Spring Boot

Stand-alone, production-grade Spring application with minimum configurations

– Avadhut

About Me

- I'm Avadhut!
- Working as Senior Integration Engineer and consultant
- Open-source enthusiast
- Worked in Java since Java-1.5
- Active community member for couple of open-source projects
- Worked with Integration, Fuse, Camel, Karaf, Kafka and messaging platforms for quite a bit
- Did full production deployments, architecture review and performance tuning for couple of employers and lot of Red Hat customers

** You can find me on:

Official Website: https://kodtodya.github.io/talks

Twitter: https://twitter.com/kodtodya

Pre-Requisite for Spring-Boot Course

- Mandatory knowledge of Java as well as overall programming (We will use Java-11 for this course)
- Mandatory knowledge of Spring framework(any version)
- Linux(Any flavor) and Mac are strongly preferred, avoid Windows if possible
- Willingness to learn an awesome technology.

Who is this course for?

- **Developer:** who would like to learn how to write and run an application that leverages features of Spring-Boot

- **Architects:** who want to understand the role of Spring-Boot in the enterprise pipeline and how it can fit in existing application stack

- **DevOps:** who want to understand how Spring-Boot works with regards to application deployment and it’s troubleshooting

Welcome...!!!

- A warm welcome to Spring Boot - 2.x course..

