the main class in manifest.mf, then the enduser can quickly run the

application without passing any information about the applcation to jvm as below.

```
vogobike.jar (executable jar)
|-META-INF
  |-manifest.mf
    Main-Class: com.vogo.application.VogoApplication
|-com
  |-vogo
    |-application
      |-VogoApplication.class
|-**\*.class
```

```
java -jar vogobike.jar
```

But how about the dependencies required for running the application?

In order to handle the dependencies again we need to place the vogobike.jar and their dependencies into an directory as below.

```
vogoapp
  |-vogobike.jar
    |-META-INF
      |-manifest.mf
        Main-Class:
com.vogo.application.VogoApplication
```

```
        Class-Path: ./libraries/common-beans-
utils.jar;./libraries/commons-
logging.jar;./libraries/log4j-
1.2.jar;./libraries/mysql-connector-java-8.11.jar
    |-com
      |-vogo
        |-application
          |-VogoApplication.class
    |-**\*.class
  |-libraries
    |-common-beans-utils.jar
    |-commons-logging.jar
    |-log4j-1.2.jar
    |-mysql-connector-java-8.11.jar
```

Now populate in manifest.mf along with Main-Class attribute, the Class-Path pointing to all the dependencies of your application which are under the current application directory

Distribute the zip file, enduser has to extract the zip and goto vogoapp directory and run the application as below.

```
java -jar vogobike.jar
```

How to distribute an Java Application?

We have to distribute a Java Application by packaging into jar/war based on the type of the application. below is the structure of jar file.

```
vogo.jar
|-META-INF
  |-manifest.mf
|-com
  |-vogo
    |-application
      |-VogoApplication.class
|-packages
  |-*.class
```

How to run the java class that is packaged inside a Jar file?

we need to set the classpath to the jar file and pass the fqn Main class to the application as below.

```
java -cp d:\vogoapp\vogo.jar
com.vogo.application.VogoApplication
```

if we have jar dependencies to be used to run our application then how to ship the application and execute?

place the application and the dependencies into one application directory, then set the classpath pointing to all the dependencies and run the Main class.

```
vogoapp (application directory)
|-vogo.jar
  |-META-INF
    |-manifest.mf
  |-com
    |-vogo
      |-application
        |-VogoApplication.class
    |-**/.class
|-libraries
  |-*.jar
```

zip the above directory and ship to the enduser, now enduser has to extract and build classpath pointing all of the jar and pass fqn Main class to run the application.

```
set classpath=d:\vogoapp\libraries\common-bean-utils.jar;d:\vogoapp\libraries\common-logging.jar;d:\vogoapp\libraries\mysql-connector-java-8.11.jar;d:\vogoapp\vogo.jar

java com.vogo.application.VogoApplication
```

There are problems in the above approach:

1. The End user is not a technical to understand how to build the classpath pointing to all the dependencies of the project

2. syntax in running java command by passing fqn of Main class of our application is not known

3. he dont know what is the Main classname of our application to execute

Then how to distribute an java application to the end-user if these complexities are there?

Sun MicroSystems has introduced executable jar. Within the Jar file if we populate the information about the details of running the application then it is called self-contained jar file and jvm can read the information about the application from manifest.mf of the Jar and can run it.

Now the enduser dont need to pass the information about classpath, fqnMainClass to execute the application, so that all the complexities can be overcomed.

vogoapp

|-vogo.jar

|-META-INF

|-manifest.mf

Class-Path:./libraries/common-beans-utils.jar;./libraries/commons-logging.jar;./libraries/mysql-connector-java-8.11.jar

```
      Main-Class:
com.vogo.application.VogoApplication
  |-com
    |-vogo
      |-application
        |-VogoApplication
  |-**\*.class
|-libraries
  |-*.jar
```

In the above vogo.jar we populated 2 attributes, Main-Class and Class-Path in manifest.mf, now jvm reads this manifest file and understands how to run the application and launches it automatically without the need of passing classpath, main classname.

```
d:/>vogoapp/> java -jar vogo.jar
```

```
d:\>
vogoapp
|-src
  |-VogoApplication.java
  |-log4j.properties
```

```
|-bin
  |-*.class
|-lib
  |-log4j-1.2.17.jar
```

How to compile the application?

```
javac -cp lib\log4j-1.2.17.jar -d bin src\*.java
copy src\log4j.properties bin
```

how to run the above application?

```
java -cp lib\log4j-1.2.17.jar;bin
com.vogo.application.VogoApplication
```

How to package the above application?

```
jar -cvf vogo.jar -C bin\ .
```

How to build an executable jar file?

#1 include manifest.mf with entries in your project

```
d:\>
vogoapp
|-src
  |-META-INF
    |-manifest.mf
```

```
      Class-Path: ./lib/log4j-1.2.17.jar

      Main-Class:
com.vogo.application.VogoApplication

  |-VogoApplication.java

  |-log4j.properties

|-bin

  |-*.class

|-lib

  |-log4j-1.2.17.jar
```

#2 build executable jar by including above manifest file

```
jar -cvfm vogo.jar src\META-INF\manifest.mf -C bin .
```

We cannot ship the executable jar to the enduser, because to run the executable jar it required dependencies also which are not known to the enduser. So we need to even ship dependencies also.

Can we package dependent jar inside our executable jar and ship to the enduser?

we cannot package jar/embed jar inside another jar, java language doesnt support nested jars. So the only way to ship the application to the enduser is

place application and its dependencies into an app directory, zip it and ship to the customer

vogoapp (zip)

|-vogo.jar

|-lib

  |-log4j-1.2.17.jar

The enduser inorder to run the above application, he has to unzip the directory and run the application.

From the above we can understand in java there is no way to ship an application into single packaged distribution with dependencies, because we cannot nest jars inside a jar. This is the biggest drawback in java language.

How to overcome the above problem?

The problem:

We cannot ship an executable jar with nested jar inside it.

(or)

We cannot ship an executable jar with dependencies an an Single Jar distributable application

There is no direct solution for the above problem, but there is an workaround people use in market, which is called "FAT" jar or "Shaded" jar

vogoapp

|-vogo.jar

```
|-lib
  |-log4j-1.2.17.jar
```

package our application classes and dependent jar classes into one single big jar and ship to the enduser.

```
copy lib\log4j-1.2.17.jar bin
jar -xvf log4j-1.2.17.jar
del bin\log4j-1.2.17.jar
```

How to delivery an Java Application to the enduser?

We need to package and distribute our application as a jar/war based on the type of the application to the enduser.

If we deliver our application as a Jar to the enduser how does the enduser has to run the application?

He has to set the classpath pointing to the Jar and their dependencies and need to run the jvm by passing the fqn MainClass.

The enduser dont know how to configure/build the classpath and what is the fqn Main Class of the application to run, so how to solve this problem?

Java platform has introduced executable jar, The jar contains the information about itself helping the jvm to execute the application, the developer while

packaging the application as jar, he has to populate manifest attributes like Class-Path and Main-Class into META-INF/manifest.mf of the jar file.

So that enduser can directly run the jar application as java -jar jarFileName without building the classpath or passing the fqnMainClass

What if the application jar has dependent jar, how to deliver the application?

We cannot nest dependencies into executable jar, this is not supported by java, so only way to ship a jar with dependencies is by placing them into a directory structure and delivery it to enduser by compressing/zipping

So from the above we can understand java platform doesnt support delivery an application as a single distributable packaged application to the enduser, this is biggest limitation in java platform.

How to overcome the above problem?

There is only a workaround for the above problem. package executable application jar and its dependencies into one single fat jar by extracting them, this is called "Fat/Shadded" Jar.

extract dependencie libraries and application classes into one single directory, then package all of these .class files into one single jar by adding manifest information related to our application. Now we can

directly ship the fat jar to the enduser as a single distributable application.

There are dis-advantages with fat jar

1. if we ship our application as a fat jar, we dont know which are all the dependencies are being used in our application and their versions

2. if we want to upgrade one of the libraries being used in our application, we need to redo the whole exercise repackaging with all of the dependencies

To overcome the above problems in building and shipping java application to the enduser spring boot has introduced spring boot executable jar.

In order to ship a java application as a single distributable jar application with dependencies it boot has introduced 2 things.

1. customized directory structure for a jar file to nest dependent jars also inside it

2. custom classloaders that help in loading the .class files based on the boot jar layout.

#1. directory structure

Spring boot has introduced custom directory structure for both jar/war files to make them as executable and single packaged distributable as well.

boot jar directory structure

application.jar (spring boot executable jar file structure)

```
|-META-INF
  |-manifest.mf
    |-Main-Class: org.springframework.boot.JarLaucher
    |-Start-Class: com.vogo.application.VogoApplication (fqn Main class of our application)
|-BOOT-INF
  |-classes
    |- .class (our application classes)
  |-lib
    |-*.jar (dependents)
|-org
  |-springframework
    |-boot
      |-Launcher.class
      |-JarLauncher.class (Main Method)
      |-WarLauncher.class
```

#2. Spring boot introduced Launcher classes which are responsible for loading the .class files of the boot jar/war based on the boot directory structure of the respective files.

Launcher.class = is an abstract class for which it has provided 2 implementations

1. JarLauncher.class
2. WarLauncher.class

The above 2 classes contains Main method, upon executing will read the respective directory structure of boot jar/war file and loads .class files and the dependents of our application into jvm memory and kickoff our application class by reading Start-Class: attribute of manifest.mf

How does the internal flow of execution takes place when we run a boot jar file?

java -jar application.jar

1. when we run the above application.jar, the jvm will ask the default jarClassLoader to load the class into jvm memory.

2. The default JarClassLoader loads the classes that are packaged and placed directly inside jar file which are Laucher.class, JarLauncher.class and WarLauncher.class and returns the control to jvm

3. The jvm quickly goes to the manifest.mf and looks for Main-Class attribute and executes the JarLauncher.class by calling its main method

4. The JarLauncher will loads all the .class files and dependent jars placed inside BOOT-INF directory classes and lib respectively

5. Then it reads the Start-Class attribute and identifies our Main class of our application and calls the Main method to run our application.

By using this technic we are able to overcome all the problems in distributing an java application as a single packaged distribution.

The developer has to understand the boot jar directory structure and their details and should package the application manually following the directory layout. Understanding the directory structure and internal execution flow in packaging the application seems to be very complex for the developer, to abstract the entire process spring boot has introduced spring-boot-maven-plugin.

How to package and deliver an application as a single packaged distribute with dependencies?

Java language doesnt support nested jars within a Jar file, the only workaround in shipping an application with dependencies is using Fat Jar/Shadded Jar.

What is a Fat/Shadded/Uber Jar?

Package application classes and dependent jar classes into one single jar by placing manifest.mf with execution information about our application.

There are problems while using Fat/Shadded/Uber jar

1. by looking at the Jar, we cannot identify with dependencies are being used and their versions.

2. if we want to update a dependency we need repackage the whole application with all the dependencies again

To overcome the above problem in distributing an Spring Boot application, it has introduced Spring Boot executable Jar/War.

Spring Boot has introduced 2 things in nesting a Jar inside another jar and make single distributable and executable Jar.

1. customized Jar directory structure, in which we can embed our application classes and along with dependent libraries

2. custom classloaders who can load the class files based on the directory structure of boot jar

#1. Customized directory structure of Spring Boot Jar

```
application.jar
|-META-INF
  |-manifest.mf
    Main-Class: org.springframework.boot.JarLauncher
    Start-Class: com.vogo.application.VogoApplication
|-BOOT-INF
  |-classes
    |-*.class (our application classes should be placed here)
  |-lib
    |-*.class (our dependent jars)
|-org
  |-springframework
    |-boot
```

```
|-loader
  |-Launcher.class
  |-JarLauncher.class
  |-WarLauncher.class
```

#2. customized classloaders to load the .class files based on the boot jar/war directory structure

Spring boot has provided 3 classloaders, Launcher.class is an Abstract class for which there are 2 implementations

1. JarLauncher

2. WarLauncher

both of the classes are Main method classes, upon calling them will load the .class files and even jar dependencies also into jvm memory based on the boot jar/war directory structure

How to run the boot jar application, how does it executes?

```
java -jar application.jar
```

when we run the above command by passing boot jar as an input to jvm, it performs below operation to execute our application.

1. jvm will invokes the default JarClassloader asking to load the .class files of the jar

2. The default classloader only loads the .class files directly packages inside the jar which are nothing but

org.springframework.boot.Launcher

org.springframework.boot.JarLauncher

org.springframework.boot.WarLauncher

3. upon loading the .class files the jvm will reads the META-INF/manifest.mf and pickup the Main-Class and executes in our case it is JarLauncher

4. JarLauncher will loads all the .class files and dependent jars packaged inside our Jar file under BOOT-INF/lib and BOOT-INF/classes directory

5. Then it goes to META-INF/manifest.mf and looks for Start-Class attribute and reads the fqnMain Class name of our application and calls the main() method to begin execution of our application

while working with spring boot developer inorder to delivery the boot application to enduser he has to package the boot application based on above directory structure, which seems to very complex to build and understand. To abstract the complexity in building/packaging the application based on the above structure spring boot has provided maven and gradle plugins

1. spring-boot-maven-plugin (maven plugin)
2. org.springframework.boot plugin (gradle plugin)

while setting up spring boot project along with starter dependencies we are adding plugin in pom.xml, due to this reason.

pom.xml

```
<project>
```

```
  <modelVersion>4.0.0</modelVersion>
  <!--<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-
parent</artifactId>
    <version>2.5.4</version>
    <relativePath/>
  </parent>-->
  <groupId>boot</groupId>
  <artifactId>expexec</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <dependencyManagement>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-
parent</artifactId>
      <version>2.5.4</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-
plugin</artifactId>
        <version>2.5.4</version>
        <executables>
          <executable>
            <phase>package</phase>
            <goals>
              <goal>repackage</goal>
            </goals>
          </executable>
        </executables>
      </plugin>
    </plugins>
  </build>
</project>
```

The spring-boot-maven-plugin will be executed during the maven build under package phase of the default lifecycle. Now the boot plugin performs the below job

1. takes the application classes and places them within BOOT-INF/classes directory within the Jar

2. reads all the dependencies in pom.xml declared and copies them into BOOT-INF/lib directory

3. copies Launcher.class, JarLauncher.class and WarLauncher.class into jar package directly

4. it reads the packaging type of our application, if it is a Jar then in manifest.mf it populates Main-Class as JarLauncher, and if packaging type is War then it populates Main-Class: as WarLauncher

5. Along with that boot plugin will identifies the Application class of our project based on @SpringBootApplication annotated class and write its fqn class as Start-Class in manifest.mf

6. builds the whole contents into Boot jar

1. Introduction to Spring Boot

2. Features of Spring Boot

2.1 AutoConfiguration

2.2 Starter Dependencies

2.3 Actuator Endpoints

2.4 DevTools

2.5 Embedded Containers

2.6 Spring Boot CLI

3. How to setup a Spring Boot Application

3.1 What are starter dependencies how do they work and explored creating our own starter dependency

3.2 Multi-Module Maven Project

3.3 There are 3 ways are there in Setting up Spring Boot Application in maven

3.3.1 directly create a maven project by adding starter dependencies and plugin configuration

3.3.2 declare spring-boot-starter-parent as a parent project in pom.xml

3.3.3 spring-boot-starter-parent as dependency import

3.4 Spring Boot project using Gradle

4. How does spring boot application works

4.1 what is @SpringBootApplication annotation, why do we use it

4.2 How does SpringApplication.run(..) works

5. What is SpringBoot Executable Jar, explored packaging and execution flow

5.1 spring-boot-maven-plugin, how does it helps

What will happens when we call SpringApplication.run(Config.class, args)?

1. Creates an empty environment object

2. detects the external configuration of our application and loads into the environment object

3. print boot banner

4. identifies the type of WebApplicationType

4.1 if spring webmvc jars are found under the classpath, treats the WebApplicationType = WEB and it creates AnnotationConfigServletWebServerApplicationContext

4.2 if spring webflux jars are found under the classpath, treats the WebApplicationType = REACTIVE and it creates AnnotationConfigReactiveWebServerApplicationContext

4.3 if none of the above then treats the WebApplicationType = NONE and creates the AnnotationConfigApplicationContext

5. Instantiates the spring factories and registers with ioc container

6. executes ApplicationContextInitializer

7. prepareContext

8. refreshContext

9. instantiates and executes CommandLineRunners and ApplicationRunners and returns there reference of ioc container it has created

10. during the above stages of starting up the application, it publishes various different types of events and invokes the listener to perform operation

How does SpringApplication.run(Config.class, args) detects the external configuration, what are the sources from which it reads and loads the external configuration of our application into environment object?

Spring Boot has provided several ways of detecting and loading the external configuration into environment object, through which we can configure our application components and framework components as well. So that it helps us in avoiding harding the application values and makes us easy to switch from one env to another env.

These are the places where it detects and loads the external configuration into our application environment object

1. if we are using devtools module as part of our application, then it looks for a file under user home directory with name spring-boot-devtools.properties and loads that configuration into env object

2. we can create an environment variable called SPRING_APPLICATION_JSON containing value as json text of keys and values. The SpringApplication.run() method parses the json text and loads them into environment object as properties

```
set SPRING_APPLICATION_JSON={"fuelType":"Petrol",
"capacity": 20}
```

3. All the init parameters we configure at ServletConfig/ServletContext level will be read and loaded into the environment object (only applicable for web application type)

```
  <context-param>
    <param-name>datasource.jdbc.url</param-name>
    <param-
value>jdbc:mysql://localhost:3306/db</param-value>
  </context-param>
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServl
et</servlet-class>
    <init-param>
      <param-name>namespace</param-name>
```

```
      <param-value>webconfig</param-value>
    </init-param>
  </servlet>
```

4. detects and loads all the environment variables of our system into environment object

5. Jvm System Properties

6. RandomValuePropertySource that we use as part of properties file using ${random.int} in application.properties will be generated and loaded into env object

7. it loads the application.properties and application.yaml files under the below locations

  7.1 under the project directory

  7.2 under the project/config directory

  7.3 then looks under the classpath of our application (resources directory)

  7.4 looks under the classpath within the sub-directory config (resources/config directory)

How does SpringApplication.run(Config.class, args) will detects and loads the external configuration of our application into environment object?

The SpringApplication.run(Config.class, args) will detects and loads the external configuration of our application from various different places as described below.

1. if we have enabled devtools module, the it looks for an file under user home directory called "spring-boot-devtools.properties" and loads into the env object

2. it looks for an environment variable with name SPRING_APPLICATION_JSON containing a Json String of Key/Value, if found it parses the Json key and value and loads into env object

3. it loads the init parameters of both ServletConfig level and ServletContext level in a typical web application

4. env variables

5. jvm system properties

6. RandomValueSourceProperty we declared in application.properties

7. detects and loads application.properties | application.yml from any of the below locations

7.1 directly under project directory

7.2 under config sub-directory of the project

7.3 directly from classpath of our application (resources directory)

7.4 config sub-directory of the classpath of our application (resources/config directory)

```
set SPRING_APPLICATION_JSON={"signalType": "beta",
"frequency": 91.1, "bandwidth": 200,
"radioType":"digital", "manufacturer": "philips",
"price": 5000}
```

How to customize the Spring Boot Application?

SpringApplication.run(Config.class, args) method performs lot of startup activities during the time bring up the application like creates an empty enviornment, detects and loads the external configuration into environment object, print banner and creates an ioc container etc. In addition to the

above activities being performed, we want customize or add additional activities in bring up the application. For eg..

