Spring is java framework used to build java enterprise application. Spring Boot is bootstrap or quick start the spring application.

Spring Boot makes it easy to create stand-alone, production-grade, spring based Applications that you can “just run”.

Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring. Spring Boot takes an opinionated view of building production-ready applications.

# Spring

* Java Based Application Framework/Template.
* Programming and Configuration Model.
* Infrastructure support

# Some of its problems

* Huge Framework.
* Multiple setup step
* Multiple configuration
* Multiple build and deployment step.

Features of Spring Boot:

* **Opinionated**: It follows “**Opinionated** Defaults Configuration” Approach to reduce Developer effort. Due to **opinionated** view of **spring boot**, what is required to get started but also we can get out if not suitable for application. **Spring Boot** uses sensible defaults, “opinions”, mostly based on the classpath contents
* **Convention over configuration**: **Spring** has always favored **convention over configuration**, which means it takes up the majority of working uses cases into consideration and goes by it rather than picking an exact **configuration** and dependencies required for a specific application development.
* **Stand alone**: Spring boot comes with embedded tomcat, reduces the deployment and build configuration effort.
* **Production ready**: the out come of spring boot is ready to run.
* **Auto Configuration: Spring Boot autoconfiguration** represents a way to automatically **configure** a **Spring** application based on the dependencies that are present on the classpath. This can make development faster and easier by eliminating the need for defining certain beans that are included in the **auto**-**configuration** classes.
* **Dependence Starters: Version control of interdepended package is easy.**
* **In-Memory databases: like H2.**
* **Spring Boot provide Actuators and monitoring tools**

# Maven

Maven is dependence management tool and is used to build the application. Mention the dependence in pom.xml file and run below command

mvn clean install

It will download all the dependence from Remote Repo to local Repo .m2 using setting.xml file (Remote Repo URL is mentioned in setting.xml file).

# Creating a Spring Boot project

* Using simple Maven project:

STEP 1:

To convert Maven project to Spring boot project add the parent dependence. Our application will become the child for parent project its Maven child-parent inheritance. This spring parent project has all the features of Spring Boot.

<parent>

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

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

<version>2.1.11.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

Change the Java version using

<properties>

<java.version>1.8</java.version>

</properties>

NOTE: whenever we change the pom.xml we need make project update.

Maven > update project

Then add all the application dependent dependences. By adding web dependences, it import Spring MVC dependences which is used to build REST Service.

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

STEP 2: How to tell its spring application? Using @SpringBootApplication

How to tell Spring Boot this is start point application? Using @SpringBootApplication

How to make setup, run and start the application?

Create the servlet container and host application in that servlet container and make it available.

Using SpringApplication.*run*(KafkaExampleApplication.**class**, args);

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** KafkaExampleApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(KafkaExampleApplication.**class**, args);

}

}

SpringApplicatoin is static class which is having static method run. Perform following action.

1. **Sets up default configuration**. (convention over the configuration)
2. **Starts Spring application context**. Spring has containers that run on application Server.spring has containers for all application services(Business service, Data Services etc).This containers are called as application context. Every spring application has context, it context run as application runs.
3. Performs class path scan.
4. Starts the tomcat Server.

* Spring Initializr: Using Spring boot initializer website (spring.io)
* Spring CLI: it is used run Groovy script.

# Bill Of Materials

When new version of spring is released, and application need to be upgraded to new version. Just change in parent section corresponding application dependences which support to newer version will also change(combination of dependence what go well will be downloaded). This is called Bill of Materials.