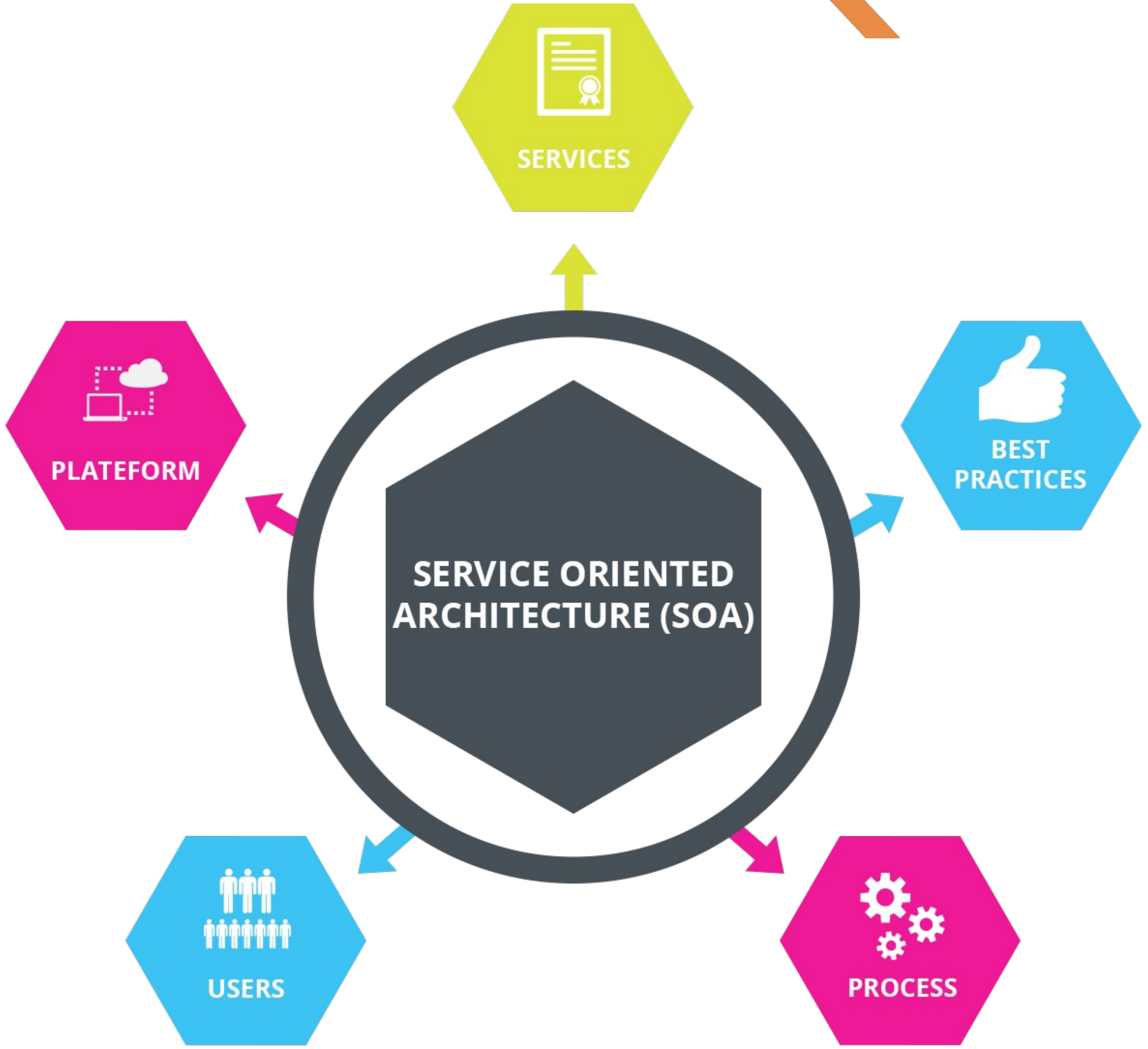
Course Structure

- **Part – 1 : Fundamentals**
  - Service Oriented Architecture (SOA)
  - Micro-services
  - Spring Boot and why Spring boot?
  - Bootstrapping
  - Logging
  - Embedded Tomcat and it’s need
  - Embedded Tomcat configuration
  - Code Structure
- **Part – 2 : Coding time**
  - Runners
  - Application Properties
  - Spring Boot Annotations
- **Part – 2 : Coding time**
  - Actuator
  - Interceptor
  - Internationalization
  - Servlet Filter
  - CORS Support
  - Spring Boot Exception Handling API
  - Scheduling
- **Part – 3 : Administration**
  - Eureka Server
- **Part – 4 : Testing**
  - Unit Testcase