1. along with reading the default configuration files during the startup, we want to read/load our own application configuration into env object with which we want to create ioc container

2. we want to customize the way the banner is printed

3. Instead of creating one ioc container we want to create nested ioc containers

We can customize all the activities that takes place during the time of starting up the spring boot application which are nothing but customizing the spring boot application

There are 2 approaches are available in customizing the spring boot application

1. configuration approach using which we can customize the spring boot application

2. programmatically we can customize the startup activites of the spring boot application

Not all the customizations can be performed through configuration approach, few of them can be achieved only through programmatic approach, due to which there are 2 ways of customizing the spring boot application are there

Customizing through configuration approach

---

one of the customization we can perform through configuration approach is changing or turnoff the spring boot banner. There are multiple configuration options are available to achieve this.

1. SpringApplication.run(..) looks for a file with name banner.txt under the classpath of the application, if found, it stops printing the default banner and prints the contents of the file as banner

2. Instead of writing banner.txt we can write banner.image and image also will be detected and printed. (if both are presented it prints both)

3. we can write a property in application.properties|application.yaml through which we can control the banner generation in spring boot application

spring.main.banner-mode: off | on | console

4. we can change the banner.txt filename or its location by specifying the below property in application.properties file

spring.banner.location= [location to banner file]

How to customize the Spring Boot Application?

There are lot of activities being performed by SpringApplication class during the startup of our Spring Boot Application, we can modify or add

additional activities in starting up a spring boot application which is nothing but customizing the Spring Boot Application.

There are 2 ways we can customize Spring Boot Application

1. Configuration Approach

2. Programmatic Customization of Spring Boot Application

#1. Customize the Spring Boot Application through Configuration approach

1.1 we can change the banner

- we can write banner.txt or banner.(img) under classpath of our application, which will picked up automatically by SpringApplication and will render

- we can turn off the banner through spring.main.banner-mode: off/console

- we can change the banner location spring.banner.location: location

1.1 spring.main.lazyInitialization = true

will delays the instantiation of the bean definitions within the ioc container

#2. Programmatic Customization of Spring Boot Application.

Fluent build api of Spring Boot

The SpringApplication class is performing the startup activities during the time of starting the spring boot application, if we want to modify or add additional activities we need to configure/modify the

SpringApplication classes asking him to perform the activities.

How to customize or modify SpringApplication class asking him to perform activities differently, that is where Fluent Builder api comes into picture.

Fluent Builder Api is a design pattern

What is Factory class, why do we use it?

Factories are the classes used for creating the object of another class. there are 2 reasons why we go for factory

1. To abstract the complexity in creating the object of another class
2. To hide the implementation class name of another class from others

```
class A {
  int i;
  int j;
  int k;
}
```

We want to create the object of class A, through the help of Factory so we can create AFactory class which is responsible for creating the object A.

```
class AFactory {
  public static A createA() {
    A a = new A();
    return a;
  }
}
```

In the above code AFactory creates the object A as empty, usually Factories doesnt allows us to populate data into objects while creating. To solve the problem Builder Pattern is being used.

```
class ABuilder {
  int i;
  int j;
  int k;

  ABuilder(int i) { // i became mandatory
    this.i = i;
  }
  public ABuilder i(int i) {
    this.i = i;
    return this;
  }
  public int i() {
```

```
    return i;
  }
  public ABuilder j(int j) {
    this.j = j;
    return this;
  }
  public int j() {
    return j;
  }
  public ABuilder k(int k) {
    this.k = k;
    return this;
  }
  public int k() {
    return k;
  }

  public A build() {
    A a = new A();
    a.setI(i);
    a.setJ(j);
    a.setK(k);
    return a;
  }
}
```

```
A a = new ABuilder(19).j(20).build();
```

The Spring Boot application will start execution when we call static method run on class SpringApplication, SpringApplication.run(Config.class, args). This method performs various different startup activities based on default configuration.

If we want to customize the startup activities of our application, we need to create SpringApplication class with our own configuration values using which we need to run the application.

How to create the object of another class by populating the data into it?

Builder design pattern has to be used.

So Spring Boot developers has provided an Builder class called **SpringApplicationBuilder**, into which populate the configuration values or customizations with which we want to create the object of SpringApplication class and then call run() method.

1. How to turn off the banner through programmatic approach

```
@SpringBootApplication
class BootBannerApplication {
  public static void main(String[] args) {
```

```
    ApplicationContext context =
SpringApplication.run(BootBannerApplication.class,
args);

    // this will kick start the application with
default settings or configuration which we dont want


    SpringApplicationBuilder builder = new
SpringApplicationBuilder(BootBannerApplication.class)
;

    builder.bannerMode(Banner.Mode.OFF);

    SpringApplication springApplication =
builder.build();


    ApplicationContext context =
springApplication.run(args);

  }

}
```

What is Fluent Builder api, how to customize Spring Boot Application using Fluent Builder api?

The SpringApplication.run() static method performs various different startup activities while starting the Spring Boot Application, if we want to customize the startup activities in bringing up the Spring boot Application, we need to create the object of SpringApplication class by populating our own settings and configuration options with which we need to run the Spring Boot Application.

Spring Boot has provided SpringApplicationBuilder a fluent builder api class through which we can

customize in creating the object of SpringApplication and we can call instance method run() on it.

There are lot of configuration parameters/settings are available in creating the object of SpringApplication, So to ease the instantiation process in creating the object of SpringApplication class, Spring Boot has provided SpringApplicationBuilder.

```
abstract class Bike {
  String manufacturer;
  String rtaRegNo;
  String chasisNo;
  String engineNo;
  String color;
  String fuelType;
  int mileage;
  double price;
  // accessors
}
class KTMBike extends Bike {
  String suspentionType;
}
class RoyalEnfieldBike extends Bike {
  String breakingType;
}
```

In general we use Factories for creating the object of another class, there are 2 reasons we go for factory

1. we can hide implementation class of another class from our class, so that we can achieve loose coupling

2. we can abstract the complexity in creating the object

```
class BikeFactory {
  public static Bike newBike(String type) {
    if(type.equals("ktm")) {
      bike = new KTMBike();
    }else if(type.equals("royalenfield")) {
      bike = new RoyalEnfieldBike();
    }
    return bike;
  }
}

class Ride {
  public void drive() {
    Bike bike = BikeFactory.newBike("ktm");
  }
}
```

Here Ride class dont know the actual classname of KTHBike, without using the classname it is able to get the object of another class. which makes my class loosely coupled.

Here the object of the bike has been created as empty, but we want bike object to be created with data, which we cannot achieve it easily using Factory.

In order to create the object of another class with data being populated we need to use Builder Pattern.

1. loosely coupling

2. abstract complexity

3. helps in quickly creating the object by populating data

Now use a builder for creating the object of another class, if we want builder to create the object of another class with data, give the data to the builder and ask him to build the object of another class.

From the above we can understand build has instance factory method build() which will creates and returns the object of other class.

```
class KTMBikeBuilder {

  String manufacturer;

  String rtaRegNo;

  String chasisNo;

  String engineNo;
```

```
  String color;
  String fuelType;
  int mileage;
  double price;
  String suspentionType;


  public KTMBikeBuilder(String chasisNo, String
engineNo) {
    this.chasisNo = chasisNo;
    this.engineNo = engineNo;
  }
  // fluent api methods
  public KTMBikeBuilder manufacturer(String
manufacturer) {
    this.manufacturer = manufacturer;
    return this;
  }
  public String manufacturer() {
    return this.manufacturer;
  }
  public KTMBikeBuilder color(String color) {
    this.color = color;
    return this;
  }
  public String color() {
```

```
    return this.color;
  }
  public Bike build() {
    Bike bike = new Bike();
    bike.setChasisNo(this.chasisNo);
    bike.setEngineNo(this.engineNo);
    bike.setColor(color);
    bike.setManufacturer(manufacturer);
    return bike;
  }
}
```

Now to create the object of Bike, create the object of KTMBikeBuilder and populate data into builder object, then call build() that will create the object of KTMBike with data that we gave to the builder already.

```
Bike bike = new KTMBikeBuilder("e039833",
"ch39383").manufacturer("ktm").color("orange").fuelTy
pe("petrol").build();
```

Similar to the above, the Spring boot has provided SpringApplicationBuilder in which populate the settings or configuration options with which we want to create the object of SpringApplication class. So that we can customize the startup activities.

```
SpringApplicationBuilder builder = new
SpringApplicationBuilder(Config.class);

builder.bannerMode(Banner.Mode.OFF);

SpringApplication springApplication =
builder.build();
```

The above line of code has created SpringApplication class object with configuration/settings we passed to the builder.

```
ApplicationContext context =
springApplication.run(args); // instance method run()
```

What is Nested Bean Factories, why do we use it?

To keep the bean definition independent within their ioc containers and use them together we can go for nested bean factories.

For eg.. while working with webmvc we place business and persistence tier components in parent ioc container, and webmvc components in child ioc container nesting the parent. So that if we remove webmvc we can independently use business and persistence tier components withour application without any impact.

We can nest/place one ioc container inside another ioc container, so that we can manage dependencies between the bean definitions across the ioc containers. While working with Nested ioc container

we have 2 ioc containers, one is parent container another one is child ioc container.

First we should create parent ioc container, using which we create child ioc container by keeping parent inside it. always the child container bean definitions can refer parent container, but parent cannot refer child container bean definitions.

In Spring Boot application SpringApplication.run() method is creating the ioc container by taking JavaConfig.class. But we want to 2 nested ioc containers one as parent another as child with 2 JavaConfig classes. This can be achieved through Fluent Builder api.

```
// no source code
class Rocket {
  public void ignite() {
    sop("launching rocket");
  }
}
// no source code
class Launcher {
  private Rocket rocket;

  public void launch() {
    rocket.ignite();
```

```
  }
}

@Configuration
class ParentConfig {
  @Bean
  public Rocket rocket() {
    return new Rocket();
  }
}

@Configuration
class ChildConfig {
  @Bean
  public Launcher launcher(Rocket rocket) {
    Launcher launcher = new Launcher();
    launcher.setRocket(rocket);
    return lanucher;
  }
}

@SpringBootApplication
class NestedBootApplication {
  public static void main(String[] args) {
```

```
    SpringApplicationBuilder builder = new
SpringApplicationBuilder();


builder.parent(ParentConfig.class).sources(ChildConfi
g.class);

    SpringApplication springApplication =
builder.build();

    ApplicationContext context =
springApplication.run(args);



  }

}
```

Create the object of SpringApplicationBuilder and pass him the parent configuration class and child configuration classes, asking him to populate these while creating SpringApplication class object.

When we call build() method on builder it will populate both configuration classes into SpringApplication, so that when we call run() method, it will instantiate 2 ioc containers one as parent and another as child container nesting the parent.

Note:-

While working with Nested ioc containers, the same environment object will be used in creating both the ioc containers.

What is event-based processing model, why do we use it?

There are multiple programming models are available in programming world.

1. Linear programming model = all the lines of code within your project executes sequentially right from the top to the bottom which is called "Linear execution model" as well.This is the default programming model for most of the programming languages

2. Multi-Threaded model = The program will be divided into multiple paths of execution, where all the paths of the program are executed simulateneously in a inter-leaved fashion which is called Multi-Threading.

The programming languages has to provide additional apis in supporting the Multi-Threaded execution

3. Interface based programming model

The classes will talk to another class by holding the reference of other. The reference be an Concreate reference or a Interface type. by holding the reference of another class if we talk to another class always there will a level of coupling it doesnt matter what type of reference you hold. because you are calling the methods of another class, if the method signature has been changed your class will be effected.

4. Event-Driven programming model.

Few things about event driven programming model

1. no class holds the reference of another class in communicating with the other class, so they dont see each other interfaces.

2. The only means of communicating with another class is by publishing event. both parties (source/handler) should agree upon event rather than interfaces to communicate

3. in event based programming always the communication takes place in one-direction only.

4. both the parties are disconnected from each other (asynchronous programming)

From the above we achieve loosely coupled asynchronous based programming through event-driven processing model.

Configuration Properties in Spring Boot

```
@Component
@ConfigurationProperties(prefix="book")
class Book {
  //@Value("${author}")
  String author;
  //@Value("${isbn}")
  String isbn;
  //@Value("${title}")
  String title;
  //@Value("${publisher}")
```

```
  String publisher
  //@Value("${version}")
  int version;
  //@Value("${price}")
  double price;
  // acessors
  // toString
}

@Component
class Author {
  String name;
  int age;
  String qualification;
}
```

application.properties

```
-----------------------
book.author=Chethan Bhagath
book.isbn=32938
book.title=Five point someone
book.publisher=rupa publication
book.version=1
book.price=230
```

application.yml

```
book:
  author: chethan bhagath
  isbn: 3938
  title: fivepoint someone
  publisher: rupa publisher
  version: 1
  price: 243
```

```
@SpringBootApplication
@EnableConfigurationProperties
public class ConfigurationPropsApplication {
  public static void main(String[] args) {
    ApplicationContext context =
SpringApplication.run(ConfigurationPropsApplication.c
lass, args);
    Book book = context.getBean(Book.class);
    sop(book);
  }
}
```

In the above example in order to inject the dependent values into Book bean definition we have to write @Value() annotation on each property of the class, so the more number of attributes it takes more time to write the annotations in injecting the values.

To overcome this, spring boot has introduced **ConfigurationPropertiesBeanPostProcessor.** The ConfigurationPropertiesBeanPostProcessor should be declared as bean definition, so that it would be automatically registered with ioc container and will be invoked for each bean definition of the ioc container before intialization and after initialization

So to that we need to write a bean definition method in java configuration class, instead SpringBoot has provided an annotation called **@EnableConfigurationProperties**

BeanPostProcessor

if we want to perform common post processing logic for each and every bean definition within our ioc container then use BeanPostProcessor.

The ioc container after instantiating a bean definition, before performing initialization it calls bean post processor to perform post processing logic, then after the lifecycle operation on bean definition has been completed, before placing the bean definition within ioc container again the ioc container calls bean postprocessor after initialization to perform post processing logic.

In order for ioc containers to call BeanPostProcessor we need to register the bean post processor with ioc container

How to register beanpostprocessor with ioc container?

if we are creating ioc container with ApplicationContext, then declare them as bean definitions, so those are automatically picked during the instantiation ioc container and will be registered.

In case of ConfigurationProperitesBeanPostProcessor how to register it with ioc container?

Just write an annotation **@EnableConfigurationProperties**, with which it register ConfigurationPropertiesBeanPostProcessor with ioc container.

---

how many programming methodologies are there and what are those?

There are 4 programming methodologies are there broadly

1. **Linear programming model**
2. **Multi-Threaded programming model**
3. **Interface based programming**
4. **Event-Driven Asynchronous programming**

There are plenty of advantages we get when we use event-driven programming model.

4.1 the components of our application are completely loosely, no 2 classes knows or sees the interfaces of each other, since they communicate by publishing the events.

4.2 disconnect communication model, where callee will not be blocked until caller complete the execution, nothing but asynchronous programming

4.3 in this programming communication always takes place in uni-directional only

In event-driven programming model there are total 4 actors are there

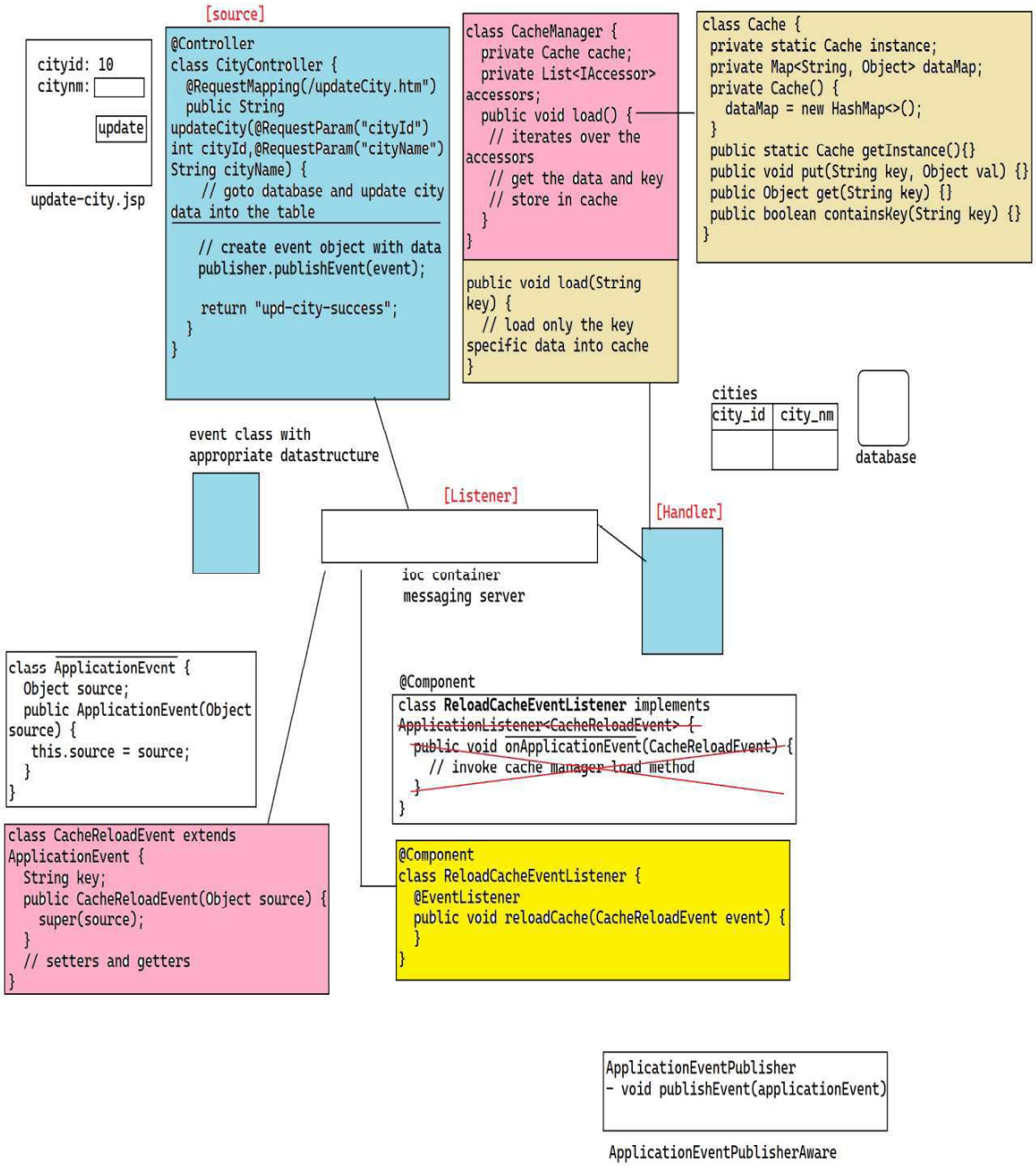
1. source = always a person who wants some operation to be performed, but source dont know how to perform the operation so he should ask someone to perform the operation.

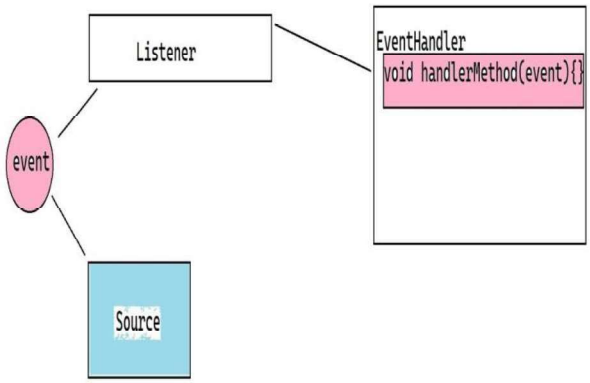
Source will publish an event asking someone to perform the operation.