# Embedded Servlet Container

* Conveniences
* Servlet Container config is now application config.
* Standalone application
* Useful for Microservice architecture

By default, Spring Boot use Tomcat as the default embedded server, to change it to Jetty, just exclude Tomcat and include Jetty like this :

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>

<dependency>

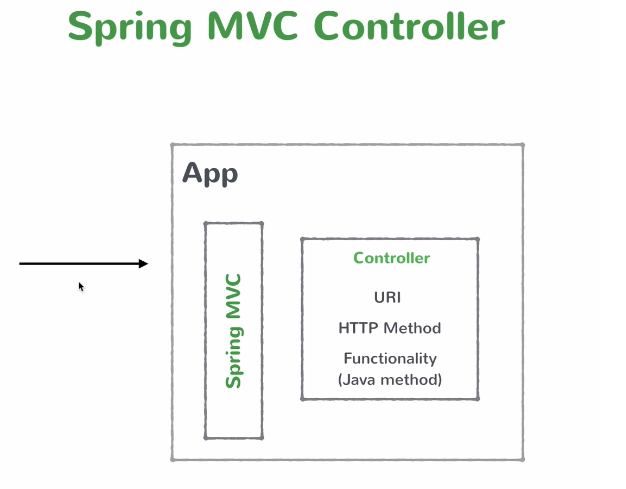
<groupId>org.springframework.boot</groupId>

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

</dependency>

# How Spring MVC Works

View Tire is handled by spring MVC, which help in building the controller and map request to responses.



# Creating a business service

Create class and annotate with @service, Business services are singleton scope, when application starts up spring creates a instance of this service and keeps in its memory and registries that service and this service are injected to depended class.

@Service is from Stereotype class.

import org.springframework.stereotype.Service;

Annotations:

@SpringBootApplication: If the class in annotated with @SpringBootApplication means its start point of Spring Boot application.

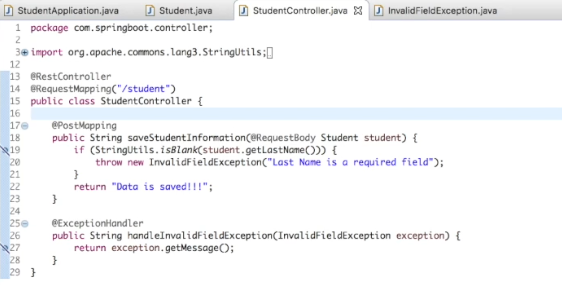
@RestController – makes class as Spring MVC web controller, Comes from web package. Its convince annotation that combines @Controller and @ResponceBody(is used sent response in JSON format)

@RequestMapping("/kafka") – Used to map the URL and exposed the resources and execute method below the requestmapping.

@Service – is used make class service.

@Autowired – is used to inject dependence

@ExceptionHandler: is used in controller method to catch the exception thrown by other method in controller. This will handle only controller level exception and its not best practices. To handle different exception we have write Handler method in respective controller, its not good. the solution for this is @ControllerAdvice



@ControllerAdvice: To elegantly handle exception in common place.

@RestControllerAdvice: Its convince annotation that combines @controllerAdvice and @ResponceBody

//@ControllerAdvice

@RestControllerAdvice

**public** **class** ExceptionHandlerControllerAdvice {

@ExceptionHandler(NoPersonException.**class**)

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

//@ResponseBody

**public** ExceptionResponse handlerNoPersonException(NoPersonException exception,HttpServletRequest request) {

String errorMessage =exception.getMessage();

String requestURI = request.getRequestURI();

StackTraceElement stack = exception.getStackTrace()[0];

ExceptionResponse er = **new** ExceptionResponse(errorMessage, requestURI, stack);

**return** er;

}

@ExceptionHandler(Exception.**class**)

@ResponseStatus(value=HttpStatus.***INTERNAL\_SERVER\_ERROR***)

//@ResponseBody

**public** ExceptionResponse handlerGenericException(Exception exception,HttpServletRequest request) {

String errorMessage =exception.getMessage();

String requestURI = request.getRequestURI();

StackTraceElement stack = exception.getStackTrace()[0];

ExceptionResponse er = **new** ExceptionResponse(errorMessage, requestURI, stack);

**return** er;

}

}

@Component: marks java class as bean so that at the component scan mechanism of spring can pick it up and pull it to the application context.

@Service: it behavious same as component and currently not provide any addition behavour, its best practice to use @service at service layer/Biasness layer, It specifics actual intent of class better.

@Repository: is used at the DAO layer and its also makes sure that unchecked exception thrown at the DAO layer is eligible to be translated into spring data access exception.