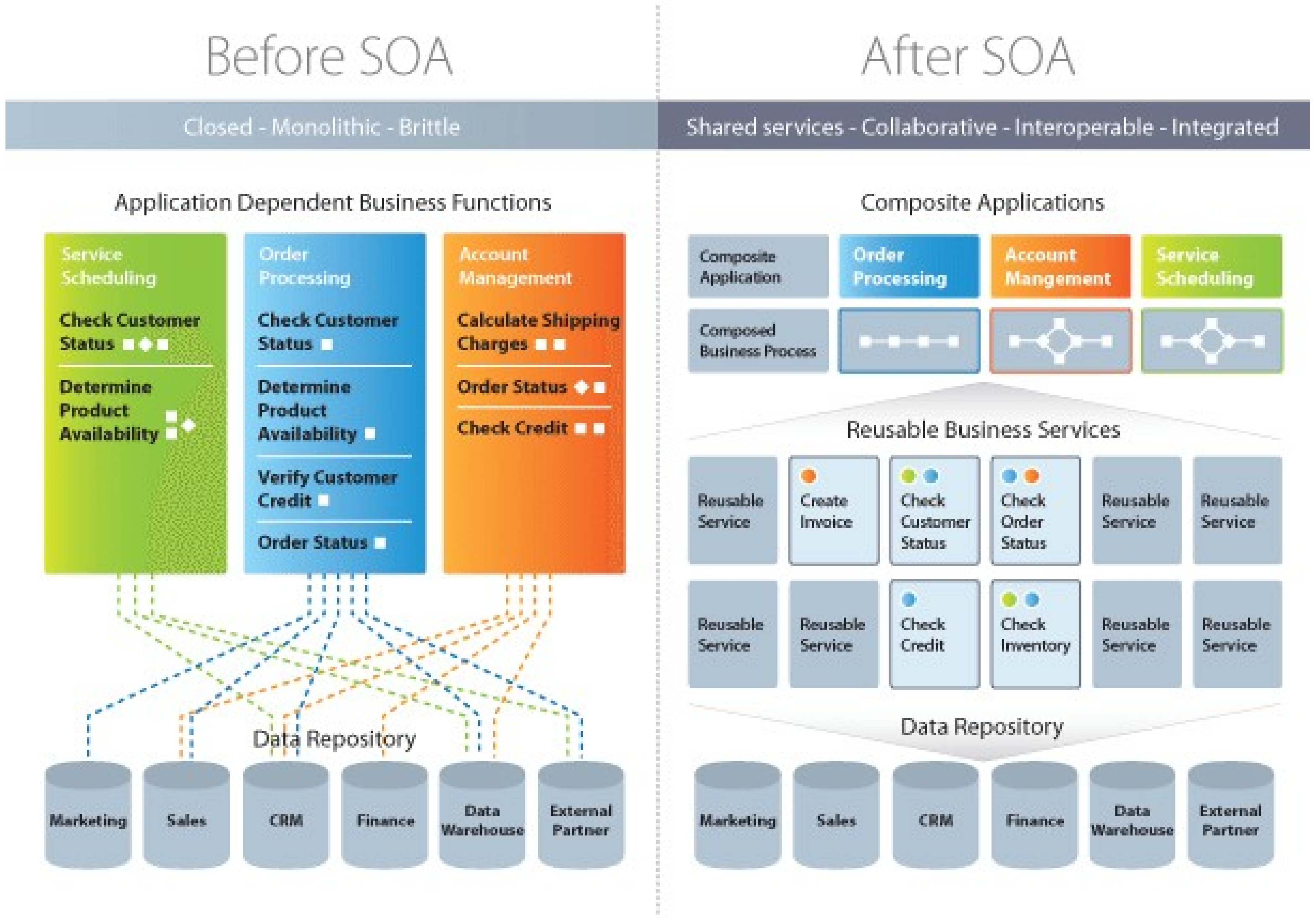
Spring Boot – 2.x

Part-1 | Fundamentals

Service Oriented Architecture (SOA)

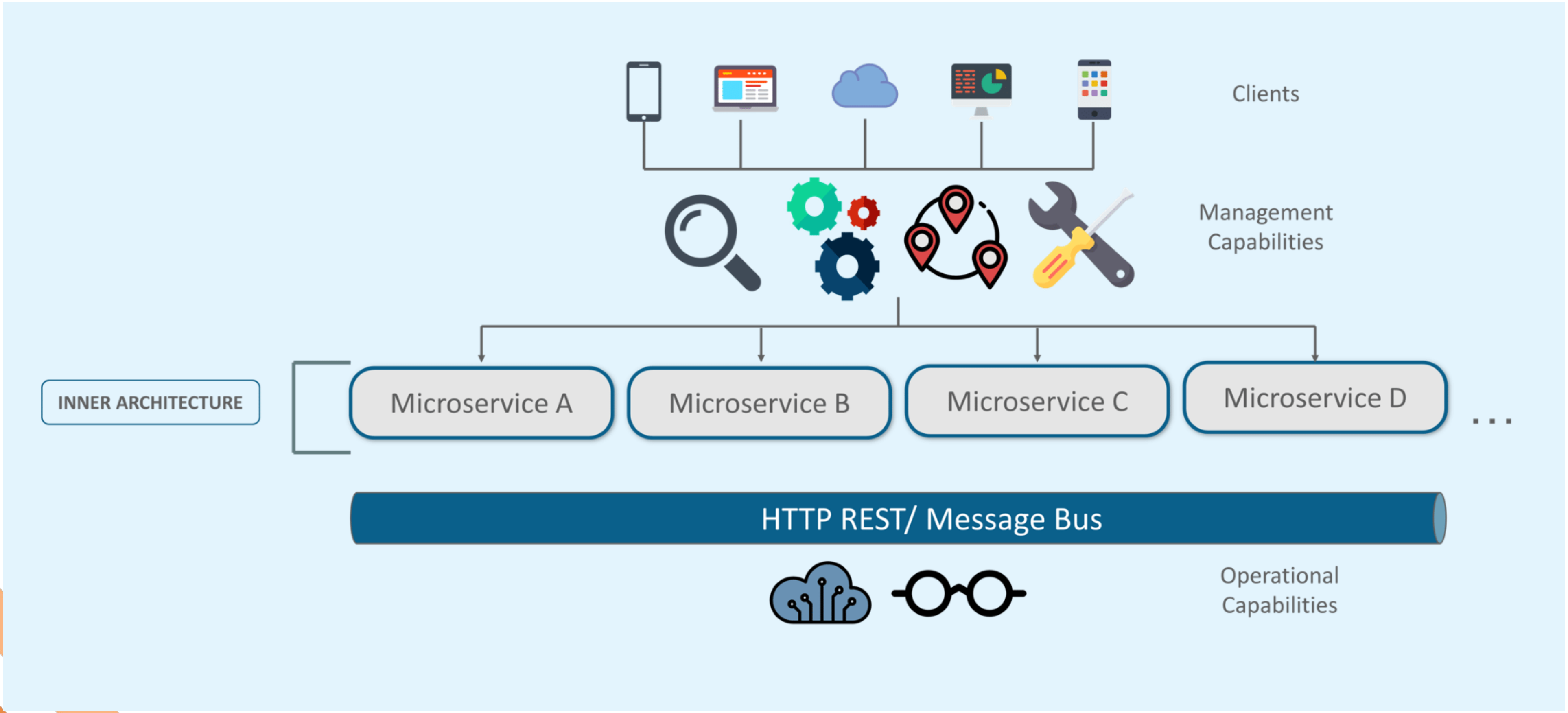


Service Oriented Architecture (SOA)