2. event = event is an object encapsulated with the information about the operation/action to be performed, the source who published the event and any other information wrapped inside which is required for performing the operation.

3. listener = Listener is always a person who listenes for an an event from Source, upon receiving the event it checks type of event and the source who has published and based on which it identifies an appropriate handler who can handle and process the event. invokes the handler method by passing event object asking to perform the operation.

4. handler = Handler is one who knows how to perform the operation for a specific event. Handler contains the event handler methods and is registered with listener.





Source is always the person who want to perform an action or an operation within our application, but Source dont know how to perform the operation. if Source by himself cannot perform the operation, atleast he has to ask someone to perform the operation by publishing an event.
"Source is always a person who triggers an event or publishes an event asking someone to perform operation"

Event is an object encapsulated with the action or operation, the source who generated the event and the data we want to pass to the other person who performs the operation. It contains 3 things
1. action
2. source
3. data to be used to perform operation

EventHandler:
is a component who knows how to perform operation based on the type of the event. An event handler may contain one handler method or multiple handler methods to perform operation.

Either the Source nor the Event will directly call EventHandler, because it becomes tightly coupled.

Listener:
Listener always listens for an event from the Source, once it has received the event, it will identify appropriate handle and its method and invokes it by passing event.

Why do we need to use Event Driven Programming?

What is event-driven programming, why do we need to use it?

event-driven programming is a message oriented asynchronous communication model, where the callee and the caller dont see the interfaces of each other, but they communicate with each other by exchanging the messages/events between them.

it is an one way communication model.

1. loosely coupled

2. uni-directional communication

3. asynchronous

Java has provided an api to implement event-based programming which is observer api, where it is too low-level api where programmer has to write lot of code in implementing an application. In Jee platform jms api is available to implement event driven programming, which is an enterprise level messaging solution which becomes heavy weight for simple application requirements.

Instead of using observer api which is too low-level or jms api which is heavy weight spring has provided an a better support for event-driven programming within its core container which Spring Event-Driven model.

How to work with event-driven programming in Spring Framework?

Event:-

Spring has provided an standard base class to represent an event which is ApplicationEvent class. for all the events to identify the source who is publishing the event we Source object, So ApplicationEvent class has been declared with Object source as an attribute to populate the information about the source.

We can write our own event class extends ApplicationEvent and we declare additional attributes to populate and carry the data to Handler/Listener

```
class ReloadCacheEvent extends ApplicationEvent {
  String key;
  public ReloadCacheEvent(Object source, String key)
{
    super(source);
    this.key = key;
  }
  // accessors
}
```

When the source has published the event, the ioc container acts as an Listener in receiving the events and invoking the handler.

How to write an Handler class?

Since ioc container has to invoke the handler we should write the Handler class by implementing an interface called ApplicationListener or annotate the handler method with @EventListener and declare the event we want to recieve as a input to perform the operation.

```
@Component (register with ioc container)

class ReloadCacheEventApplicationListener implements
ApplicationListener<ReloadCacheEvent> {

  public void onApplicationEvent(ReloadCacheEvent
event) {}

}


(or)


@Component (register with ioc container)

class ReloadCacheEventApplicationListener {

  @EventListener

  public void onReloadEvent(ReloadCacheEvent event)
{}

}
```

```
@Component

class CityController /*implements
ApplicationEventPublisherAware*/ {

  @Autowired

  ApplicationEventPublisher publisher;
```

```
  public void updateCity(int cityId, String cityName)
{

    // update the data in database by invoking
service class

    ReloadCacheEvent event = new
ReloadCacheEvent(this, "cities");

    publisher.publishEvent(event);

  }


  //ioc container

  public void
setApplicationEventPublisher(ApplicationEventPublishe
r publisher) {

    this.publisher = publisher;

  }

}
```

To publish the events to the ioc container, Spring has provided ApplicationEventPublisher which is an internal object of the ioc container, in order to get the reference of internal object of ioc container we need to use Aware interface which is ApplicationEventPublisherAware.

ApplicationEvents and Listeners in Spring Boot

When we launch spring boot application by executing the below line of code.

```
SpringApplication.run(Config.class, args);
```

During the time of starting up the Spring boot application using SpringApplication class it performs various different activities in bring up the spring boot application like

1. create empty environment object (e1)

2. detect the external configuration of our application and load into environment object (e2)

3. print banner (e3)

4. detect the type of the application based on the classes under the classpath and instantiate the appropriate ioc container

   4.1 if webmvc jar is found under classpath treat WebApplicationType = WEB and instantiate AnnotationConfigServletWebServerApplicationContext

   4.2 if webflux jar is found under classpath treat WebApplicationType = REACTIVE and instantiate AnnotationConfigReativeWebServerAppliationContext

   4.3 otherwise treat WebApplicationType = NONE and instantiate AnnotationConfigApplicationContext (e4)

5. Instantiate and register Spring Factories with ioc container (e5)

etc

```
ApplicationContext context =
SpringApplication.run(Config.class, args);
```

Spring boot event handling

How to customize or perform additional activities during the startup of Spring Boot application?

SpringApplication class during the time of bringing up the Spring Boot application, it performs various different activities. In order to allow us in perform additional activities during the startup SpringApplication class publishes different types of events indicating the corresponding stage of boot activity.

Programmer can write Listener to handle appropriate event and perform additional operation which is nothing but Spring boot events and listeners.

There are 6 different types of events published by SpringApplication class during the startup

1. ApplicationStartingEvent

after calling SpringApplication.run(Config.class, args), before performing any of the activities, indicating the application is about to start it publishes ApplicationStartingEvent.

2. ApplicationEnvironmentPreparedEvent

after creating the environment object, after identifying and loading the application configuration into env object, the SpringApplication class publishes ApplicationEnvironmentPreparedEvent.

3. ApplicationStartedEvent

after ioc container has been created, after loading the bean definitions into ioc container, before instantiating the objects for the bean definition, it publishes ApplicationStartedEvent

4. ApplicationPreparedEvent

after the objects for the bean definitions are instantiated before running the CommandLineRunners and ApplicationRunners, it publishes ApplicationPreparedEvent

5. ApplicationReadyEvent

after executing the commandlinerunners and ApplicationRunners successfully before returning the object of ioc container to the User/Application, it publishes ApplicationReadyEvent

6. ApplicationFailedEvent

during any of the stages of starting up the application, it there is a failure encountered, indicating the reason for the failure the ApplicationFailedEvent will be published.

Now to perform some additional activities during any of the above stages, we can write our own listener in handling those events, and register the Listener with SpringApplication asking it to call the Listener for an appropriate event.

We can register the Listener to SpringApplication using FluentBuilder api.

```
class BootStartupActivityListener {

  @EventListener

  public void
onApplicationStartingEvent(ApplicationStartingEvent
event) {

    // write the logic for performing processing

  }

}


@SpringBootApplication

class BootEventListenerApplication {

  public static void main(String[] args) {

    //ApplicationContext context =
SpringApplication.run(BootEventListenerApplication.cl
ass, args);

    SpringApplication springApplication = new
SpringApplicationBuilder(BootEventListenerApplication
.class).listener(new
BootStartupActivityListener()).build();


    ApplicationContext context =
springApplication.run(args);

  }

}
```

Application Properties

How to work with configuration properties in Spring Boot Application?

```
@Component
@ConfigurationProperties
class Contract {
  int contractNo;
  String contractTitle;
  String description;
  double budget;
  // accessors
}
```

application.properties

```
contractNo=939
contractTitle=Drinage Repair
description=underground drinage
budget=100000
```

```
@SpringBootApplication
class ConfigPropsApplication {
```

```
  public static void main(String args[]) {
    ApplicationContext context =
SpringApplication.run(ConfigPropsApplication.class,
args);


  }
}
```

In the above example, SpringApplication itself will detect and read application.properties/yaml/yml file from projectdir or projectdir/config or classpath or classpath/config location and loads into the env object during the startup of Spring Boot Application. we dont have to use @PropertySource annotation to instruct ioc container to read and load application.properties into env object.

Instead of using a properties file Spring supports even an yaml used as an configuration file to define application configuration.

YAML stands for Yet Another markup language and renamed to An't markup language. Using YAML we define application configuration information in an effective way than using properties/xml/json approach.

The other configuration formats are verbose when compared with yaml, and it is short and easy to understand.

properties = problem with properties is it can represent configuration in keys and values, we cannot

represent multiple valued information like lists or arrays.

xml/json = too verbosed, too much amount of information describing the data should be written around the data.

xml:

```
<contract>
  <contractno>234</contractno>
  <contract-title>Drinage Work</contract-title>
  <budget>393</budget>
</contract>
```

json:

```
{
  "contractno": 234,
  "contract-title": "drinage work",
  "budget": 393
}
```

instead of using xml/json we can represent using yaml in less verbosed format

How to represent the data in yaml?

#1

An yaml file can contain multiple documents of data inside it.

a start the document is indicated with three hypens (---)

an end of the document indicated with three dots (...)

application.yaml

```
---

amount: 3903

name: perk

...

---

contractno: 393

title: drinage contract

...
```

#2 in yaml we can represent data in 2 data types

1. list

2. dictionaries

and intendation plays an crucial role in identifying the data.

2.2 dictionaries

dictionaries are usually used for represent object of properties as key/value pair

```
contractno: 12
contractTitle: drinage work
budget: 234
```

yaml can recognize String, numbers and booleans automatically.

we dont need to embedded string in double quotes.

2.3 list

we can represent list of values using (-) hypen

```
- dal
- rice
- jeera
- turmeric
```

---

person.json

```
{
  "name": "adrew",
  "age": 23,
```

```
  "gender": "male",
  "mobile": "9283832",
  "emailAddress": "adrew@gmail.com",
  "address": {
    "addressLine1": "221 Bakers Street",
    "city": "Hyderabad"
    "state": "telangana",
    "pincode": 30393,
    "country": "India"
  },
  "qualification": ["mca", "mba", "ms"]
}
```

person.yaml | yml

```
---
name: adrew
age: 23
gender: male
mobile: 9283983
emailAddress: adrew@gmail.com
address:
  addressLine1: 221 bakers street
  city: hyderabad
  state: telangana
  pincode: 39349
```

```
  country: India
qualification:
  - mca
  - mba
  - ms
...
```

players.json

```
-------------
[
    {
      "name", "Dhoni",
      "age": 34,
      "rank": 7
      "type": "wicket keeper"
    },
    {
      "name": "virat",
      "age": 29,
      "rank": 2,
      "type": "captain"
    }
]
```

players.yaml

```
-----------------
---
- name: dhoni
  age: 34
  rank: 7
  type: wicket keeper
- name: virat
  age: 29
  rank: 2
  type: captain
...
```

tourpakcages.json

```
{
  "packageno": 93839,
  "packagename": "Shimla and Manali",
  "days": "5 nights and 4 days",
  "places": [
    "shimla",
    "manali",
    "chandighar"
  ],
  "travelmode":[
```

```
    "plane",
    "car"
  ],
  "permitrequired": false,
  "accomodationType": "3 star"
  "amount": 90393
}
```

tourpackages.yaml

```
-------------------
---
packageno: 93983
packagename: shimla and manali
days: 5 nights and 4 days
places:
  - shimla
  - manali
  - chandighar
travelmode:
  - plane
  - car
permitrequired: false
accomodationType: 3 star
amount: 90393
```

The above eg we discussed can be replaced with yaml configuration as well below.

```
@Component
@ConfigurationProperties
class Contract {
  int contractNo;
  String contractTitle;
  String description;
  double budget;
  // accessors
}
```

application.yaml

```
contractNo: 939
contractTitle: Drinage Repair
description: underground drinage
budget: 100000
```

```
@SpringBootApplication
class ConfigPropsApplication {
  public static void main(String args[]) {
```

```
    ApplicationContext context =
SpringApplication.run(ConfigPropsApplication.class,
args);


  }

}
```

The SpringApplication class will detect, read and load the application.yaml into the environment object during the bootup of the spring boot application.

How to work with configuration properties in Spring Boot application?

```
@Component

@ConfigurationProperties(prefix="contract")

class Contract {

  int contractNo;

  String contractName;

  String description;

  double budget;

  // accessor

}
```

application.yml

```
@SpringBootApplication
class CPApplication {
  public static void main(String[] args) {
    ApplicationContext context =
SpringApplication.run(CPApplication.class, args);
    Contract contract =
context.getBean(Contract.class);
    sop(contract);
  }
}
```

The **application.properties|yaml** is an standard spring boot configuration in which we are going to configuration spring boot autoconfiguration specific properties in that configuration file. We can even write our own class specific properties in the application.properties|yaml, but usually best practise is to separate autoconfiguration properties from our own component configuration.

By writing auto configuration related properties and our own component configurations separetely we can quickly identify the configuration being used in our application.

```
app-global.properties
contract.contractNo=303
contract.contractName=Road Contract
```

```
contract.description=NH35 Road works

contract.budget=393833


@PropertySource("classpath:app-global.properties")

@SpringBootApplication

class CPApplication {

  public static void main(String[] args) {

    ApplicationContext context =
SpringApplication.run(CPApplication.class, args);

    Contract contract =
context.getBean(Contract.class);

    sop(contract);

  }

}
```

To load our own custom properties file in on Spring Boot Application class we can write @PropertySource annotation to load the property file values into environment object.

```
app-global.yaml

contract:

  contractNo: 3939

  contractName: Road Contract

  description: NH35 Road works

  budget: 393849
```

We cannot use **@PropertySource annotation for load yaml file** into environment object. PropertySource annotation can only load simple key/value pairs only into environment, it doesnt support yaml file.

How to use our own custom yaml file in configuring the our component configuration and load into env of the ioc container?

There are 2 ways of working with our own custom yaml file being loaded into env object ioc container

1. YamlPropertySourceLoader
2. YamlFactoryBean

#1. YamlPropertySourceLoader

YamlPropertySourceLoader will not load yaml file into environment object. using YamlPropertySourceLoader we load an yaml file and convert into PropertySource object and returns to us. There after we need load PropertySource into env object

```
YamlPropertySourceLoader yamlPropertySourceLoader =
new YamlPropertySourceLoader();

List<PropertySource> propertySources =
yamlPropertySourceLoader.load(context.getResource("cl
asspath:app-global.yaml"));
```

Now we need load the PropertySource objects into environment object. after creating the ioc container before instantiating the bean definitions we need

load our custom yaml file into environment object of the ioc container.

SpringApplication class, invokes ApplicationContextInitializer to help in perform initialization activity ontop of ioc container post creation of ioc container before instantiating the bean definitions.

Now we can make use of ApplicationContextInitializer to perform initializing logic on top ioc container before it begins instantiating the bean definitions.

```
class YamlConfigApplicationContextInitializer
implements ApplicationContextInitializer {

  public void
initialize(ConfigurableApplicationContext context) {


    YamlPropertySourceLoader yamlPropertySourceLoader
= new YamlPropertySourceLoader();

    List<PropertySource> propertySources =
yamlPropertySourceLoader.load(context.getResource("cl
asspath:app-global.yaml"));


    ConfigurableEnvironment env =
context.getEnvironment();

    for(PropertySource ps: propertySources) {

      env.getPropertySources().addLast(ps);
```

```
    }
  }
}



@SpringBootApplication
class CPApplication {
  public static void main(String[] args) {
    YamlConfigApplicationContextInitializer
initializer = new
YamlConfigApplicationContextInitializer();


    SpringApplication springApplication = new
SpringApplicationBuilder(CPApplication.class).initial
izers(initializer).build();
    ApplicationContext context =
springApplication.run(args);


  }
}
application.properties|yaml = SpringApplication
@PropertySource = ioc container
custom yaml = ApplicationContextInitializer |
ApplicationEnvironmentPreparedEvent
```

In Spring Boot configuration file **(application.properties/yaml)** we need to configure

spring boot auto configuration properties, so that those can be used by autoconfiguration classes to configure framework components as bean definitions. Even though we can configure our application components configuration in standard spring boot configuration files, it is not recommended to be written in that file, because the auto configuration properties and application component configurations will be inter-mixed and difficult to understand the configuration file.

So always it is recommended to place our component configuration in separate configuration file. If we use an custom yaml configuration file, then how to load it into env object of ioc container?

We cannot use **@PropertySource annotation** to load custom yaml file into env object, we need to use any one of the 2 approaches.

1. YamlPropertySourceLoader
2. YamlFactoryBean

#1. YamlPropertySourceLoader

YamlPropertySourceLoader loads the contents of an Yaml file and converts into List<PropertySource> objects and returns to us.

Now we should load these property sources into env object of the ioc container, how to do it?

After SpringApplication class has created the ioc container, before it calls prepare and refresh context to instantiate the bean definition objects of the ioc container we should perform initialization on ioc container by loading property source objects into

env of the ioc container, this can be done by taking the help of ApplicationContextInitializer

SpringApplication class upon creating the ioc container, it looks for an initializer being registered, if available it calls initialize(ConfigurationApplicationContext) asking to perform initialization on the ioc container.

```
class YamlConfigApplicationContextInitializer
implements ApplicationContextInitializer {

  public void
initialize(ConfigurableApplicationContext context) {

    YamlPropertySourceLoader yamlPropertySourceLoader
= new YamlPropertySourceLoader();

    List<PropertySource> propertySources =
yamlPropertySourceLoader.load("app-global",
context.getResource("classpath:app-global.yaml"));

    ConfigurableEnvironment env =
context.getEnvironment();

    for(PropertySource propertySource :
propertySources) {


env.getPropertySources().addLast(propertySource);

    }

  }

}
```

```
SpringApplication springApplication = new
SpringApplicationBuilder(Config.class).initializers(n
```

ew
YamlConfigApplicationContextInitializer()).build();

#2. YamlFactoryBean

YamlFactoryBean help us in reading the yaml file and creates properties collection by populating yaml data and returns properties collection to us.

From the above we can understand Yaml files can be read in 2 ways

,

@PropertySource = it takes of loading an properties into environment object of the ioc container, we pass properties filename as an input, so that it reads the properties file and loads into env object.

Now we have Yaml file, if we can read the Yaml file using **YamlFactoryBean** and convert into properties collection and pass it as an input to @PropertySource then it takes care of loading the properties collection automatically into env object.

Now we need to tell @PropertySource annotation, execute the code i have written which gives you Properties collection, then load into env object.

That is where to be called upon by PropertySource, PropertySourceFactory was introduced in SpringBoot 2.5.4

```
@PropertySource(name="app-global",
value="classpath:app-global.yaml", factory=
YamlPropertySourceFactory.class)

@PropertySource(name="custom-props",
value="classpath:custom.yaml",
factory=YamlPropertySourceFactory.class)


class YamlPropertySourceFactory  implements
PropertySourceFactory {

  public PropertySource createPropertySource(String
name, EncodedResource resource) {

    YamlPropertiesFactoryBean
yamlPropertiesFactoryBean = null;


    yamlPropertiesFactoryBean = new
YamlPropertiesFactoryBean();


yamlPropertiesFactoryBean.setResources(resource.getRe
source());

    Properties properties =
yamlPropertiesFactoryBean.getObject();


    PropertiesPropertySource propertySource = new
PropertiesPropertySource(properties);

    return propertySource;

  }
```

}

How to use our custom yaml file for configuring the our own application components?