@Configuration: marks the class contains one or more Bean defined inside and its configuration class

@Bean: is used to explicitly declare single Bean rather spring doing automatically for us

@Value: is used to load custom properties and its to load individual property

@Value("${custom.properties:default values}")

String property;

@ConfigurationProperties: It allow you to map entire properties file to java object and can be used to handle complex properties structure.

@Autowired: Spring @Autowired annotation is used for automatic [dependency injection](https://www.journaldev.com/2394/java-dependency-injection-design-pattern-example-tutorial). [**Spring framework**](https://www.journaldev.com/16922/spring-framework) is built on [dependency injection](https://www.journaldev.com/2410/spring-dependency-injection) and we inject the class dependencies through spring bean configuration file.

We can use Spring @Autowired annotation for spring bean autowiring. @Autowired annotation can be applied on variables and methods for autowiring byType. We can also use @Autowired annotation on constructor for constructor based spring autowiring.

There are different ways through which we can autowire a spring bean.

1. autowire byName – For this type of autowiring, setter method is used for dependency injection. Also the variable name should be same in the class where we will inject the dependency and in the spring bean configuration file.
2. autowire byType – For this type of autowiring, class type is used. So there should be only one bean configured for this type in the spring bean configuration file.
3. autowire by constructor – This is almost similar to autowire byType, the only difference is that constructor is used to inject the dependency.

@Qualifier: annotation – This annotation is used to avoid conflicts in bean mapping and we need to provide the bean name that will be used for autowiring. This way we can avoid issues where multiple beans are defined for same type. This annotation usually works with the @Autowired annotation. For constructors with multiple arguments, we can use this annotation with the argument names in the method.

@Primary to give higher preference to a bean when there are multiple beans of the same type.

@Service

@Primary

**public** **class** Cat **implements** Animal {

}

@**Required** annotation in **spring** is a method-level annotation applied to the setter method of a bean property and thus making the setter-injection mandatory. This annotation indicates that the **required** bean property must be injected with a value at the configuration time.

Problem: Animal interface, Dog implements Animal and Cat implements Animal

If the interface have multiple implementation while creating the Binding it fails. Application fails to start.

@Autowired

Animal animal;

Error:

Field animal in com.example.sachin.RestServiceDemo.controller.AutowiredController required a single bean,

Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed

Possible Solution:

1. Add the @Qualifier

@Qualifier("dog")

@Autowired

Animal animal;

1. Type binding

@Autowired

Animal dog;

1. Setter Binding

Animal animal;

@Autowired

**void** setAnimal(@Qualifier("cat") Animal animal) {

**this**.animal = animal;

}

OR

@Autowired

**void** setAnimal(Animal dog) {

**this**.animal = dog;

}

1. Constructor binding

@Autowired

**public** AutowiredController(@Qualifier("dog") Animal animal) {

**super**();

**this**.animal = animal;

}

Construction injection are mandatory, where as setter injection are non mandatory.

@Conditional/Condition interface: to validate or check the some condition of Bean before loading in to application context. Used write condition to check or redefined property or we can write own custom condition inside it. If we have user defined logic associated with loading the bean , we can use conditional annotation in those scenario . It is used to develop an “If-Then-Else” type of conditional checking for bean registration.

@Conditional annotation is at more higher level when compared to @Profile annotation. @Profile annotation should be used for loading application configuration based on conditional logic.  
@Profile annotation is restricted to write conditional checking based on predefined properties. @Conditional annotation does not have this restriction.

**Difference between @Conditional and @Profile Annotations**

Both Spring @Profiles and @Conditional annotations are used to develop an “If-Then-Else” conditional checking. However, Spring 4 @Conditional  is more generalized version of @Profile annotation.

* Spring 3.1 @Profiles is used to write conditional checking based on Environment variables only. Profiles can be used for loading application configuration based on environments.
* Spring 4 @Conditional annotation allows Developers to define user-defined strategies for conditional checking. @Conditional can be used for conditional bean registrations.