Spring boot internally uses snake yaml library for reading the Yaml files.

There are 2 ways of reading the Yaml file are there

1. YamlPropertySourceLoader = it loads the yml file and converts into List of PropertySource objects and returns to use.

2. YamlPropertiesFactoryBean = it reads the yaml file and converts into Properties collection and returns to us.

After reading the Yaml file using one of the above ways how to load into environment object of ioc container?

There are 2 ways of loading the PropertySource into env object.

1. ApplicationContextInitializer = SpringApplication class, after creating the ioc container before refreshing the ioc container, it calls ApplicationContextInitializer if registered to perform initialization logic on the ioc container.

2. PropertySourceFactory = PropertySourceFactory acts as an factory class for creating the object of PropertySource, which can be passed as an input to @PropertySource annotation, which takes care of calling the PropertySourceFactory and loads the PropertySource into environment object of ioc container.

What are profiles what is the purpose of it (@Profile)?

We can use profiles for switching from one environment to another environment easily.

```
@Component
class ConnectionManager {
  @Value("${db.driverClassname}")
  private String driverClassname;
  @Value("${db.url}")
  private String url;
  @Value("${db.username}")
  private String username;
  @Value("${db.password}")
  private String password;

  // accessors
}
```

```
db-dev.properties
--------------
db.driverClassname=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/db
db.username=root
db.password=root

db-test.properties
------------------
db.driverClassname=com.oracle.jdbc.Driver
db.url=jdbc:oracle:thin:@localhost:1521/xe
db.username=sys
db.password=welcome#123

@Configuration
@PropertySource("classpath:db-dev.properties")
@Profile("dev")
class DevConfig {

}
@Configuration
@PropertySource("classpath:db-test.properties")
@Profile("test")
class TestConfig {
```

```
}


@Configuration
@ComponentScan(basePackages={"com.profiles.beans"})
@Import({DevConfig.class, TestConfig.class})
class JavaConfig {


}



class Test {
  public static void main(String[] args) {
    ApplicationContext context = new
AnnotationConfigApplicationContext();
    context.getEnvironment().setProfiles("test");
    context.register(JavaConfig.class);
    context.refresh();


    ConnectionManager cm =
context.getBean(ConnectionManager.class);


  }
}
```

How to work with profiles in Spring Boot?

```
@Component
@ConfigurationProperties(prefix="db")
class ConnectionManager {
  private String driverClassname;
  private String url;
  private String username;
  private String password;

  // accessors
}
```

application.properties

```
-----------------------
spring.profiles.active=test
```

application-dev.properties

```
----------------------------
db.driverClassname=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/db
db.username=root
db.password=root
```

application-test.properties

```
db.driverClassname=com.oracle.jdbc.Driver

db.url=jdbc:oracle:thin:@localhost:1521/xe

db.username=sys

db.password=welcome#123
```

```
@SpringBootApplication

class BootProfileApplication {

  public static void main(String[] args) {

    ApplicationContext context =
SpringApplication.run(BootProfileApplication.class,
args);

    ConnectionManager cm =
context.getBean(ConnectionManager.class);


  }

}
```

SpringApplication class has been designed to support profiles by default. by default SpringApplication class reads application.properties. To support different profiles SpringApplication has been designed to application-[profile].properties when we set the profile automatically which is Spring Boot profile support.

How to work with profiles in spring boot with yaml?

application.yml

```
-----------------
---
spring:
  profiles:
    active: dev
...
---
spring:
  profiles: dev
db.driverClassname=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/db
db.username=root
db.password=root
...
---
spring:
  profiles: test
db.driverClassname=com.oracle.jdbc.Driver
db.url=jdbc:oracle:thin:@localhost:1521/xe
db.username=sys
db.password=welcome#123
...
---
```

```
spring:
  profiles: prod
db.driverClassname=com.oracle.jdbc.Driver
db.url=jdbc:oracle:thin:@localhost:1521/xe
db.username=sys
db.password=welcome#123
...
```

```
java -Dspring.profiles.active=dev -jar
BootProfileApplication.jar
```

What are Command-Line Runners and Application Runners, why do we need to use them?

While working with Spring Core application, we want to perform onetime startup activity after ioc container has been created, after bean definitions are instantiated before performing the operation within our application, then we can write such one-time startup logic immediately after creating the ioc container, as the reference of ioc container will be returned to us after creating.

```
ApplicationContext context = new
AnnotationConfigApplicationContext(JavaConfig.class);

// control comes to us here, after ioc container was
created

so write the startup logic in the nextline before
performing any of the operations.


This proves the total control of using the ioc
container is there within the hands of programmer.

-----------------------------------------------------
-----------------------------------------------------

web.xml


<listener>

  <listener-
class>org.springframework.web.context.ContextLoaderLi
stener</listener-class>

</listener>

<context-param>

  <param-name>contextConfigLocation</param-name>

  <param-value>/WEB-INF/application-
context.xml</param-value>

</context-param>


<servlet>

  <servlet-name>dispatcher</servlet-name>
```

```
  <servlet-
class>org.springframework.web.servlet.DispatcherServl
et</servlet-class>

</servlet>

<servlet-mapping>

  <servlet-name>dispatcher</servlet-name>

  <url-pattern>*.htm</url-pattern>

</servlet-mapping>


class MyApplicationInitializer implements
WebApplicationInitializer {

  public void onStartup(ServletContext context) {

    AnnotationConfigWebApplicationContext rootContext
= null;

    AnnotationConfigWebApplicationContext
servletContext = null;


    ContextLoaderListener listener = null;

    DispatcherServlet ds = null;


    rootContext = new
AnnotationConfigWebApplicationContext();

    rootContext.register(RootConfig.class);


  }

}
```

we want to perform one-time startup operation after the ioc container has been created after all the bean definitions are instantiated, before we perform the operation in our application, where do we need to write such a logic?

In Spring Core:

ApplicationContext context = new AnnotationConfigApplicationContext(JavaConfig.class);

Immediately after creating the ioc container, within the next line we can write the logic for perform one-time startup activity, before performing the operation using the ioc container.

In Spring Web Mvc application:

In a configuration, who creates the ioc container?

The ContextLoaderListener creates parent ioc container and places that ioc container in ServletContext, there after the **DispatcherServlet creates the child ioc container** by nesting the parent ioc container, once the init() method of the DispatcherServlet completed execution, our application will be ready for accepting the requests and there is no control here for the programmer to perform one-time startup activity.

The only way we can perform one-time startup activity is by creating our own ApplicationContextLoaderListener extending from ContextLoaderListener, but here we can only use parent ioc container bean definitions, the child bean

definitions are not available because DispatcherServlet has not yet instantiated.

Instead if we use Programmatic registration of DispatcherServlet and ContextLoaderListener, is there anyway we can perform one-time startup activity?

```
class DispatcherServletWebApplicationInitializer
implements WebApplicationInitializer {

  public void onStartup(ServletContext context) {

    DispatcherServlet dispatcherServlet = null;

    ContextLoaderListener contextLoaderListener =
null;

    AnnotationConfigWebApplicationContext
rootApplicationContext = null;

    AnnotationConfigWebApplicationContext
servletApplicationContext = null;


    rootApplicationContext = new
AnnotationConfigWebApplicationContext();


rootApplicationContext.register(RootConfig.class);


    contextLoaderListener = new
ContextLoaderListener(rootApplicationContext);

    context.addListener(contextLoaderListener);
```

```
    servletApplicationContext = new
AnnotationConfigApplicationContext();


servletApplicationContext.register(WebMvcConfig.class
);



    dispatcherServlet = new
DispatcherServlet(servletApplicationContext);

    ServletRegistration.Dynamic dynamic =
context.addServlet("dispatcher", dispatcherServlet);

    dynamic.setLoadOnStartup(1);

    dynamic.setMappings(new String[]{"*.htm"});

  }

}
```

Even in programmatic registration approach we cannot perform one-time startup activity, because even we create both ioc containers in the WebApplicationInitializer class, those containers are refreshed by the DispatcherServlet within the init() method of it, so there is no way to use the ioc container inside the WebApplicationInitializer class.

Now looks like in a Spring Web Mvc application, whichever the approach we are using in configuring DispatcherServlet and ContextLoaderListener, there is no way we get the reference of ioc container to perform one-time startup activity, this has been identified as a big gap within a Spring Framework application by Spring boot developers and to fill

this gap they introduced CommandLineRunners and ApplicationRunners.

No matter whichever the type of our application always our application begins execution with **SpringApplication.run()** method only while working with Spring Boot. The SpringApplication will perform lot of startup activities in bringing up the application like

1. create empty ioc container

2. detect and load external application configuration into ioc container

3. print banner

4. detect the type of application and instantiate appropriate ioc container

5. instantiate/register spring factories with ioc container

6. execute ApplicationContextInitializer

7. prepare context

8. refresh context (ioc container has been fully instantiated)

9. execute CommandLineRunners or ApplicationRunners if available

10. publish events and call listeners

The **SpringApplication class** to help us in performing onetime activity for our application, after instantiating the ioc container, after refreshing the ioc container before returning the reference of the

ioc container to the programmer, it searches for CommandLineRunner and ApplicationRunner bean definition within ioc container, if available in executes them automatically

if CommandLineRunner or ApplicationRunner fails execution, the application would be marked as failed and terminated. upon successful execution of CommandLineRunner or ApplicationRunners the reference of ioc container will be returned to us, so that we can perform application logic using the ioc container.

CommandLineRunners or ApplicationRunners both are meant for perform one-time startup activity, there is no different between them, apart from the type of arguments being passed.

The Spring Boot has provided 2 standard interfaces for writing a CommandLineRunner or a ApplicationRunner as below.

```
interface CommandLineRunner {
  void run(String... args);
}

interface ApplicationRunner {
  void run(ApplicationArguments args);
}
```

What are CommandLineRunners and ApplicationRunners what is the purpose of it?

To perform one-time startup activity within a Spring Boot Application, CommandLineRunners and ApplicationRunners are being used.

While working with Spring Framework and using mvc module in developing an application, there is no way the programmer can perform one-time startup activity within the application after creating the ioc container, after instantiating the object for the bean definitions, before the application begins serving the requests.

To overcome the above problem and ensure irrespective of the type of the application, spring boot has streamlined the startup activities of the Spring Boot application through the help of SpringApplication class.

Now programmers can write their application startup logic within CommandLineRunners or ApplicationRunners there by the SpringApplication takes care of calling them after the ioc container has been initialized.

To be executed by the SpringApplication class, we need write CommandLineRunner or ApplicationRunner implementing the standard interfaces provided by Spring Boot.

```
interface CommandLineRunner {
```

```
  void run(String... args);
}
interface ApplicationRunner {
  void run(ApplicationArguments args);
}
```

once we write our own CommandLineRunners and ApplicationRunners we need define them as bean definitions within ioc container. The SpringApplication class, after creating the ioc container, after executing ApplicationContextInitializer, prepare and refresh context it will goto ioc container searching for bean definition which are of type CommandLineRunner or ApplicationRunner and invokes them automatically.

during the execution of these, if there is an exception, the application startup will be marked as failed and terminates the application, else the reference the ioc container will be returned to the programmer.

What is the difference between CommandLineRunner and ApplicationRunner?

In case of CommandLineRunner the run() method will be passed with String args, where we can access all the command line parameters we supplied during the running the application as Strings.

In case of ApplicationRunner, the run() method takes ApplicationArguments class object using which we can extract NonOpsArguments and OpsArguments easily.

```
ApplicationContext context =
SpringApplication.run(Config.class, args);

// will not be executed
```

How to work with Spring Jdbc Application?

```
/*
class Test {
  public static void main(String[] args) {
    ApplicationContext context = new
AnnotationConfigApplicationContext(JavaConfig.class);
    GaurageService service =
context.getBean(GuarageService.class);

  }
}

@Configuration
@Import(PersistenceConfig.class)
@ComponentScan(basePackages="com.vogo.service")
class JavaConfig {
```

```
}
*/
class GuarageBo {
  int guarageClubRegNo;
  String guarageName;
  String proprietorName;
  String contactNo;
  String emailAddress;
  String location;
  // accessors
}

class GuarageDto {
  int guarageClubRegNo;
  String guarageName;
  String proprietorName;
  String contactNo;
  String emailAddress;
  String location;
  // accessors
}

@Service
class GuarageService {
  @Transactional(readOnly="true")
```

```
  public List<GuarageDto> getGuarages() {
    // write logic for calling dao and convert bo to
dto and return
  }
}

@Repository
class GuarageDao {
  @Autowired
  private JdbcTemplate jdbcTemplate;

  public List<GuarageBo> getGuarages() {
    return jdbcTemplate.query("select * from
guarage", (rs, rowNum)->{
      GuarageBo bo = new GuarageBo();
      bo.setGuarageClubRegNo(rs.getInt(1));
      bo.setGuarageName(rs.getString(2));
      ...
      return bo;
    });
  }
}
/*
@Configuration
@PropertySource("classpath:db.properties")
```

```
@ComponentScan(basePackages={"com.vogo.dao"})
class PersistenceConfig {
  @Autowired
  private Environment env;

  @Bean
  public DataSource dataSource() {
    DriverManagerDataSource dataSource = null;
    dataSource = new
DriverManagerDataSource(env.getProperty("db.url"),
env.getProperty("db.username"),
env.getProperty("db.password"));

dataSource.setDriverClassname(env.getProperty("db.dri
verClassname"));
    return dataSource;
  }

  @Bean
  public JdbcTemplate jdbcTemplate(DataSource
dataSource) {
    JdbcTemplate jdbcTemplate = new
JdbcTemplate(dataSource);
    return jdbcTemplate;
  }

  @Bean
```

```
  public PlatformTransactionManager
transactionManager(DataSource dataSource) {
    DataSourceTransactionManager transactionManager =
new DataSourceTransactionManager(dataSource);
    return transactionManager;
  }
}

db.properties
--------------
db.driverClassname=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/db
db.username=root
db.password=root
*/

application.yaml
----------------
spring:
  dataSource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/vogodb
    username: root
    password: root
```

```
class VogoApplication {

  public static void main(String[] args) {

    ApplicationContext context =
SpringApplication.run(VogoApplication.class, args);

    GuarageService gs =
context.getBean(GuarageService.class);


  }

}
```

How to work with Spring Data Jpa with Spring Boot?

```
@Entity

@Table(name="store")

class Store {

  @Id

  @GeneratedValue(stragety=GenerationType.AUTO)

  @Column(name="store_no")

  int storeNo;

  @Column(name="store_nm")

  String storeName;

  @Column(name="contact_no")

  String contactNo;

  @Column(name="email_address")

  String emailAddress;
```

```
  String address;
  String city;
  String state;
  int zip;
  String country;
  // accessors
}

class StoreDto {
  int storeNo;
  String storeName;
  String contactNo;
  String emailAddress;
  // accessors
}

interface StoreRepository extends
JpaRepository<Store, Integer> {

}

@Service
class StoreService {
  @Autowired
  private StoreRepository storeRepository;
```

```
  @Transactional(readOnly=true)
  public List<StoreDto> getStores() {}
}

/*
@Configuration
@PropertySource("classpath:db.properties")
@EnableJpaRepositories(basePackages={"com.vogo.reposi
tories"})
class PersistenceConfig {

  @Bean
  public DataSource
dataSource(@Value("${db.driverClassname}") String
driverClassname, @Value("${db.url}") String url,
@Value("${db.username}") String username,
@Value("${db.password}") String password) {
    DriverManagerDataSource dataSource = new
DriverManagerDataSource(url, username, password);
    dataSource.setDriverClassname(driverClassname);
    return dataSource;
  }

  @Bean
  public LocalContainerEntityManagerFactoryBean
entityManagerFactory(DataSource dataSource) {
```

```
    LocalContainerEntityManagerFactoryBean
localContainerEntityManagerFactoryBean = null;

    HibernateJpaAdapter jpaAdapter = null;


    localContainerEntityManagerFactoryBean = new
LocalContainerEntityManagerFactoryBean(dataSource);

    jpaAdapter = new HibernateJpaAdapter();

    jpaAdapter.setShowSql(true);

    jpaAdapter.setDdl(true);



localContainerEntityManagerFactoryBean.setJpaAdapter(
jpaAdapter);


localContainerEntityManagerFactoryBean.setPackagesToS
can("com.vogo.entities");


    return localContainerEntityManagerFactoryBean;

  }


  @Bean

  public PlatformTransactionManager
transactionManager(EntityManagerFactory emf) {

    JpaTransactionManager transactionManager = new
JpaTransactionManager(emf);

    return transactionManager;

  }
```

```
}


db.properties
db.driverClassname=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/vogodb
db.username=root
db.password=root


@Configuration
@ComponentScan(basePackages={"com.vogo.service"})
@Import({PersistenceConfig.class})
@EnableTransactionManagement
class JavaConfig {


}
*/
class Test {
  public static void main(String[] args) {
    ApplicationContext context = new
AnnotationConfigApplicationContext(JavaConfig.class);
    StoreService ss =
context.getBean(StoreService.class);
    List<StoreDto> stores = ss.getStores();
  }
}
```

In the above example we are able to eliminate most of the boiler-plate logic we write while working with Jpa api by using DataJpa. But to make datajpa work we need to configure lot of SpringFramework components as bean definitions which seems to a time taking and complex job to memorize and configure.

Go for Spring Boot, boot autoconfigurations takes care of configuring datajpa framework components as bean definitions automatically, so we dont need to write any configuration apart from passing configuration values for the corresponding properties to the autoconfiguration classes.

application.yml

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/vogodb
    username: root
    password: root
  jpa:
    properties:
      - showSql: true
      - ddl: true
```

```
@SpringBootApplication
class VogoApplication {
  public static void main(String[] args) {
    ApplcationContext context =
SpringApplication.run(VogoApplication.class, args);
  }
}
```

Spring Boot AutoConfiguration

Auto Configurations takes care of configuring Spring Framework components as bean definitions. The programmer dont have to write the code in tuning up the Spring Framework for our application, rather the framework tune up itself to our application.

Spring Boot goes with opinionated view in determining which components of the Framework are needed for our application to work and configures most of them by their defaults. If our requirements are diverging from defaults, then with a minimal configuration being passed to Spring Boot we can customize the way auto configurations takes care of configuring the framework components.

In case if we dont want the component to be autoconfigured, we can altogether disable them as well.

Auto Configuration

takes care of configuring framework components as bean definitions, so we can eliminate lot of configuration in our application and can speedup the development of the application

- the work based on opiniated view and configures with defaults

- diverging requirements, we can customize by supplying configuration values

- if we dont need we can disable.

Framework has lot of classes

based on which modules we are using those module components only should be configured as bean definitions by autoconfigurations

In Spring Boot how do we say we want to use these Spring Modules?

To help us in quickly using Spring Framework modules along with its dependent libraries, Spring Boot has provided starter dependencies, so we can determine which modules we are using based on starters, so we need to autoconfigurations based on starters we are using

So spring boot ships for every spring-boot-starter one autoconfiguration along with that, so to summarize autoconfigurations work along with starter dependencies.

for eg..

if we are using spring-boot-starter-jdbc, internally we have spring-boot-starter-jdbc-autoconfiguration will be imported automatically.

How does starter dependencies works in spring boot?

Starter dependencies are nothing but maven artifacts declared with spring modules and external third-party libraries as transitive dependencies within them.

identifying and importing module dependencies and external dependencies is difficult

spring-boot-starter-jdbc (maven central repo)

|-pom.xml

|-dependencies

|-spring-core

|-spring-context

|-spring-context-support

|-spring-el

|-spring-beans

|-commong-logging

|-spring-boot-starter-jdbc-autoconfiguration

spring-boot-starter-jdbc-autoconfiguration (maven project)

|- auto configurations class has been written by spring developers to configure jdbc module classes as bean definitions

The spring boot developers followed standard naming conventions in providing starters and autoconfigurations.

Each Spring Starter dependency is named with

spring-boot-starter-* and the corresponding autoconfigurations are shipped with naming convention as spring-boot-starter-*-autoconfiguration

DriverManagerDataSource - sql driver

JdbcTemplate

DataSourceTransactionManager

NamedParameterJdbcTemplate

SqlUpdate

SqlInsert

SqlQuery

MappingSqlQuery

8 jdbc classes = wrong

1. What are auto configurations? why do we need them?

Auto Configurations are the components that takes care of configuring Spring Framework components as bean definitions. There are opinionated in identifying and configuring the Framework components, most of the time they configure Framework components as bean definitions with default values.

If the requirements of our application are diverging from the defaults, we dont need to configure the framework components instead with minimal efforts we can tune up the auto configurations for our application by supplying the configuration values.

In case if we dont need any one or more bean definitions configured through auto configurations we can disable them.

2. How are those exists in Spring Boot?

Auto Configurations works closely with starter dependencies, based on the Spring Framework modules we are using in our application, we need those related Spring Framework components to be configured as bean definitions. while working with Spring Boot we import module dependencies through the help of boot starters, so spring boot ships autoconfigurations along with starter dependencies.

For each boot-starter dependency, spring boot has provided an boot-starter-autoconfiguration dependency as a transitive dependency, so that when we include starter dependency the relevant autoconfiguration also will be added to our project. which contains the auto configuration classes of the relevant spring modules which takes care of configuring the bean definitions of the module classes.

The spring boot has comeup with per each spring framework component, it has provided on auto configuration class to configure it as bean definition.

How does these auto-configuration classes works?

When we add a boot starter dependency in our project, the relevant autoconfiguration dependency also will be added into our project due to maven transitive dependency. if the auto configurations are by default executed, then all of the module dependencies specific to the starter will be configured as bean definitions, but there is no guarantee that all of

the classes of that module are being used in our project due to which huge amount of system resources (jvm memory and cpu) will be wasted.

To help us in identifying and controlling the execution of autoconfigurations, spring boot has provided conditional annotations. Each autoconfiguration class will be associated with an conditional annotation, based on the truthness of the condition only, the auto configuration class will be executed.

Spring Boot for all of their auto configurations provided they have associated them with conditional annotations to enforce or control when they need to execute and configure framework component as bean definition, (this is why it is called opinionated view).

There are 5 types of conditional annotations are there

1. Conditional on Class = These Conditional on Class annotations help us in executing the auto configurations based on the classes available under the classpath of the project.

```
@ConditionalOnClass(name="")
```

```
@ConditionalOnMissingClass(name="")
```

2. Conditional on Bean = These conditional on Bean annotations can be used for executing an auto configuration class only when a relevant bean definition is found within the ioc container.

```
@ConditionalOnBean(name="")
```

```
@ConditionalOnMissingBean(name="")
```

3. Conditional on Property = These conditiona on property auto configurations are executed if there exists a property in environment of the ioc container.

```
@ConditionalOnProperty(name="", value="")
```

For eg..

application.yaml

```
database.type=mysql
```

```
@Configuration
@ConditionalOnProperty(name="database.type",
value="mysql")
class MySqlDataSourceAutoConfiguration {


}
```

we can write our own auto configuration like MySqlDataSourceAutoConfiguration which would gets executed based on the property "database.type" defined in our application with value as mysql.

4. Conditional on Resource = if there is a resource found under the classpath of the application, then only execute the auto configuration

```
@ConditionalOnResource
```

For eg..

mysql-database.properties

db.driverClassname

db.url

db.username

db.password

```
@Configuration
@ConditionalOnResource("classpath:mysql-
database.properties")
class MySqlDataSourceAutoConfiguration {

}
```

5. Conditional on Application Type = The Conditional on Application annotations help us in executing an autoconfiguration class based on application type

@ConditionalOnNonWebAppliation

@ConditionalOnWebApplication

@ConditionalOnJava

How does these auto configurations are executed, and when those are executed?

When does those are executed:-

These auto configuration classes should be execute after ioc container has been created after loading the metadata of the application, before ioc container

begins instantiating the objects of the bean definitions, the auto configuration classes should be registered and used for instantiating the bean definitions.

That is where SpringApplication, when we call run() method,

1. create empty env object

2. detects and loads external configuration of our application into env

3. print banner

4. detects and instantiates appropriate ioc container

5. execute ApplicationContextInitializer

6. register spring factories and instantiate them (auto configurations are executed)

How does those auto configurations are identified and executed:

For each boot-starter, the boot-starter-autoconfiguration will be imported transitively. These auto configuration class will differ from boot-starter to starter we are using.

So how does Spring Application class knows which boot-starter has what auto configurations to be executed?

To help the SpringApplication class to identify the autoconfigurations of that starter, they ship an spring.factories file in autoconfiguration jar dependency under META-INF directory.

The file contains an property called org.springframework.boot.autoconfigure.EnableAutoConfiguration pointing to the fqn autoconfiguration classes of the boot-starter.

during the startup SpringApplication class will read each spring.factories file of the autoconfiguration jar and detects the auto configuration classes and register as part of step#6 defined above.

boot-starter-jdbc

boot-starter-jdbc-autoconfigure (jar)

|-META-INF

  |-spring.factories

org.springframework.boot.autoconfigure.EnableAutoConfiguration=fqn autoconfigurations starter

By default when we write @SpringBootApplication annotations the auto configurations are enabled, which means #6 will be executed. so after detecting the autoconfiguration by reading through spring.factories file, the auto configuration classes are registered to the ioc container by SpringApplication class.

Then ioc container instantiates the auto configuration classes and detects the conditional annotations, evaluates and based on the truthness of the condition, the bean definition methods of the autoconfigurations are executed by ioc container.

registering with ioc container = Spring Application

instantiating/executing autoconfigurations = ioc container itself

What are AutoConfigurations, why those are provided?

Auto Configurations are the configuration classes that takes care of configuring Spring Framework components as bean definitions for our application, so that programmer dont have to write the code/configuration in configuring the Framework components as part of the application, it save huge amount of development time and cost of development in building an application.

The auto configurations will determine which framework components should be configured as bean definitions based on opinionated view and configure them with defaults. if the requirements are diverging from defaults with minimal efforts we can pass configuration to auto configurations asking them to customize the configuration framework components. If needed we can disable them as well.

How does auto configurations are provided?

Auto Configurations are shipped along with spring boot starters. For each spring boot starter the

relevant auto configurations are added as transitive dependency. So that those autoconfigurations takes care of configuring the module specific framework components as bean definitions.

How does auto configurations works?

Auto Configurations works based on conditional annotations, those are not executed by default upon startup of the application.

There are 5 types of conditional annotations are there

1.@ConditionalOnClass (or) @ConditionalOnMissingClass

2.@ConditionalOnBean (or) @ConditionalOnMissingBean

3.@ConditionalOnProperty(name="", value="")

4.@ConditionalOnResource("location")

5.@ConditionalOnNonWebApplication (or) @ConditionalOnWebApplication or @ConditionalOnJava

The auto configurations classes provided by the Spring Boot has been annotated with conditional annotations by the developers itself.

How does these auto configurations will be registered and executed?

At the time of startup this Spring Boot application through SpringApplication.run(), after SpringApplication class has instantiated the ioc container and executed ApplicationContextInitializer detects and registers the spring factories with the ioc container.

it goes to autoconfiguration jar

autoconfiguration.jar

|-META-INF

  |-spring.factories

    |-
org.springframework.boot.autoconfigure.EnableAutoConfiguration=fqn autoconfiguration class

reads the spring.factories file from META-INF directory and identifies the list of auto configurations classes declared with key EnableAutoConfigurations and picks them and register with ioc container.

There after ioc container takes care of instantiating the autoconfiguration classes and evaluates conditional annotations, based on truthness of condition, it will execute bean definition methods of the auto configurations.

How to create our own custom auto configuration classes?

In general we have to create our own auto configurations when we are working in a product development and want to distribute our own product libraries to others. If we want to make our product libraries to be easily used by other people, without manually configuring them, then we can ship along with product libraries, the autoconfigurations also to configure our components as bean definitions within their application.

Let us say we are distributing an software library to be used by the other application developers. In order to use our software library by the other application developers they need to identify the third-party libraries required for our library in addition they need to configure our library components as bean definitions. Looks like using our library in developing the application is a complex job.

To make our library easy to use by others we can do the below.

1. create library-starter (maven empty project), by declaring library dependency, third-party libraries and library-autoconfiguration as transitive dependencies.

2. create library-autoconfiguration, in which write auto configuration classes which takes care of configuring library classes as bean definitions based on conditional annotations.

3. declare spring.factories file in autoconfiguration jar under META-INF directory and declare all the auto configurations of ours as key and value.

ship to the world.

Now people who want to use our library need to add library-starter in their project as dependency so that everything works by default.

cloudtalk (library)

```
|-pom.xml
```

intern our library uses third-party libraries like log4j etc

```
class VoipConnection {
  void connect(String dailNo) {}
}

class CloudTalkServer {
  String voipServerAddress;
  String agentCode;
  String authorizationCode;

  VoipConnection initialize() {
    // makes connect to exchangeServer through exchangeServerUrl, agentCode, authorizationCode
  }
}

class CloudCall {
  CloudTalkServer cloudTalkServer;

  public int dial(String dialNo) {
    VoipConnection connection = cloudTalkServer.initialize();
    connection.connect(dialNo);
```

```
    return code;
  }
}

------------------------------------------------------
------------------------------------------------------
QuickConnect
|-pom.xml
  |-cloudtalk
  |-log4j

@Component
class ConnectPanel {
  @Autowired
  CloudCall cloudCall;
  public void login(String leadFile) {
    // goes to one entry after another entry in
leadFile

  }
}

@Configuration
@PropertySource("classpath:cloudtalkconfig.properties
")
```

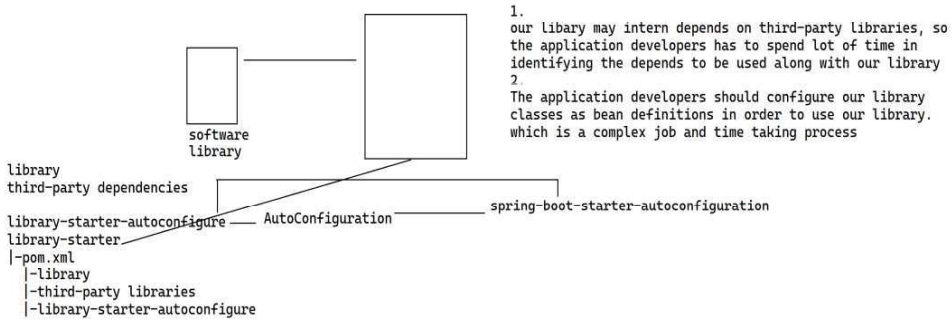
```
class QuickConnectConfig {

  @Autowired

  private Environment env;


  @Bean

  public CloudTalkServer cloudTalkServer() {

    CloudTalkServer cloudTalkServer = new
CloudTalkServer();

    // populate the values

    return cloudTalkServer;

  }


  @Bean

  public CloudCall cloudCall(CloudTalkServer
cloudTalkServer) {

    CloudCall cc = new CloudCall();

    cc.setCloudTalkServer(cloudTalkServer);

    return cc;

  }

}


application.yml

cloudtalk.use=true

cloudtalk.serverAddress=

cloudtalk.agent=
```

```
cloudtalk.authorizationCode=


------------------------------------------------------
------------------------------------------------------


cloudtalkparent
|-pom.xml
  type="pom"
  groupId: beetel
  artifactId: cloudtalkparent
  version: 1.0
  modules:
    cloudtalk
    cloudtalk-starter-autoconfigure
    cloudtalk-starter
|-cloudtalk
  |-pom.xml
    |-parent:
      groupId: beetel
      artifactId: cloudtalkparent
      version: 1.0
    |-dependency:
      log4j, scope: provided
|-cloudtalk-starter-autoconfigure
  |-pom.xml
```

```
    |-parent
      groupId: beetel
      artifactId: cloudtalkparent
      version: 1.0
    |-dependencies
      |-cloudtalk, scope: provided
      |-spring-boot-starter-autoconfiguration
  |-src
    |-main
      |-java
        |-CloudTalkServerAutoConfiguration
        |-CloudCallAutoConfiguration
      |-resources
        |-META-INF
          |-spring.factories
            |-
EnableAutoConfiguration=CloudTalkServerAutoConfigurat
ion,CloudCallAutoConfiguration
|-cloudtalk-starter
  |-pom.xml
    |-parent
      groupId: beetel
      artifactId: cloudtalkparent
      version: 1.0
    |-dependencies
```

```
      |-cloudtalk
      |-log4j
      |-cloudtalk-starter-autoconfigure



@Configuration
//@ConditionalOnResource("classpath:cloudtalkconfig.properties")
//@PropertySource("classpath:cloudtalkconfig.properties")
@ConditionalOnClass(name=CloudTalkServer.class)
class CloudTalkServerAutoConfiguration {
  @Autowired
  private Environment env;

  @Bean
  public CloudTalkServer cloudTalkServer() {
    CloudTalkServer cloudTalkServer = new CloudTalkServer();
    // populate property values from env
    return cloudTalkServer;
  }

}

@Configuration
```

```
@ConditionalOnBean(name="cloudTalkServer")

@ConditionalAfterClass(CloudTalkServerAutoConfigurati
on.class)

class CloudCallAutoConfiguration {


  @Bean

  public CloudCall cloudCall(CloudTalkServer
cloudTalkServer) {

    CloudCall cloudCall = new
CloudCall(cloudTalkServer);

    return cloudCall;

  }

}
```



Spring boot exit codes

Every program upon completing its execution, returns an exit code indicating the success or failure of its completion to the underlying operating system.

zero = indicates a successful completion of execution

non-zero = indicates the failure of execution and terminated (acts as an error code)

In a Java Application, it has to return an exitcode indicating the operating system or jvm the status of completion. by default the java application upon successful execution returns a zero as an exitcode. If the program while executing ran into an exception and terminated, then a non-zero exitcode will be returned by the java application automatically.

In case if we want to return an exitcode explicitly from our java program other than the default exit code to the operating system, then we can use System.exit(exitCode);

```
System.exit(1);
```

How to return an exitcode indicating the status of the execution of our spring boot application?

In Spring Boot application upon completion of the execution of our program by default return an exit-code indicating the success or failure of the completion of the program.

SpringApplication class takes care of registering shutdownHook(), on top of ioc container to ensure the bean lifecycle management methods are executed during the termination of the program gracefully. From this we can easily understand all the necessary echo system required for closing/terminating or gracefully shutting down the Spring Boot Application is in place.

There are few cases where we might want to customize the default exitcode is returned by our Spring Boot Application.

1. How to customize the exitcode returned by our Spring Boot Application?

```
@SpringBootApplication
class Application {
  public static void main(String[] args) {
    ApplicationContext context =
SpringApplication.run(Application.class, args);

    System.exit(SpringApplication.exit(context));
  }
}
```

Spring Boot Exit Code

Every program upon completing its execution, it is going to return an exitcode to the operating system indicating the success or failure in executing the program.

Zero = program executed terminated successfully

Non-Zero = program has encountered an error during execution and has terminated abnormally

In case of Java Programs also they return an exitcode indicating the operating system, the status of its execution. By default if java program completed execution without any exception it returns 0 indicating the success, if it has encountered any exception during execution, then a non-zero exitcode will be returned indicating the failure of execution

We can customize or change the exit code with which we want to terminate the java application by using System.exit(int)

How can we handle exitcode in Spring Boot Application?

Spring Boot Application is also an Java Application only, so the default behaviour of java application will be applicable to Spring Boot Application also, if the Spring Boot Application completed its execution successfully and terminated then it returns Zero otherwise it returns Non-Zero.

Unlike Spring Framework application, incase of Spring Boot the SpringApplication class has taken care of registering shutdownhook ontop of ioc container, so that no matter how our application is getting terminated always the bean lifecycle management methods will be called and our application will be terminated gracefully.

In addition to the above SpringApplication class has provided additional interfaces supporting the customization/handling of exit codes in Spring Boot Application using the below.

1. **ExitCodeGenerator**
2. **ExitCodeExceptionMapper**
3. **ExitCodeEvent**

How to terminate a Spring Boot Application upon completion of its execution?

Spring Boot has build the exitcode echo system as part of SpringApplication class, lets explore it.

```
@SpringBootApplication
class Application {
  public static void main(String[] args) {
    ApplicationContext context =
SpringApplication.run(Application.class, args);

    System.exit(29); //
```

```
  }
}
```

In a Spring Boot application dont call System.exit(int) directly in terminating and return an exitcode to the jvm, this is not recommended.

Spring Boot has provided an exit() method inside the SpringApplication class which we need to call in terminating an Spring Boot Application.

```
@SpringBootApplication
class Application {
  public static void main(String[] args) {
    ApplicationContext context =
SpringApplication.run(Application.class, args);
    try {

    }finally {
      int exitCode = SpringApplication.exit(context);
// exit code echo system spring boot has build
      System.exit(exitCode);
    }
  }
}
```

The SpringApplication.exit(context) = is a method used for gracefully terminating/closing the ioc container in Spring Boot Application. upon calling SpringApplication.exit(context) it performs several operations as below.

1. it looks for ExitCodeGenerator bean definitions within ioc container and make collection of them.

2. it tries to close the ioc container gracefully (context.close();)

2.1 while closing the ioc container, it there is an exception, then the exit() of SpringApplication will generate a random number indicating the failure of the termination and returns to us

2.2 if there is no exception and ioc container container has closed successfully, the SpringApplication class will call the ExitCodeGenerator class takes the exitcode and returns to us.

3. indicating the terminate of the program it publishes an ExitCodeEvent and looks for a listener to be called.

```
@Component
class ApplicationExitCodeGenerator implements
ExitCodeGenerator {
  public int generateExitCode() {
    return 10;
  }
}
```

Spring Boot Exit Code

How to work with customizing the exitcode of an Spring Boot Application?

In java application we can directly call System.exit(int) for return an custom exit code than the default, but the same is not recommended in Spring Boot Application.

Spring Boot has provided an exit(ApplicationContext) method within SpringApplication class which we need to invoke at the end of Spring Boot Application. The exit() method performs several closer activities inorder to terminate the application and returns an exitcode indicating the status of termination, there bywhich we need to pass the exitcode to System.exit(int) to customize the exit code of Spring Boot Application.

What closer activities SpringApplication.exit(int) method does?

1. Looks for all the ExitCodeGenerators in the ioc container and make collection of them

2. It will close the ioc container

  2.1 if there is no exception while closing the ioc container and it has terminated gracefully, it invokes the ExitCodeGenerator.generateExitCode() method and return the exit code to us

  2.2 if there is an exception encountered while closing the ioc container, it generates an random

integer number and returns to jvm by terminating the application

3. before returning the exitcode it publishes the ExitCodeEvent with exitcode it is return and invokes the EventListener, so that our application can see the exitcode with which it is terminating

```
@SpringBootApplication
class Application {
  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(Application.class, args);
    try {
      // execute the logic
    } finally {
      System.exit(SpringApplication.exit(context));
    }
  }
}
```

How to customize the success exitcode of the SpringBootApplication?

Write an ExitCodeGenerator and register with ioc container.