## Spring Condition Interface

We need to have a class implementing an interface named Condition provided by Spring. It has a method matches and our conditional logic should go inside this method. Then we can use the class we have defined in the @Conditional annotation to check for a condition. Following the Condition interface that we need to implement in our custom condition class.

public interface Condition {

boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);

}

@CompontentScan({basePackage=”com.example.sachin.RestServiceDemo.controller “})

We need to use compontentscan annotation if package is not present in application parent package

# Using application properties

Spring has default way of working its very opinioned and had behavior of convention over configuration. It lets u less configuration by taking default configuration. We need coustomize configuration to make spring work way u want. The way of doing it is application.properties file.

Spring Boot support both application.properties and application.yml

Yml is superset of json and its human readable data serialization language and its commonly used for configuration and its elegant way of representing configuration.

<https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>

## Spring Boot Profiles for DEV and PROD Environments

<https://stackabuse.com/spring-boot-profiles-for-dev-and-prod-environments/>

### What are Spring Boot Profiles?

One of the core design principles behind Spring Boot is that it encourages convention over configuration. This means that the vast majority of app configurations use sensible default values that can be overridden when necessary, but in general a Spring Boot app will work out of the box with no custom configuration required.

However, usually some customization is necessary and oftentimes we need environment specific customization. This is where profiles come in handy. A profile in Spring Boot can be thought of as a context that defines a specific set of app settings, variables, and behaviors. Each time the app is built, the developer can specify which profile to use. If no profile is specified, the default will be used.

In this article, we will create a DEV profile and PROD profile to enable environment-specific configuration properties.

We can easily define the profiles by adding the following XML to the project's pom.xml file:

<profiles>

<profile>

<id>dev</id>

<properties>

<activatedProperties>dev</activatedProperties>

</properties>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

</profile>

<profile>

<id>prod</id>

<properties>

<activatedProperties>prod</activatedProperties>

</properties>

</profile>

</profiles>

Note the <activeByDefault>true</activeByDefault> tag, which means that the development profile will be used by default assuming no profile is specified at build time.

Profiles work in conjunction with Spring Boot properties files. By default, Spring Boot parses a file called application.properties – located in the src/main/resources directory – to identify configuration information.

Our first task will be to add a parameter in that file which will tell Spring to use a different environment-specific property file corresponding to the active profile (i.e. the profile that the app is currently being run with). We can do this by adding the following to the application.properties file:

spring.profiles.active=@activatedProperties@

Now we need to create the two new environment-specific property files (in the same path as the existing application.properties file), one to be used by the DEV profile and one to be used by the PROD profile. These files need to be named the following:

* application-dev.properties
* application-prod.properties

Within each of these files, properties can be defined that will only be applied when the corresponding profile is active.

In order to make this concept clearer, let's consider a real-world example involving Google Analytics and Google Tag Manager configuration on a Spring Boot site, which I'll go over in detail in the next few sections.

### Specifying the Profile at Build Time

When building the application for production, we need to specify the PROD profile is to be used so that the real tracking ID values are used. Here is how this is done using Maven as the build tool:

$ mvn -Pprod clean install

The key to this command is the -P flag, which is used to specify the profile to use for the build.

If we want to set the profile after the code is built, we can use a Java VM argument at application launch. This is done as follows:

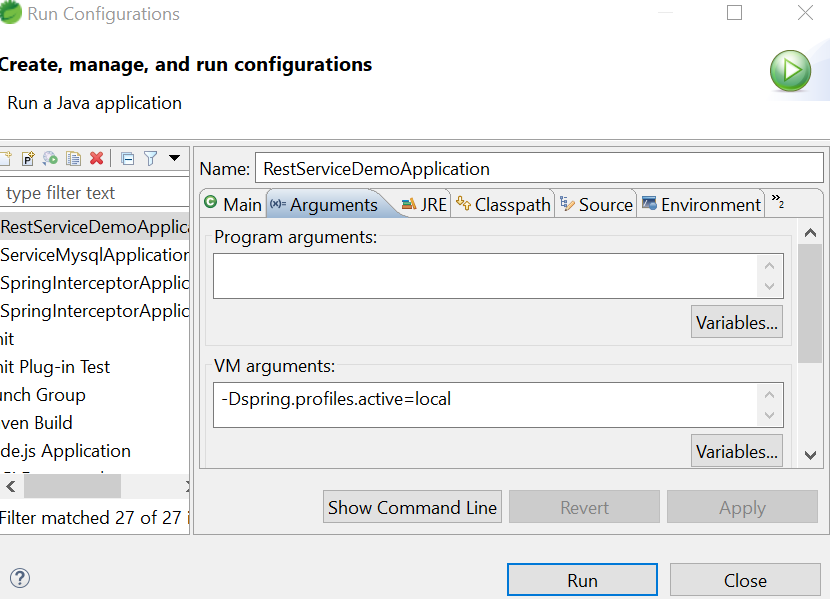
$ java –jar -Dspring.profiles.active=prod app.jar