```
@Component
```

```
class ApplicationExitCodeGenerator implements
ExitCodeGenerator {

  public int generateExitCode() {

    return 29;

  }

}
```

Spring Boot Exit Code

---

1. ExitCodeGenerator

SpringApplication.exit(context);

2. ExitCodeExceptionMapper

ExitCodeExceptionMapper is used for mapping the startup failures that we encouter after SpringApplication class has created the ioc container, while executing the CommandLineRunners or ApplicationRunners into our own Custom ExitCode based on the Type of the exception.

Once the ioc container has returned to us, the total control of execution the application logic is within the hands of programmer, the SpringApplication class will not call ExitCodeExceptionMapper to map the application Exceptions into exitcode. The programmer himself has to sourround his code within try/catch block and map into exitcode of his own.

```
@SpringBootApplication
```

```
class Application {


  @Bean
  public CommandLineRunner loadCache() throws
FileNotFoundException {
    return (args)-> {
      throw new
FileNotFoundException("d:\\cities.properties not
found");
    }
  }


  public static void main(String[] args) {
    ApplicationContext context =
SpringApplication.run(Application.class, args);


    try {
      Rocket rocket = context.getBean(Rocket.class);
      rocket.launch();
    }catch(LaunchLevelException e) {
      SpringApplication.exit(context);
      System.exit(10);
    }finally {
      System.exit(SpringApplication.exit(context));
    }
```

```
  }
}

@Component
class ApplicationExitCodeExceptionMapper implements
ExitCodeExceptionMapper {
  public int getExitCode(Exception e) {
    if(e instanceof FileNotFoundException) {
      return 100;
    }
    return 1;
  }
}
```

Programmer has to write his own class implementing from ExitCodeExceptionMapper interface and override int getExitCode(Exception). based on the type of the Exception we can return an appropriate exitcode indicating the failure. So that SpringApplication class returns the exitcode to jvm while terminating the application.

3. ExitCodeEvent

While terminating the application SpringApplication class will publish an ExitCodeEvent wrapping the ExitCode with which our application is being terminated. We can get the exitcode of our application by writing an ApplicationListener class,

so that SpringApplication calls it before returning the exitcode.

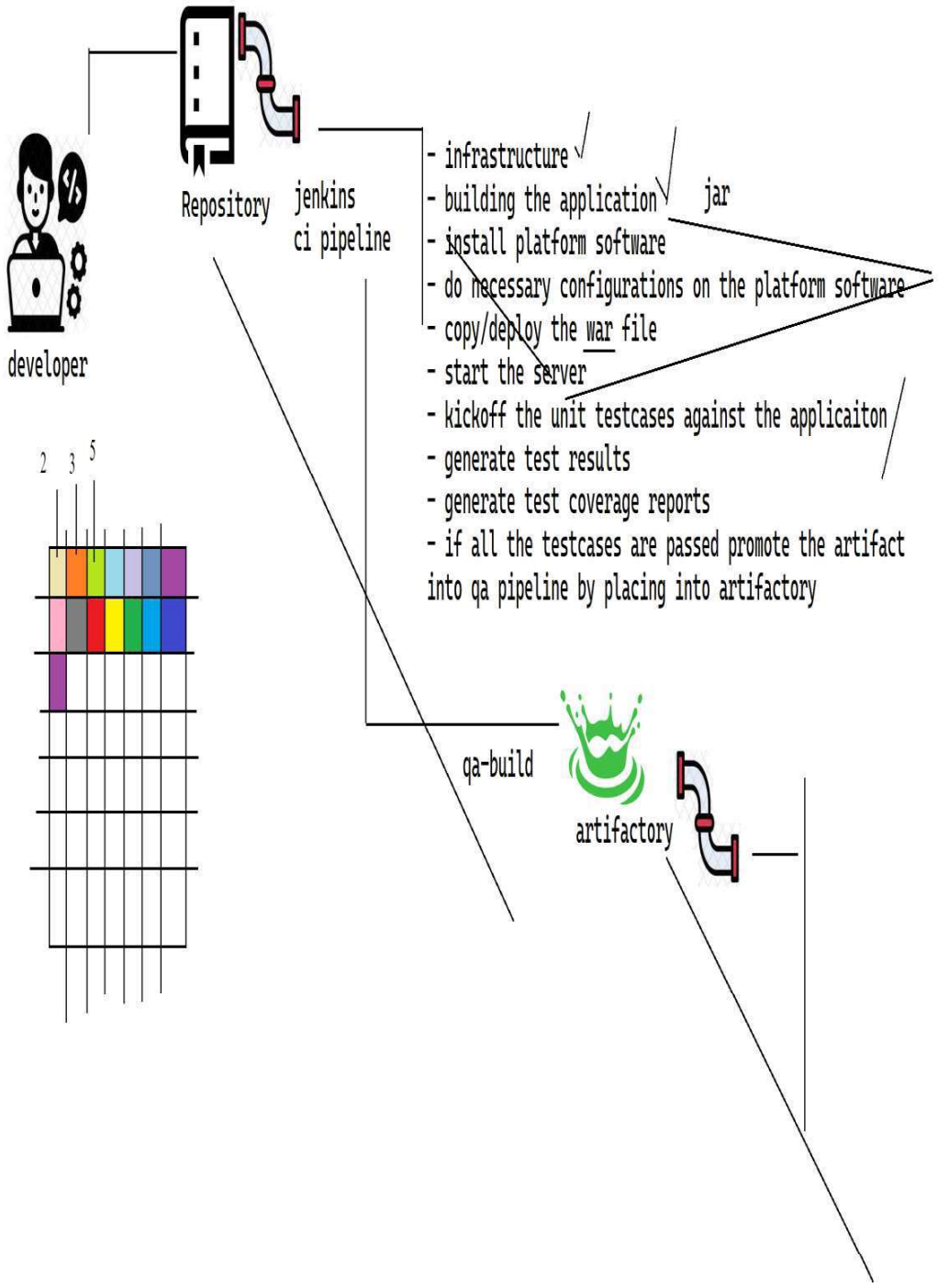
```
@Component
class ApplicationExitCodeEventListener implements
ApplicationListener<ExitCodeEvent> {
  public void onApplicationEvent(ExitCodeEvent event)
{
    System.out.println("exitcode : "
+event.getExitCode());
  }
}
```

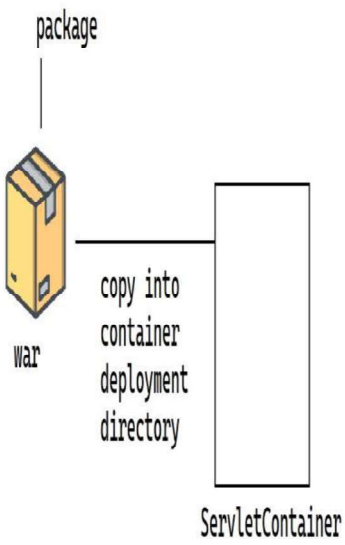
Spring Boot with Spring Web Mvc

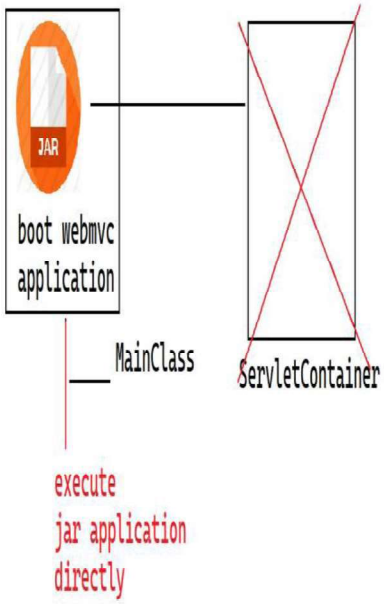
In general when we are working spring mvc application we package our application as a war packaging and deploy the application on servlet container and start the container to run our application. But while working with SpringBoot WebMvc Application we dont need to package our application necessarily as "war", rather we package our application as Jar only. We dont need to explicitly deploy our application onto an ServletContainer rather we can run our web application out of the Main method directly without copying/deploying and starting the server.

By looking at the above, we would wonder what type of benefits we get if we are packaging the application as Jar and running out of main method, seems to be a brainless thing,

There are lot of advantages of running the application directly out of a main method without packaging and deploying the war application on ServletContainer, let us try to explore.


Repository
jenkins
ci pipeline
- infrastructure
- building the application
jar
- install platform software
- do necessary configurations on the platform software
- copy/deploy the war file
- start the server
- kickoff the unit testcases against the applicaiton
- generate test results
- generate test coverage reports
- if all the testcases are passed promote the artifact into qa pipeline by placing into artifactory
developer
2
3
5
qa-build
artifactory





1. why not war, why are we packaging in jar only?
2. how to execute a web application out of a main method, where does server container comes from?
3. if we build spring boot webmvc application can we deploy it as a war application on a ServletContainer or not?

Spring Boot Web Mvc

When we are working with Spring Web Mvc application, we package our application as a war and deploy it on the Servlet Container/Application Server. But while working with Spring Boot in building web mvc application we dont package our application as war, rather we package the application as Jar and run out of MainClass without deploying the application in a Servlet container manually.

Internally Spring Boot takes care of deploying the application within the Embedded Servlet Container and run our application.

What is Embedded Servlet Container?

Instead of downloading and installing the servlet container on a machine, the container is shipped as a jar dependency out of our build code. The application will be deployed on the embedded container through application itself.

Why do we need to use embedded containers, why we need to package an web application as a Jar, what kind of advantages do we get here?

There are lot of benefits are there in packaging and running an application out of Jar MainClass and deploying on embedded servlet container, let us try to explore

1. With embedded servlet containers, the server is shipped as part of the code itself. when we clone the repository along with our source code we get the server as dependency in build file. during the build process the server will be downloaded as jar dependency from the artifactory repository. Out of the code itself the server would be started and the application would gets deployed and executed.

From the above we can understand, all we need to run our application is the Source code there is no additional setup or installation required to run our application, this make our code easily carriable and executable anywhere

2. In a traditional deployment after setting up the server we need to make required configurations in the container like security, datasource configurations, jms queues/topics, connectionpool configurations etc prior to deploying the application. Unless we have these changes in place applied manually on the container we cannot deploy the application,this means one has to know the configurations to be made on the server to run our application which makes application execution very complex

In place of this if we use embedded servlet container as the server is being kicked off from the code, the required configurations on the server can be done through the code itself so that it greatly reduces the efforts of configuring and executing the application.

3. While adopting the devops process and setting up CI/CD pipeline we need to write lot of code in

building and running the application in automation process.

To install the platform software like Application Server or Servlet Container to run our application we need to write the code using ShellScript or Software Configuration Management Tools like Ansible, Puppet, Salt, Chef etc. In addition to make required configurations to be made on server to run our application we need to write phython, wlst or shellscript code.

There after we need to build our application and copy the application artifact on to the server environment and kick the container. All the above things are labourious job which required huge time and great level of automation

With the help of embedded servlet containers inplace all the above process of installing and configuring the platform software has been avoided. Simple build the application and execute out of a Jar itself, which helps in quickly adopting the CI/CD pipeline process

4. The embedded servlet containers fully support development and delivery of the microservices based applications. In case of microservices each service should be developed out of its own code based and should be executes independent of another (in a separate container). In our application comprises of 20 microservices, it would be hard to setup 20 containers on a machine and configure them to run each of the microservices separately. Instead of this if we can run the application out of the embedded containers we dont need to install, configure the container to run the services which makes the

development and delivery and deployment process easy while working with microservices.

From the above we might think the spring boot web mvc application can only work on embedded servlet containers which is wrong perception. we can package spring boot web mvc application as a war as well and deploy in traditional application server or servlet container as well.

Based on the nature/type of your application you can decide to run it out of a embedded container or a traditional war deployment.

For eg.. if we are building microservices based application, we would want our application to run out of embedded container itself, as the application is relative small in nature and deosnt required any jee platform services

But we are building an traditional webapplication which required enterprise class-level features provided through jee containers, we can package and run the application out of an embedded container during development and convert into war package and deploy on standalone application server during production deployment.

While working with Spring Boot through spring-boot-starter-webmvc the embedded servlet container also added as transitive dependency. We dont have to write the code in registering/deploying our application in the embedded servlet container and run the container, all of these things would be taken care by the Spring

Boot itself, we just need to run our application as any other spring boot application just by calling

SpringApplication.run(Config.class, args);

Spring Boot supports 4 embedded servlet containers currently

1. tomcat (default)
2. jetty
3. undertow
4. netty (reactive application deployment)

if we want we can explicitly switch from one container to another container by configuring exclusions and adding other container as dependency in spring-boot-starter-webmvc

---

While working with Spring Boot we dont have to write the code in deploying and running the application on embedded servlet container, the Spring Boot itself takes care of deploying and running the application directly on embedded servlet container.

We dont need to add any additional dependency in adding the embedded servlet container, when we include spring-boot-starter-web, it internally adds embedded tomcat server as a default server in deploying and running the applications.

Spring Boot supports 4 embedded servlet containers

1. Tomcat (default)
2. Jetty
3. Undertow
4. Netty (Reactive)

How to develop spring web mvc application with datajpa using Spring boot?

vogo application

- list all the bikes in the bikes table

#1 DispatcherServlet and ContextLoaderListener (programatic approach)

```
class VogoDispatcherServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer
{

  public Class<?>[] getRootConfigClasses() {
```

```
    return new Class<?>[] {RootConfig.class};
  }
  public Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {VogoWebMvcConfig.class};
  }
  public String[] getServletMappings() {
    return new String[]{"*.htm"};
  }
}
```

#2 HandlerMapping and ViewResolver

```
@EnableWebMvc
class VogoWebMvcConfig implements WebMvcConfigurer {
  public void
configureViewResolver(ViewResolverRegistry registry)
{
    registry.jsp("/WEB-INF/jsp/").sufix(".jsp");
  }
}
```

Replacing the ViewResolver configuration we need to write below configuration in application.yml

```
spring:
  mvc:
    view:
```

```
      prefix: /WEB-INF/jsp/
      suffix: .jsp
```

#3 Entity class

```
@Entity
@Table(name="bike")
class Bike {
  @Id
  @GeneratedValue(strategy=GenerationType.Identity)
  int bikeNo;
  String bikeModelName;
  String manufacturer;
  String rtaRegistrationNo;
  double price;
  // accessors
}
```

#4. Repository Interface

```
interface BikeRepository extends JpaRepository <Bike,
Integer> {}
```

#5

@Configuration

```
@PropertySource("classpath:db.properties")
```

```
@EnableJpaRepository(basePackages =
{"com.vogo.repositories"})

class PersistenceConfig {

  @Autowired

  private Environment env;


  @Bean

  public DataSource dataSource() {

    DriverManagerDataSource dataSource = null;

    // write logic

    return dataSource;

  }

  @Bean

  public
LocalContainerEntityManagerFactoryBean(DataSource
dataSource) {}

  @Bean

  public PlatformTransactionManager
transactionManager(EntityManagerFactory emf) {}

}
```

replacing the above 3 bean definitions in spring boot application.yml we need delcare below configuration

```
spring:

  datasource:

    driver-class-name:
```

```
    url:
    username:
    password:
  jpa:
    show-sql: true
    ddl: true
    format-sql: true
```

#6 Service class

```
@Service
class BikeService {
  @Autowired
  BikeRepository bikeRepository;
  public List<BikeDto> getBikes() {}
}
```

#7 BikeDto

```
class BikeDto {
  int bikeNo;
  String bikeModelName;
  // accessors
}
```

#8 RootConfig

```
@Configuration
@Import(PersistenceConfig.class)
@EnableTransactionManagement
@ComponentScan(basePackages = {"com.vogo.service"})
class RootConfig {
}
```

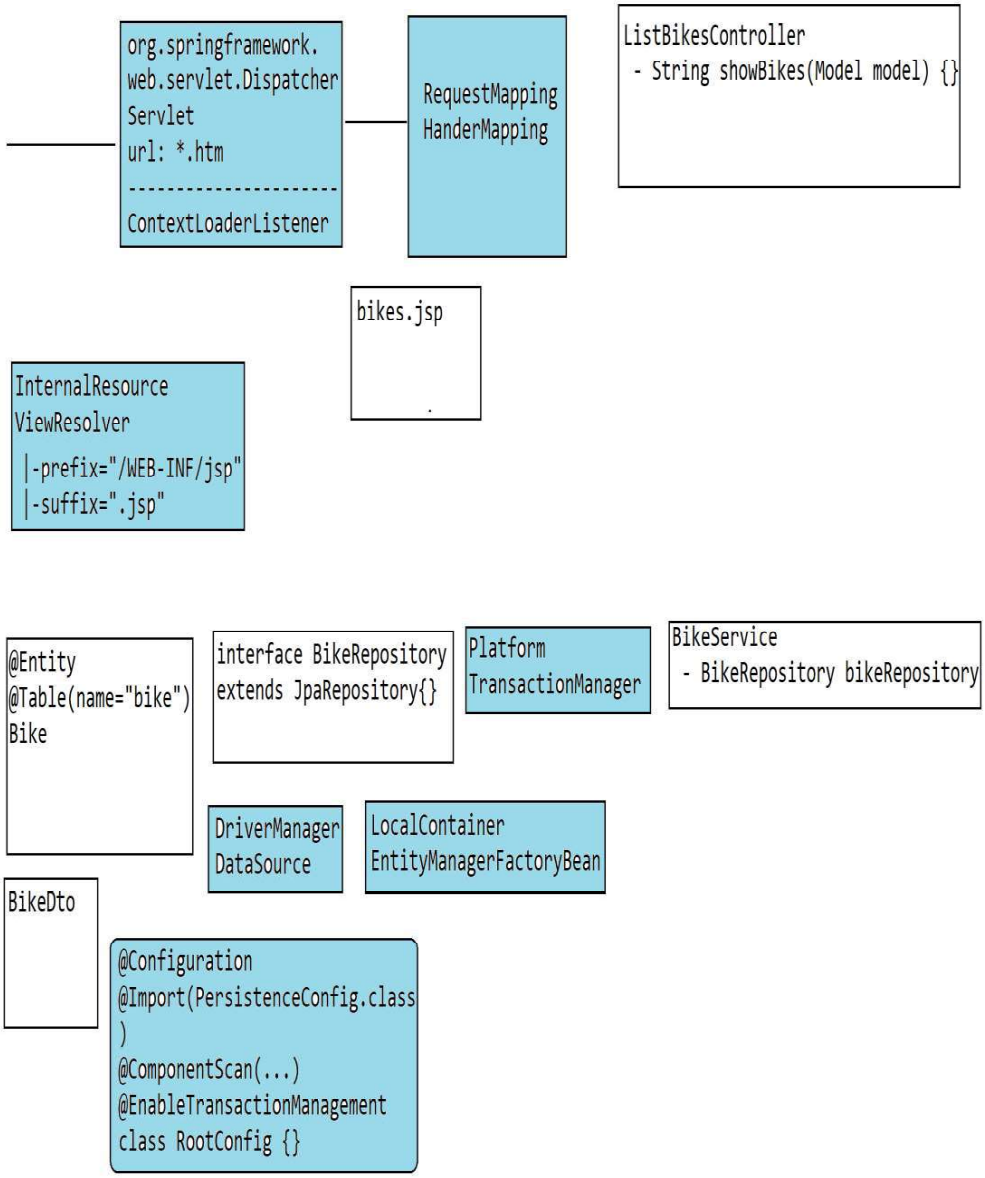
#9. Controller

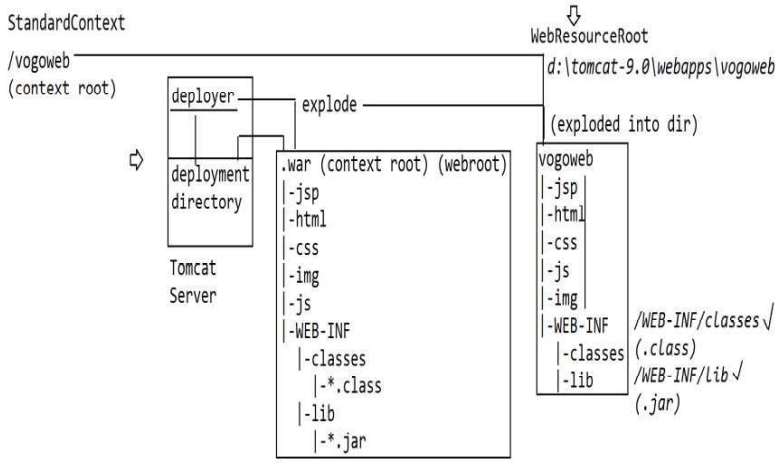
```
@Controller
class ListBikeController {
  @Autowired
  private BikeService bikeService;

  @GetMapping("/bikes.htm")
  public String showBikes(Model model) {}
}
```

#10 bikes.jsp

```
<body>
  <c:forEach items="${bikes}" var="bike">
    // display all bikes in table
```

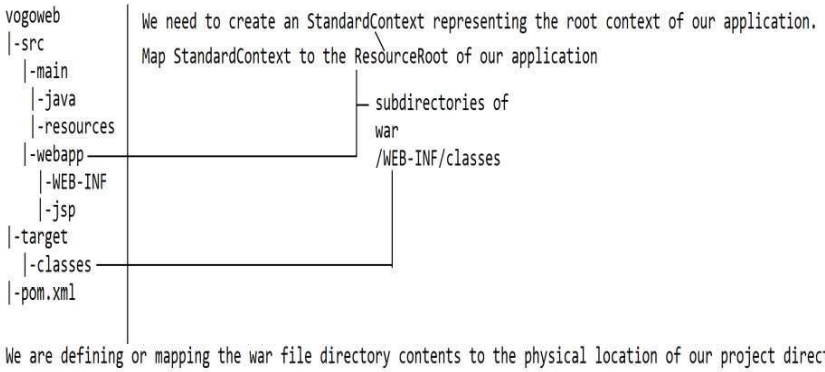
```
  </c:forEach>
</body>
```





While working with embedded servlet containers, we dont package our application as "war", because there is no physical existence of the server. Through our application code we are going to instantiate the Tomcat server and run the server. So we cannot copy or deploy our application into any physical location (directoy) on the machine, which means tomcat deployer will not comes into picture.

Now to deploy an war application into embedded tomcat server, the job of deployer has to be done through the program itself by the programmer.



We are defining or mapping the war file directory contents to the physical location of our project directories so that the tomcat server can locate the contents of war based on location we specified in serving the request.

When we add embedded tomcat server as a dependency into our project, we get tomcat api into our application, using which we need to register our application and run it.

```
Tomcat tomcat = new Tomcat(8081);
StandardContext context = (StandardContext) tomcat.addWebApp("/vogo", new File("src/main/webapp").getAbsolutePath());
WebResourceRoot directory = new StandardRoot(context);
directory.addPreResource(new DirResourceSet(directory, "/WEB-INF/classes", new File("target/classes").getAbsolutePath(), "/");

context.setResources(directory);
tomcat.start();
```

What is an embedded servlet container and how to deploy our applications on embedded servlet container?

Instead of installing and configuring the servlet container to deploy our applications, the server is shipped as a dependency by declaring in our project.

In a standalone deployment process, we use to package our application as war and deploy into onto an deployment directory of the server. The war deployer of the servlet container takes care of deploying the war application onto the Tomcat server as below.

1. picks the war from deployment directory
2. validate the war
3. it explodes the contents of the war into a directory structure format
4. then registers the information about the war contents with the tomcat server
    4.1 creates an StandardContext and binds the webroot directory with the /contextroot
    4.2 creates directory structure of the war represented by WebResourceRoot for that standard context
    4.3 defines the directory of the WebResourceRoot as /WEB-INF/classes and /WEB-INF/lib directory respectively
5. changes the state application to deployed and ready

While working with embedded containers, there is no physical location of the server or a deployment directory to deploy our war file, so there is no presence of deployer. The job of the deployer like create StandardContext, associating it to the WebResourceRoot etc has to be done through the code by the developer itself.

The Embedded Tomcat server has provided api classes to work with deploying and running the application.

#1 create an instance of the tomcat server

```
Tomcat tomcat = new Tomcat();

tomcat.setPort(8081);
```

#2 register an StandardContext representing the root directory of our application

```
StandardContext context = tomcat.addWebApp("/esweb",
new File("src/main/webapp").getAbsolutePath());


WebResourceRoot webResourceRoot = new
StandardRoot(context); // war


DirResourceSet classesSet = new
DirResourceSet(webResourceRoot, "/WEB-INF/classes",
new File("target/classes").getAbsolutePath(), "/");
```

In to the WebResourceRoot we want to map contents of our project under target/classes into war file /WEB-INF/classes directory from the root of the war

```
webResourceRoot.addPreResources(classesSet);

context.setResources(webResourceRoot);


tomcat.start();

tomcat.getServer().await();
```

What is event-driven programming, why do we need to use it?

event-driven programming is a message oriented asynchronous communication model, where the callee and the caller dont see the interfaces of each other, but they communicate with each other by exchanging the messages/events between them.

it is an one way communication model.

1. loosely coupled
2. uni-directional communication
3. asynchronous

Java has provided an api to implement event-based programming which is observer api, where it is too low-level api where programmer has to write lot of code in implementing an application. In Jee platform jms api is available to implement event driven programming, which is an enterprise level messaging solution which becomes heavy weight for simple application requirements.

Instead of using observer api which is too low-level or jms api which is heavy weight spring has provided

an a better support for event-driven programming within its core container which Spring Event-Driven model.

How to work with event-driven programming in Spring Framework?

Event:-

Spring has provided an standard base class to represent an event which is ApplicationEvent class. for all the events to identify the source who is publishing the event we Source object, So ApplicationEvent class has been declared with Object source as an attribute to populate the information about the source.

We can write our own event class extends ApplicationEvent and we declare additional attributes to populate and carry the data to Handler/Listener

```
class ReloadCacheEvent extends ApplicationEvent {
  String key;
  public ReloadCacheEvent(Object source, String key)
{
    super(source);
    this.key = key;
  }
  // accessors
}
```

When the source has published the event, the ioc container acts as an Listener in receiving the events and invoking the handler.

How to write an Handler class?

Since ioc container has to invoke the handler we should write the Handler class by implementing an interface called ApplicationListener or annotate the handler method with @EventListener and declare the event we want to recieve as a input to perform the operation.

```
@Component (register with ioc container)

class ReloadCacheEventApplicationListener implements
ApplicationListener<ReloadCacheEvent> {

  public void onApplicationEvent(ReloadCacheEvent
event) {}

}
```

(or)

```
@Component (register with ioc container)

class ReloadCacheEventApplicationListener {

  @EventListener

  public void onReloadEvent(ReloadCacheEvent event)
{}

}
```

```
@Component
class CityController /*implements
ApplicationEventPublisherAware*/ {
  @Autowired
  ApplicationEventPublisher publisher;


  public void updateCity(int cityId, String cityName)
{
    // update the data in database by invoking
service class
    ReloadCacheEvent event = new
ReloadCacheEvent(this, "cities");
    publisher.publishEvent(event);
  }


  //ioc container
  public void
setApplicationEventPublisher(ApplicationEventPublishe
r publisher) {
    this.publisher = publisher;
  }
}
```

To publish the events to the ioc container, Spring has provided ApplicationEventPublisher which is an

internal object of the ioc container, in order to get the reference of internal object of ioc container we need to use Aware interface which is ApplicationEventPublisherAware.

ApplicationEvents and Listeners in Spring Boot

When we launch spring boot application by executing the below line of code.

```
SpringApplication.run(Config.class, args);
```

During the time of starting up the Spring boot application using SpringApplication class it performs various different activities in bring up the spring boot application like

1. create empty environment object (e1)

2. detect the external configuration of our application and load into environment object (e2)

3. print banner (e3)

4. detect the type of the application based on the classes under the classpath and instantiate the appropriate ioc container

4.1 if webmvc jar is found under classpath treat WebApplicationType = WEB and instantiate AnnotationConfigServletWebServerApplicationContext

4.2 if webflux jar is found under classpath treat WebApplicationType = REACTIVE and instantiate AnnotationConfigReativeWebServerAppliationContext

4.3 otherwise treat WebApplicationType = NONE and instantiate AnnotationConfigApplicationContext (e4)

5. Instantiate and register Spring Factories with ioc container (e5)

etc

```
ApplicationContext context =
SpringApplication.run(Config.class, args);
```

How does spring boot spring webmvc application works, what are the bootstrapping process takes place when we run the Spring Boot application?

When we run SpringApplication.run(Config.class, args)

1. creates an empty env object

2. detects the external configuration of our application and loads into env object

3. print banner

4. identify the type of the application based on dependencies under the classpath

  4.1 if Web Mvc dependencies are found, then treats the WebApplicationType = WEB and

    instantiates the AnnotationConfigServletWebServerApplicationContext

  4.2 if Webflux dependencies are found, then it treats the WebApplicationType =REACTIVE and

    instantiates the AnnotationConfigReactiveWebServerApplicationContext

  4.3 else treats WebApplicationType = NONE and

instantiates the AnnotationConfigApplicationContext

we added spring-boot-starter-web, so the webmvc dependencies are available under classpath, so the SpringApplication, run() method would instantiate AnnotationConfigServletWebServerApplicationContext

5. Instantiates and Registers Spring Factories to the ioc container

6. Invoke ApplicationContextInitializer

7. prepare context

8. refresh context

- internally invokes onRefresh() method of ioc container.

How does spring boot webmvc application works, what is the internal flow of execution?

SpringApplication.run(Config.class, args)

1. it creates an empty environment object

2. detects the external configuration of the application and loads into env object

3. print banner

4. detects the type of the application based on the libraries in the classpath

if webmvc jar dependencies are found it treats WebApplicationType = WEB and

instantiates AnnotationConfigServletWebServerApplicationContext

if webflux jars are found under the classpath, it treats the WebApplicationType = REACTIVE and

instantiates AnnotationConfigReactiveWebServerApplicationContext

otherwise treats the WebApplicationType NONE

instantiates AnnotationConfigApplicationContext

since we added spring-boot-starter-web, it creates AnnotationConfigServletWebServerApplicationContext

5. instantiates and registers the Spring Factories

6. invokes ApplicationContextInitializer

7. prepare context

8. refresh context

while refreshing the context, internally onRefresh() method will be invoked on AnnotationConfigServletWebServerApplicationContext.

8.1 upon refreshing the ioc container all of the bean definitions within the ioc container has been instantiated, so that we need to expose our application by depoying on the server. since the information about the application objects are fully instantiated is known to ioc container let him take the responsibility of deploying and managing the application on the Embedded Container.

Why does the job of deploying and managing the application should be taken care by AnnotationConfigServletWebServerApplicationContext, why not SpringApplication class manages it?

deploying and managing the application on the server is specific to web/reactive applications and having it as part of SpringApplication class will apply to all the types of applications. So to make it

applicable only to web/reactive applications the specialized ApplicationContext implementations are provided respectively

From the above we can understand the job of deploying and managing the application on the embedded servlet container is taken care by AnnotationConfigServletWebServerApplicationContext.

If AnnotationConfigServletWebServerApplicationContext is taking care of managing the embedded servlet containers then it will tightly coupled with the embedded containers it is supporting. to allow extensibility and decouple the ioc container from specific embedded container, the WebServer interface and implementations are introduced.

```
interface WebServer {

  void start();

  void stop();

  void
shutdownGracefully(ConfigurationApplicationContext);

}
```

for this interface, there are 4 implementations are there to support 4 embedded servlet containers

TomcatWebServer

JettyWebServer

NettyWebServer

UndertowWebServer

These implementations classes abstracted the underlying details how to manage their respective servlet containers in stopping, starting etc. Now AnnotationConfigServletWebServerApplicationContext dont need to bother about how to start/stop or manage a specific embedded container, instead it can work with WebServer implementations to manage the Embedded Containers.

Now the ioc container should not instantiate the object of WebServer implementation, because the ApplicationContext will be tightly coupled with the specific web server. To help in decoupling the ApplicationContext from specific web server, WebServerFactory was introduced.

There are 2 types of WebServerFactories are there

1. ServletWebServerFactory

2. ReactiveWebServerFactory

The above 2 are abstract classes which has respective webserver specific implementation factories are there, because the logic for instantiating and configuring the implementation of web server differs from web server to another.

ioc container now instantiates the appropriate implementation of WebServerFactory for e.g.. if we are using tomcat server, then TomcatWebServerFactory will be used by ioc container.

Using WebServerFactory implementation it creates the object of respective WebServer implementation and invokes start() to start the server

9. execute CommandLineRunners and ApplicationRunners

10. publish events and invoke listener during the above stages of aoperations

During the startup of the Embedded Servlet container it invokes ServletContextInitializer implementation class.

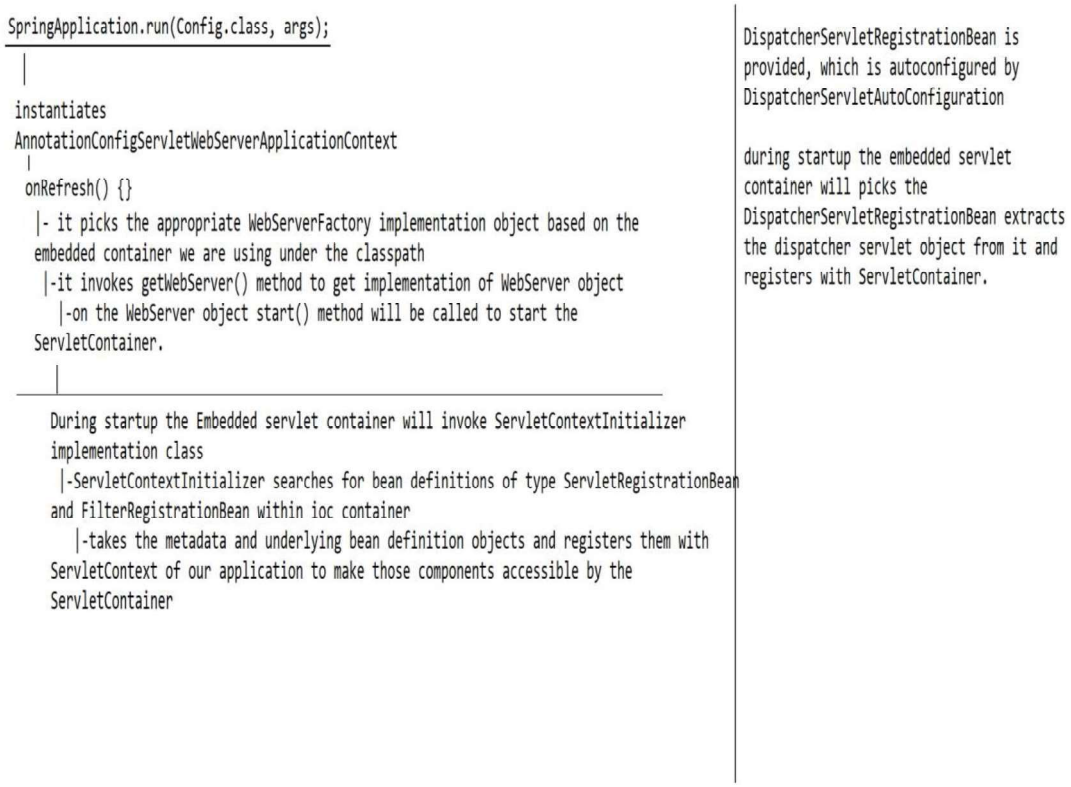
The ServletContextInitializer searches for bean definitions in the ioc container which are of type ServeltRegistrationBean or FilterRegistrationBean, if found then extracts the metadata around them and registers by adding them to the ServletContext of the embedded servlet container.

In case of spring mvc application we need DispatcherServlet to be registered with Servlet container, Spring Boot has provided DispatcherServletRegistrationBean, the DispatcherServletAutoConfiguration class will configure DispatcherServletRegistrationBean as bean definition.

WebServerFactory

|-ServletWebServerFactory

|-ReactiveWebServerFactory



How to deploy a spring boot application as a war deployment on a standalone application server or a servlet container?

1. change the packaging type of the maven project from jar to war

2. exclude embedded tomcat server as a transitive dependency in pom.xml

```
<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>
```

3. write the Spring Boot Application class extending from SpringBootServletInitializer and override the configure(SpringApplicationBuilder)

```
  @SpringBootApplication
  class Application extends
SpringBootServletInitializer {
    public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
// embedded servlet container this code will works
    }

    public SpringApplicationBuilder
configure(SpringApplicationBuilder builder) {
      return builder.sources(Application.class);
    }
  }
```

Servlet Container

-> ServletContainerInitializer implementation class
onStartup(Set<Class<?>> classes, ServletContext) {}

-> SpringServletContainerInitializer { // servlet container will call

onStartup(Set<Class<?>> classes, ServletContext) {}

}

-> implementation of WebApplicationInitializer which is nothing but SpringBootServletInitializer

SpringBootServletInitializer {

void onStartup(ServletContext context) {}

}

-> internally calls the configure(SpringApplicationBuilder) of our Application class.

How to configure a Servlet or a Filter in a Spring Boot Application?

There are 2 ways we can configure a Servlet/Filter component in Spring Boot Application

1. no source code

if we dont have source code of our servlet or filter, then we can configure them using ServletRegistrationBean and FilterRegistrationBean respectively.

wrap our Servlet object or Filter object into ServletRegistrationBean or FilterRegistrationBean objects and make them as bean definitions in the ioc container.

during the startup the Servlet Container will invoke ServletContextInitializer, which there by searches for bean definitions of type ServletRegistrationBean or FilterRegistrationBean and extracts the information and register with ServletContext of our application.

The Servlet or Filter is now acting as an bean definition within ioc container and along with that exposed through Servlet Container.

```
// no source code
class PingServlet extends HttpServlet {
  public void service(HttpServletRequest request,
HttpServletResponse response) {
    response.getWriter().print("OK!");
  }
}
```

```
@SpringBootApplication
public Application {
```

```
  @Bean
  public ServletRegistrationBean pingServlet() {
    ServletRegistrationBean bean = new
ServletRegistrationBean("ping", new PingServlet());
    return bean;
  }

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

2. source code

if we have the source code of the servlet, then we can directly annotate the class with @WebServlet annotation, and in java configuration class specify @ServletComponentScan pointing to the packages of the Servlet classes.

```
@WebServlet(urlPatters={"/ping"})
class PingServlet extends HttpServlet {}

@SpringBootApplication
@ServletComponentScan(basePackages={"com.sb.servlets"
})
class Application {
```

```
}
```

How to customize the embedded servlet container while working with Spring Boot?