Alternatively, the profile can be directly specified in the application.properties file by adding the line:

spring.profiles.active=prod

In each case, we specify prod as the active profile, which causes the application-prod.properties file to be chosen for configuration purposes. Since this file contains the real values for the tracking IDs, those will be inserted into the templates in the production build and user events will be successfully tracked on the Google platform.

In STS we can have arguments



@Service

@Profile(“prod”)

Class Prodconfig{}

Application context create above bean only in production env

Creating Spring Boot Application:

Following are step:

1. Declare parent : spring boot conversion over configuration , which help in make configuration default. That will come as part as parent jar.

<parent>

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

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

<version>1.5.1.RELEASE</version>

</parent>

1. Add the application related dependences.

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

1. Add the properties to specify version.

<properties>

<java.version>1.8</java.version>

</properties>

1. Create the class with main method.

@SpringBootApplication //tell spring boot is start for application and this is SB app

**public** **class** Demo {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

SpringApplication.*run*(Demo.**class**,args);

//it creates servlet container and run this

}

}

* Set up default configuration.
* Starts Spring application context.
* Performs class path scan.
* Starts tomcat server.

1. Add the controller.

@RestController // makes the class rest controller

**public** **class** TopicController {

@RequestMapping("/hello") // wherever there is request for /hello call method n return

String home() {

**return** "Hi sachin!";

}

}

@RequestMapping(value = "/saveCustomer", method = RequestMethod.***POST***)

**void** saveCustomer(@RequestBody Customer cust) {

System.***out***.println(cust);

Customer cs = **new** Customer("sachin"," Varthur","Gungur","Bagalkot");

//customerRepositoryCustom.saveCustomer(cs);

mongoTemplate.save(cs,"customer");

}

How request is mapped with POJO?

That is taken care by Spring MVC

How response is converted to JSON?

That is also taken care by Spring MVC

How embedded tomcat server work?

* Conveniences
* Servlet container config is now application config.
* Standalone application.
* Useful for microservices architecture.

What happening here?

Bill of materials

Spring boot internally works on Spring MVC principal.

Resources : REST API deals with entities or Resources , Resources is thing in your Application.

Service :

@Service // makes class as service

**public** **class** TopicService {

**private** List<Topics> topics = **new** ArrayList<>(Arrays.*asList*(

**new** Topics("Spring","Spring","Is a framwork"),

**new** Topics("java","java","Is a language")));

**public** List<Topics> getTopics(){

**return** topics;

}

**public** Topics getTopicById(String id){

**return** topics.stream().filter(t -> t.getName().equals(id)).findFirst().get();

}

**public** **void** updateTopics(Topics topic, String id) {

// **TODO** Auto-generated method stub

**for**(**int** i =0 ; i<topics.size(); i++){

Topics tp = topics.get(i);

**if**(tp.getName().equals(id)){

topics.set(i, topic);

**return**;

}

}

}

**public** **void** deleteTopics(String name) {

// **TODO** Auto-generated method stub

**for**(**int** i =0 ; i<topics.size(); i++){

Topics tp = topics.get(i);

**if**(tp.getName().equals(name)){

topics.remove(i);

**return**;

}

}

}

}

@RestController

**public** **class** TopicController {

@Autowired // tells spring this class is dependent this call we need create instance of the same.

**private** TopicService topicservice;

@RequestMapping("/topics")

**public** List<Topics> getTopics(){

**return** topicservice.getTopics();

}

@RequestMapping("/topic/{id}")

**public** Topics getTopicById(@PathVariable String id){

**return** topicservice.getTopicById(id);

}

@RequestMapping(method=RequestMethod.***POST***,value="/add/topic")

**public** **void** addTopics(@RequestBody Topics topic){

//Topics topics = new Topics(".net",".net","sjdsdjshd");

topicservice.addTopics(topic);

}

@RequestMapping(method=RequestMethod.***PUT***,value="/update/topic/{id}")

**public** **void** updateTopics(@RequestBody Topics topic, @PathVariable String id){

//Topics topics = new Topics(".net",".net","sjdsdjshd");

topicservice.updateTopics(topic, id);

}

@RequestMapping(method=RequestMethod.***DELETE***,value="/delete/topic/{name}")

**public** **void** deleteTopics(@PathVariable String name){

//Topics topics = new Topics(".net",".net","sjdsdjshd");

topicservice.deleteTopics(name);

}

}

1. Different way of creating Spring-boot Application?

* Spring Initializr(start.spring.io)
* Using Maven project
* Spring boot CLI (uses Groovy script)
* STS

1. Configuration Spring Boot. (application.properties)

<https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>

# What is JPA

Database Tier, It enables the ORM work with relational DB (Hibernate). JPA is way to use ORM in spring boot.

Spring Boot with Data Tier:

What is JAP? (Java Persistence API) which allows to do ORM (Object Relational Mapping) with SQL. JAP is way to use ORM.

Spring Data JPA.

* Add the dependency:

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

* Make POJO making to table by @Entity annotation and @Id
* Create the Repository interface which extends CrudRepository<Topic, String>

Topic: Table name,

String: Id data typse.

* In service Class Inject Repository by @Autowired and call inbuild methods.

# Adding Entity Relationship and Extending Repository

Make sure always write methods name in camel case.

Method mapping with db query:

Syntax: findBy<property\_name>(),

Select 🡪 findAll(),

findById(),

findByDescription(),

findByTopicId().

Connection of external DB:

Mapping relation between the two table.

Topics have many Courses.

@ManyToOne – in course class create property of topic object and make it ManyToOne relationship.

@OneTo Many – In Topic class create course property as object and make it OneToMany relationship.

# Packaging and running a Spring Boot app

To deploy Spring boot no external servlet container is required. Because it bundles in servlet container and its application configuration.

Building and deploy the Spring Boot: (Jar /war) -> change in pom.xml

(Maven build) : mvn clean install

Run: java -jar target/<project.jar>

# Spring Boot Actuator

Monitoring Spring Boot: (server health)

Add the dependency of actuator.

Actuator: group of binary added to classPath and add the features to application

Url: <http://localhost:8080/health>

To change the port

Management.port=9090

/bean

/info

<https://spring.io/guides/gs/actuator-service/>

# Spring Boot – Interceptor

Filter its a interface will run between request and run before rendering the view, Its related to Servlet API, Servlet filter is used in web layer only and we cant use outside web context. It can used for authication and limiting the request count per second/hr or person

**import** javax.servlet.Filter;

**import** javax.servlet.FilterChain;

**import** javax.servlet.FilterConfig;

@Component

@Order(1)

**public** **class** RequestFilter **implements** Filter {

@Override

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {

System.***out***.println("Global filter");

}

@Override

**public** **void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)

**throws** IOException, ServletException {

System.***out***.println("Done of Global filter");

filterChain.doFilter(servletRequest, servletResponse);

System.***out***.println("End of Global filter");

}

@Override

**public** **void** destroy() {

}

**import** org.springframework.boot.web.servlet.FilterRegistrationBean;

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

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

**import** com.example.lifeSight.component.ApiRequestFilter;

@Configuration

**public** **class** ApiFilterConfig {

@Bean

**public** FilterRegistrationBean<ApiRequestFilter> apiFilterBean(){

FilterRegistrationBean<ApiRequestFilter> apiFilter = **new** FilterRegistrationBean<>();

apiFilter.setFilter(**new** ApiRequestFilter());

apiFilter.addUrlPatterns("/createPixel/\*");