In Spring Boot the embedded servlet container is managed through Spring ApplicationContext, So how do we need to customize in configuring and starting an embedded servlet container while working with spring boot?

There are 2 ways are there in customizing the embedded servlet containers

#1 configuration approach

we can configure the server configuration properties in application.yml or properties like below.

```
server:
  port: 8081
  servlet-context:
    path: /vogo
```

The above properties are read by WebServerFactoryCustomizerAutoConfiguration and configures WebServerFactory bean definition with these values, The ioc container uses WebServerFactory for creating the object of WebServer. Now the WebServer will be instantiated using the above properties by the WebServerFactory, so that embedded servlet container will be customized.

#2 programmatic approach

There are 2 ways we can customize the embedded servlet container in programmatic approach

#2.1

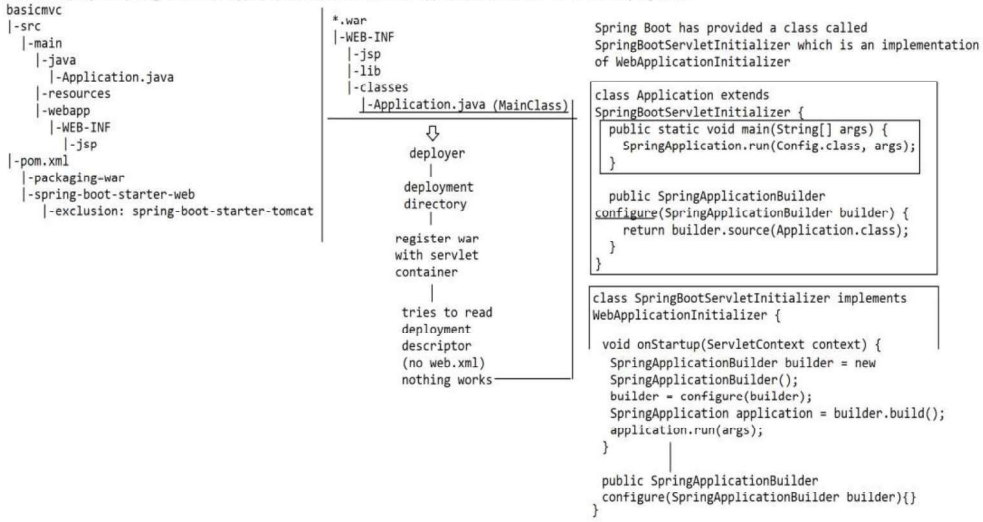
```
@Component

public class EmbeddedServerFactoryCustomizer
implements
WebServerFactoryCustomizer<ConfigurableWebServerFacto
ry> {

  public void customize(ConfigurableWebServerFactory
webServerFactory) {

    webServerFactory.setPort(8081);

  }

}
```

#2.2

```
@SpringBootApplication

class Application {

  @Bean

  public ConfigurableWebServerFactory
configurableWebServerFactory() {

    TomcatServletWebServerFactory factory = new
TomcatServletWebServerFactory();

    factory.setPort(2020);

    return factory;
```

```
    }
}
```

How to deploy a Spring boot mvc application in a standlone application server as a war deployment?



acutuator endpoints

When we build an application post completion of the testing we cannot deliver the application into production env, because to manage and monitor the application in production environment we need to add bunch of utilities like

- **healthcheck**
- **metrics**
- **threaddumps**
- **logs**
- **info**

unless we expose the apis in collecting the realtime data about our application, the monitorings tools will not be able to grab the data from our application and cannot help us in managing our application

Post completion of our development, the developer has to spend enough amount of time in building required utilities and integrate them into application to make it production ready.

1. the developer has to spend additional time in making the application production ready
2. the cost of making the application production ready increases
3. there is a delay in delivering the application into production

4. again we need to spend time on testing and certifying the additional utilities we added into our application

So to overcome the above problems in making the application production ready Spring Boot Actuator comes into picture.

Spring Boot Actuator is a prepackaged endpoints that are commonly required in monitoring and managing an application in production environment, which are build by the spring boot developers, that can be easily integratable into Spring Boot Application.

By using Actuator endpoints we can make an development application straight away ready for production deployment.

To enable or use actuator endpoints within our application, we just only need to add spring-boot-starter-actuator dependency in our project which will enable the endpoints by default

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

The actuator module has provided bunch of endpoints which we can use in monitoring our application

1. /info = provides the information about our application to the our partners/endusers like application name, author, version, features, licensing

2. /healthcheck = to check readyness and liveness of our application we use healthcheck

3. /env = to see all the environment variables configured or created on that machine

4. /beans = gives you all the bean definitions of the ioc container in our application

5. /configprops = all the properties we are using and injecting through @ConfigurationProperties can be accessed through /configprops

6. /threaddump = current jvm thread dump information

7. /metrics = memory, cpu utilization etc

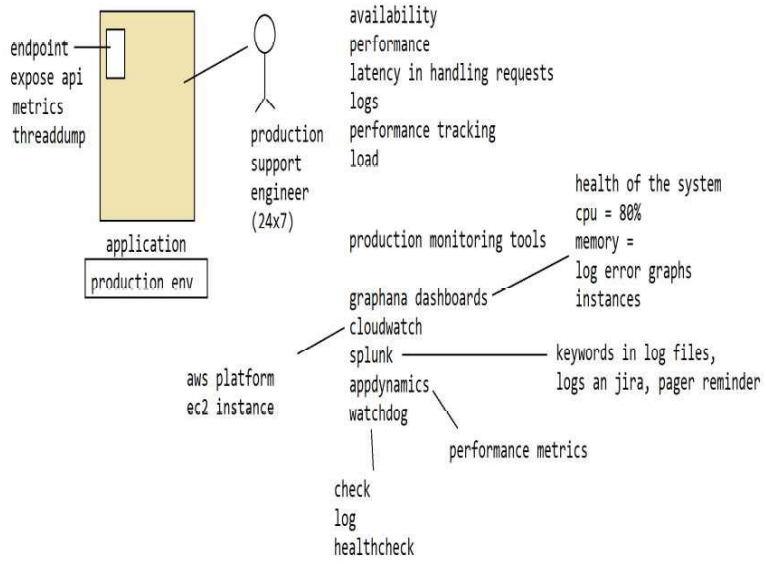
8. /loggers = displays the information about loggers and their configuration

9. /logfile = shows the application log

10. /shutdown = we can remote shutdown our application

11. /sessions = displays the http sessions that are there in our application

12. /conditions = the conditional annotations resulted to true based on which auto configurations are executed will be displayed by this endpoint



Spring boot actuator

What is spring boot actuator, why do we use it?

we can build production grade deployable application straight from the development by using acutuator module of spring boot.

actuator module provides pre-packaged/inbuilt endpoints which are required to enable our application to be deployed on production environment, it can be easily integratable in our application and can be exposed.

There are lot of actuator endpoints provided by the spring boot actuator module

1. /info = provides the information about our application

2. /health = to check the health of our application

3. /env = to see all the environment variables of our application/env

4. /beans = list down all the bean definitions within our ioc container

5. /configprops = all the configuration properties part of the environment object of our ioc container are listed

6. /threadump = shows the jvm threaddump

7. /metrics = memory, cpu usage metrics

8. /loggers = to see the loggers enabled and their logging levels

9. /logfile = shows the application log file

10. /shutdown = to shutdown the application remotely, by default disabled for security reasons

11. /sessions = shows the http sessions in our application

12. /conditions = shows the autoconfiguration conditions being passed

13. /caches = displays all the available caches

14. /httptrace = displays all the http trace information

15. /mappings = displays all the requestURL of our application which are exposed through @RequestMapping

16. /quartz = displays all the quartz beans

17. /startup = shows the startup steps data

we need to add spring-boot-starter-actuator dependency in to our pom.xml of the maven project to enable spring boot actuator

by default all of the above endpoints are accessible with a prefix /actuator/endpointName

These endpoints are exposed in 2 ways

1. jmx (java management extension) endpoints

jmx is a protocol usually used for administering and managing the jee application servers. we can use jmx api for connecting to the jee application server and manipulate or create mbean objects representing the application server objects and configure or modify application server.

Spring Boot Actuator has exposed all of the above endpoints using jmx api, where we can write jmx programs in accessing the endpoints exposed.

2. web (http) endpoints

In addition the actuator endpoints are even exposed as Http endpoints (Rest Api), where we can send a http request and can access them.

These endpoints to be accessed we need enable and expose them, unless exposed those are not going to be accessible. By default all the endpoints are enabled except shutdown endpoint due to security reason. In case if an endpoint is disabled, it means it is removed from the application.

we can enable an endpoint explicitly by writing an property in our application.yml or .properties

management.endpoint.endpointName.enabled=true/false

for eg.. if we want to enable shutdown endpoint we can do this by using the below property

management.endpoint.shutdown.enable=true

by default all the endpoints are enabled except shutdown, lets say we want to disabled all the endpoints and want to enable specific endpoints only we do this by using the below properties

management.endpoints.enabled-by-default=false // will disable all the endpoints

management.endpoint.info.enabled=true

management.endpoint.healthcheck.enabled=true

With the above we enabled the endpoints, but those are not exposed. Unless exposed those are not accessible. We can expose the endpoints using 2 technologies, either jmx or web endpoints.

By default all the endpoints are exposed through jmx technology, but when it comes to web endpoints there are only 2 endpoints exposed by default

1. info
2. healthcheck

if we want to expose web endpoints we need manually enable them, by using the below properties for each technology

management.endpoints.jmx.exposure.exclude=endpointnames (separated by comma)

management.endpoints.jmx.exposure.include=

management.endpoints.web.exposure.exclude=

management.endpoints.web.exposure.include=

for eg if we want to expose metrics and threaddump endpoints through web technology we can do the same as below

management.endpoints.web.exposure.include=metrics,threaddump

management.endpoints.web.exposure.include=* = all the endpoints are exposed

Endpoint cache

By default spring boot actuator caches the responses for read operations of the endpoints which doesnt take any parameters for 10 seconds of time to improve

performance, we can change the duration/interval cache by using below property

management.endpoint.endpointName.cache.time-to-live=10s

we can discover the endpoints that are exposed as part of our application by using the discovery page which is by default available as apart of /actuator

we can disable the discovery page using the below property

management.endpoints.web.discovery.enabled=false

Note: the web endpoints can be exposed only when we are working on web mvc application or webflux applications.

actuator endpoints

development to production grade deployable application can be build using actuator module of spring boot. actuator module has provided bunch of pre-build endpoints that can be integrated easily into our application and can be exposed in the production env which would help us in monitoring and managing the application.

to use actuator in our project we need to add starter dependency

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

In order to make an endpoint accessible we need to do 2 things

1. we need enable the endpoint = enabling an endpoint means including that endpoint as part of our application.

2. we need to expose the endpoint = making it accessible to the public

default:-

by default all the endpoints are enabled except /shutdown endpoint

How to enable an specific endpoint?

management.endpoint.endpointName.enable=true/false

how to disable all the endpoints?

management.endpoints.enable-by-default=false

exposing endpoint:

there are 2 ways we can expose an endpoint

1. jmx
2. web

by default all the endpoints are exposed over jmx, but only 2 endpoints are exposed over web.

1. healthcheck
2. info

how to access the endpoints?

we need to use the contextpath as /actuator/endpointName

how to include/exclude an endpoint to be exposed?

management.endpoint.technology.exposure.exclude=

management.endpoint.technology.exposure.include=

what is the default discovery page in accessing the endpoints list which are enabled and exposed?

/actuator

management.endpoints.web.discovery.enabled=false

the default cache interval for caching the read operations of the endpoints is 10sec

how can we change the default cache interval?

management.endpoint.endpointName.cache.time-to-live=20s

How to change the default contextpath /actuator?

management.endpoints.web.base-path=/management

Info endpoint:-

by default info endpoint will display blank information, as the information about our application is not known to Spring Boot it cannot generate info page for our application. we can seed in our application information through the below properties added in application.yml

spring.application.name=applicationName

info.author=sriman

info.version=1.0

info.released=10-10-2021

info.features=admin,updatepackage,reset

How to change the management endpoints port?

management.server.port=8088

with the above configuration, when we start our application 2 tomcat servers will be started deploying our application on server.port and actuator on management.server.port and are accessible separately

---

How to change the port on which management endpoints are exposed?

management.endpoints.server.port=8088

How to change the base-path over which we access the actuator endpoints?

management.endpoints.web.base-path=/management

The HealthCheck and Info endpoints are by default enabled by the spring boot actuator when we add them as dependency.

By default the HealthIndicator endpoints are autoconfigured and exposed by spring boot actuator

based on the starter dependencies we added into our project

For eg if we add spring-boot-starter-jdbc or spring-boot-datajpa into our project as starter dependencies, this indicates we are using database through DataSource, so spring boot actuator will autoconfigure DataSourceHealthIndicator endpoint

All the HealthCheck endpoints are implementing from HealthIndicator interface and Spring boot actuator has provided pre-defined implementations for different types of checks to be performed and enabled them through autoconfigurations based on dependencies we are using.

endpoint name	HealthIndicator ClassName
cassendra	CassandraDriverHealthIndicator
db	DataSourceHealthIndicator
diskspace	DiskSpaceHealthIndicator
mail	MailHealthIndicator
ping	PingHealthIndicator
mongo	MongoHealthIndicator
jms	JmsHealthIndicator

/actuator/health = webserver

/actuator/health/db = database

/actuator/health/diskspace = diskspace

to make the sub-components exposed we need to enable each of them and we need enable sub-components as well

management.endpoint.health.db.enabled=true

management.endpoint.health.show-components=always

management.endpoint.health.show-details=always

instead of verifying each the endpoints to check the total health of the application we can create health groups.

management.endpoint.health.group.airtelcare.include=db,diskspace,ping,mail

http://localhost:8081/actuator/health/airtelcare

Most of the time spring boot has provided several HealthIndicator implementations to perform various healthcheck on external resources of our application we are using, and even those are autoconfigured as well. But few times we want to create our own custom HealthIndicator implementations to check for external system dependencies we are using in our application.

For eg.. we might by using a payment gateway system like a RazorPay where we have an account and want to check for the status of the gateway and our account as well as part of healthcheck, for this spring boot has not provided any HealthIndicator implementation, so we need to create our own custom health indicator.

```
@Component("razorpay")
public class RazorPayHealthIndicator implements
HealthIndicator {
    private final String RAZOR_PAY_ACCOUNT_URL =
"http://razorpay.in/ac90383/status";

    @Autowired
    private RestTemplate restTemplate;

    public Health health() {
      ResponseEntity<String> response = null;
      response =
restTemplate.get(RAZOR_PAY_ACCOUNT_URL,
ResponseEntity.class);
      if(response.status == 200) {
        return Health.up().build();
      }
      return Health.down().build();
    }
}
```

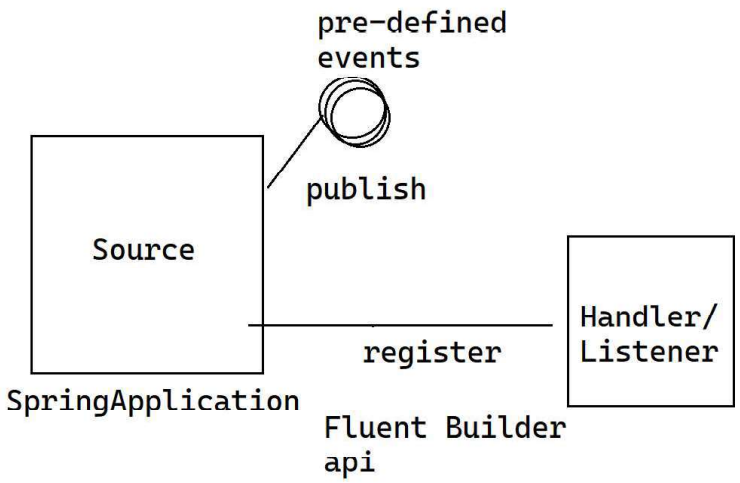
we can access the above endpoint with bean definition name

/actuator/health/razorpay

Info Endpoint:-

We can create our own Info endpoint also programmatically

```
@Component
class BootActuatorInfoContributor implements InfoContributor {
  public void contribute(Builder builder) {
    builder.withDetail("applicaiton.name", "boot-actuator").withDetail("author","Sriman").withDetail("version","1.0");
  }
}
```



HealthCheck Endpoints

The actuator module has provided several implementations of HealthIndicator interface to provide different levels of healthchecks of our application like

1. db
2. cassendra
3. mongo
4. mail
5. jms
6. diskspace
7. ping

etc

In order to use these endpoints we dont need to configure them as bean definitions, these are autoconfigured by spring boot autoconfigurations based on the starter dependencies we are using in our project.

by default the component level health endpoints are not available, we need to enable each of them and has to expose by writing properties in application.yml|properties

```
management.endpoint.health.db.enabled=true

management.endpoint.health.cassendra.enabled=true


management.endpoint.web.health.show-details: always
```

```
management.endpoint.web.health.show-components:
always
```

In case if we have any external resources we are dependent on and would like to monitor, then we need to write our own custom healthcheck endpoint

```
class RazorpayHealthIndicator implements
HealthIndicator {

  public Health health() {

    // write the logic for checking the external
resource based on that return the status

    return Health.up().build();

  }

}
```

How can we build our own Info Endpoint?

```
class AppInfoContributor implements InfoContributor {

  public void contribute(Info.Builder builder) {

    builder.withDetails("k1", "v1").build();

  }

}
```

In some cases we might want to add our own custom actuator endpoints for our application like

1. we might want to monitor the cache system we build in our application and want to perform evict or reload operations through actuator

2. we want to monitor the jms queue of our application want to know the statics of consumption

for such application requirements the spring boot actuator has not provided any in-built endpoints, we have to write our own actuator endpoints and need to expose.

if we are looking for building an administration, monitoring or management related endpoints for our application it is recommended to build them based on actuator api rather than rest endpoints because of the below.

1. we can enable/disable or expose the endpoints based on actuator echo system we build our custom endpoints through actuator api.

2.the endpoints we build will becom agnostic to the technology in which we are building those can be exposed both via jmx and http as well

3. we can easily secure them along with rest of the actuator endpoints by application the same security configuration

How to create our own custom actuator endpoints?

spring boot 2.0 has provided support for building custom actuator endpoints agnostic to the technology by providing bunch of annotations.

1. Endpoint(id="endpointName") = if we annotate any class which is configured as bean definition with @Endpoint annotation, it becomes an actuator endpoint and can be accessible with endpointid as url.

```
@Component
@Endpoint(id="cache")
class CacheEndpoint {
  @ReadOperation
  public int size() {

  }

  @DeleteOperation
  public void clearCache() {

  }

}
```

now we can add methods as part of our endpoint by using the below annotations

@ReadOperation = is equal to HTTP GET request

@WriteOperation = is equal to POST

@DeleteOperation = equal to DELETE

depends on the functionality we are writing as part of endpoint method, we need to choose appropriate annotations in exposing the method. The method declare in acuator endpoint class can take parameters and returns types but both of them should be primitive types only, to support multiple technologies.

```
@Component
@Endpoint(id="jmsqueue")
class JmsQueueManagementEndpoint {

  @ReadOperation
  public int length(String queueName) {
    return 39;
  }
  @WriteOperation
  public boolean pause(String queueName) {

  }
  @DeleteOperation
  public boolean disable(String queueName) {

  }
}
```

THE END

SPRING BOOT