**return** apiFilter;

}

}

Interceptor run between the request. Its spring specific concept, it can be used outside web context or anywhere, for security stuff in biasness layer or logging we use interceptor

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** org.springframework.lang.Nullable;

**import** org.springframework.stereotype.Component;

**import** org.springframework.web.servlet.ModelAndView;

**import** org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

@Component

**public** **class** IntersceptorHandler **extends** HandlerInterceptorAdapter{

// it will handle before hitting the controller

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

System.***out***.println("preHandle");

**return** **true**;

}

/\*\*

\* This implementation is empty. after returning from controller

\*/

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

@Nullable ModelAndView modelAndView) **throws** Exception {

System.***out***.println("postHandle");

}

/\*\*

\* This implementation is empty. is handled after response is sent and view is render

\*/

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,

@Nullable Exception ex) **throws** Exception {

System.***out***.println("afterCompletion");

}

}

**import** org.springframework.beans.factory.annotation.Autowired;

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

**import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

**import** com.example.springinterceptor.component.IntersceptorHandler;

@Configuration

**public** **class** InterceptorConfiguration **implements** WebMvcConfigurer{

@Autowired

IntersceptorHandler intersceptorHandler;

/\*\*

\* Add Spring MVC lifecycle interceptors for pre- and post-processing of

\* controller method invocations and resource handler requests.

\* Interceptors can be registered to apply to all requests or be limited

\* to a subset of URL patterns.

\*/

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(intersceptorHandler);

}

}

# Spring Boot - Multiple Yaml Files

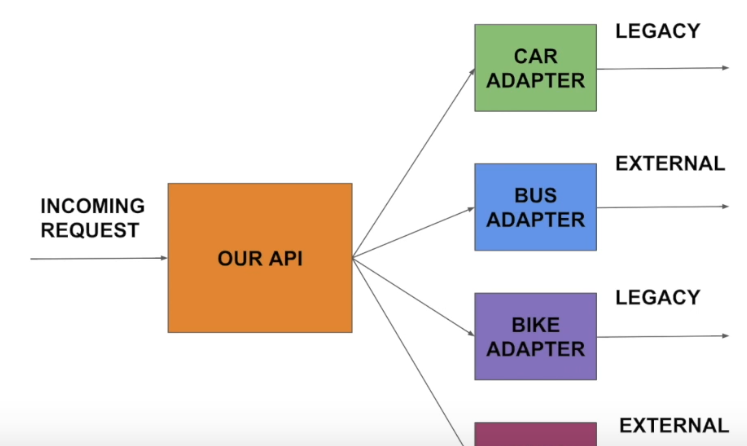
Yes we can have multiple yml file.

spring.profiles.include=complex

create file application-complex.yml

# ServiceLocatorFactoryBean

Based on some type we need call different system may be LEGACY or EXTERNAL. We can achive it by AdaptorService



# RequestBodyAdvice, ResponseBodyAdvice

It used if we want do activities away from controllers and services layer,

It can be used for duplicate check and logging incoming request

# Spring boot – CommandLineRunner interface

Spring boot’s [CommandLineRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html) interface is used to run a code block only once in application’s lifetime – after application is initialized.

@SpringBootApplication

public class Application implements CommandLineRunner {

private static final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

@Autowired

DataSource dataSource;

@Autowired

CustomerRepository customerRepository;

public static void main(String[] args) throws Exception {

SpringApplication.run(Application.class, args);

}

@Transactional(readOnly = true)

@Override

public void run(String... args) throws Exception {

System.out.println("DATASOURCE = " + dataSource);

System.out.println("\n1.findAll()...");

for (Customer customer : customerRepository.findAll()) {

System.out.println(customer);

}

System.out.println("\n2.findByEmail(String email)...");

for (Customer customer : customerRepository.findByEmail("222@yahoo.com")) {

System.out.println(customer);

}

System.out.println("\n3.findByDate(Date date)...");

for (Customer customer : customerRepository.findByDate(sdf.parse("2017-02-12"))) {

System.out.println(customer);

}

// For Stream, need @Transactional

System.out.println("\n4.findByEmailReturnStream(@Param(\"email\") String email)...");

try (Stream<Customer> stream = customerRepository.findByEmailReturnStream("333@yahoo.com")) {

stream.forEach(x -> System.out.println(x));

}

System.out.println("Done!");

exit(0);

}

}

<https://www.youtube.com/watch?v=bNqD5P-huJc&index=23&list=PLmbC-xnvykcghOSOJ1ZF6ja3aOgZAgaMO&spfreload=1>

<https://start.spring.io/>

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation-first-steps>

**Spring Boot: Common application properties**

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

java –jar <jar file>

Rest Template:

<https://howtodoinjava.com/spring/spring-restful/spring-restful-client-resttemplate-example/>

Configuration Class

@SpringBootApplication

@Bean

SpringApplication.run(Configuration.class,arg[]);

Controller

@RestController

@RequestMapping(method=Request.

Java 8 lambda basics

Spring :

* Its application framework. Used to build java based application.
* Spring is programming and configuration model.
* Infrastructure support.

Corns : huge framework., Multiple setup step and configuration steps., Multiple build and deploy steps.

Can we abstract these steps? That is spring boot.

Spring Boot:

Java Dependences:

Maven : is build and dependences management tool , it allows to declare all the dependencies in single file(pom.xlm) and all the jar are added to classpath of application.

Note: Per-set of combination of jar version is called Bill of materials.

# Async query result in Spring data JPA

@Async - annotating a method of a bean with @Async will make it **execute in a separate thread** i.e. the caller will not wait for the completion of the called method.

Let's start by **enabling asynchronous processing** with **Java configuration** – by simply adding the @EnableAsync to a configuration class:

@Configuration

@EnableAsync

public class SpringAsyncConfig { ... }

@Async

public Future<String> asyncMethodWithReturnType() {

    System.out.println("Execute method asynchronously - "

      + Thread.currentThread().getName());

    try {

        Thread.sleep(5000);

        return new AsyncResult<String>("hello world !!!!");

    } catch (InterruptedException e) {

        //

    }

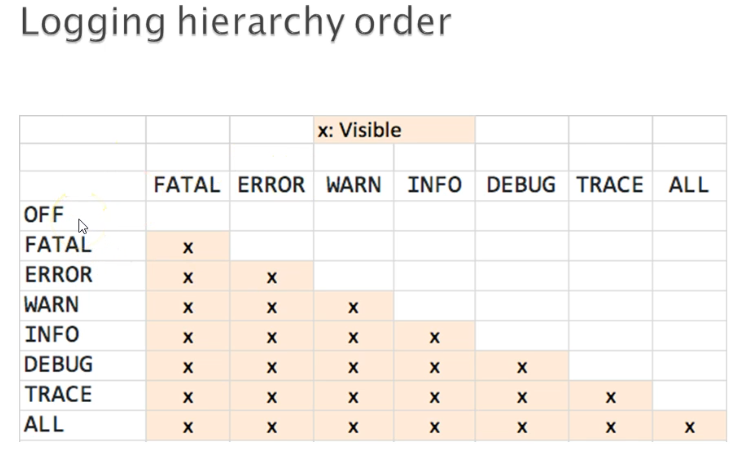
    return null;

}

Future<String> future = asyncAnnotationExample.asyncMethodWithReturnType();

# Default Logging Configuration in Spring Boot

Spring-boot-starter-logging – is default come with SLA4j + logback



Default is INFO logging level

#Logging properties

logging.level.root=info

//we can set logging level to root of application

logging.level.com.example.sachin.RestServiceDemo.controller=info

//we can set logging level to class and package level also.

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n

%d- is for data formate

%msg – to print the msg for log

%n – for new line

[%thread] – to diplay thread name

%level – to display level of log(INFO,DEBUG etc)

%logger{36} – to display the logger class name

logging.path=logs

//to redirect to log location

logging.file=logs/app.log

// is used to redirect the log and specify the log name

# Spring Boot Custom logging using logback

We have create logback.xml and specify logging level and pattern in it.

# Spring Boot Data JPA Junit Testing

# Writing Integration Tests for Rest Services with Spring Boot

Using testRestTemplate.exchange()