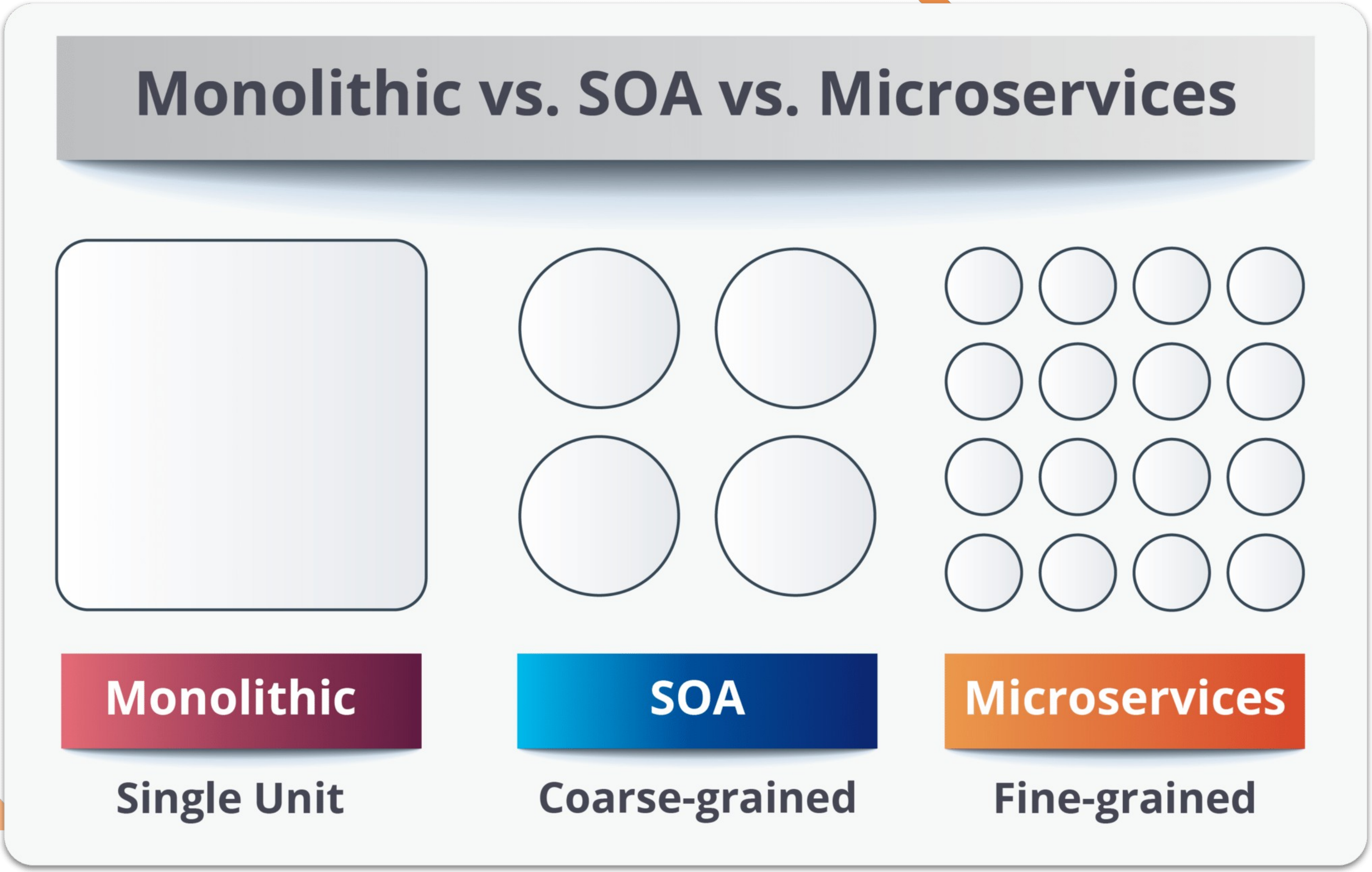


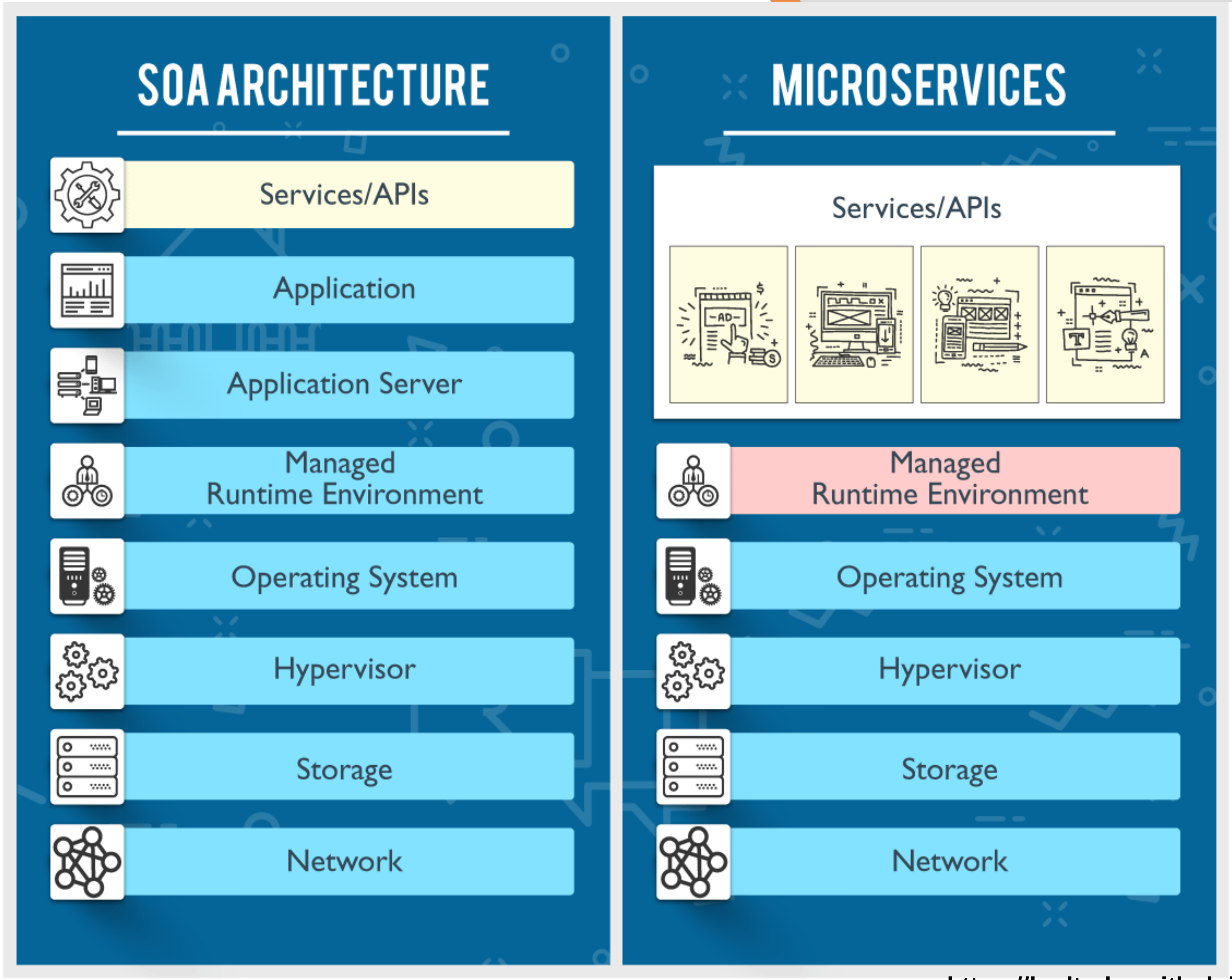
Micro-services Architecture

- Micro-services



SOA vs Micro-services




SOA ARCHITECTURE
Services/APIs
Application
Application Server
Managed Runtime Environment
Operating System
Hypervisor
Storage
Network
MICROSERVICES
Services/APIs
Managed Runtime Environment
Operating System
Hypervisor
Storage
Network

SOA vs Micro-services

SOA	Micro-services
Follows "**share-as-much-as-possible**" architecture approach	Follows "**share-as-little-as-possible**" architecture approach
Importance is on **business functionality** reuse	Importance is on the concept of "**bounded context**"
They have **common governance** and **standards**	They focus on **people, collaboration** and freedom of other options
Uses **Enterprise Service bus (ESB)** for communication	Simple messaging system
They support **multiple message protocols**	They use **lightweight protocols** such as **HTTP/REST** etc.
Multi-threaded with more overheads to handle I/O	**Single-threaded** usually with the use of Event Loop features for non-locking I/O handling
Maximizes application service reusability	Focuses on **decoupling**
Traditional Relational Databases are more often used	**Modern Relational Databases** are more often used
A systematic change requires modifying the monolith	A systematic change is to create a new service
DevOps / Continuous Delivery is becoming popular, but not yet mainstream	Strong focus on DevOps / Continuous Delivery

Spring-Boot

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

...create …. Spring based Applications ….

…. stand-alone & production-grade ….

Spring-Boot

Advantages

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

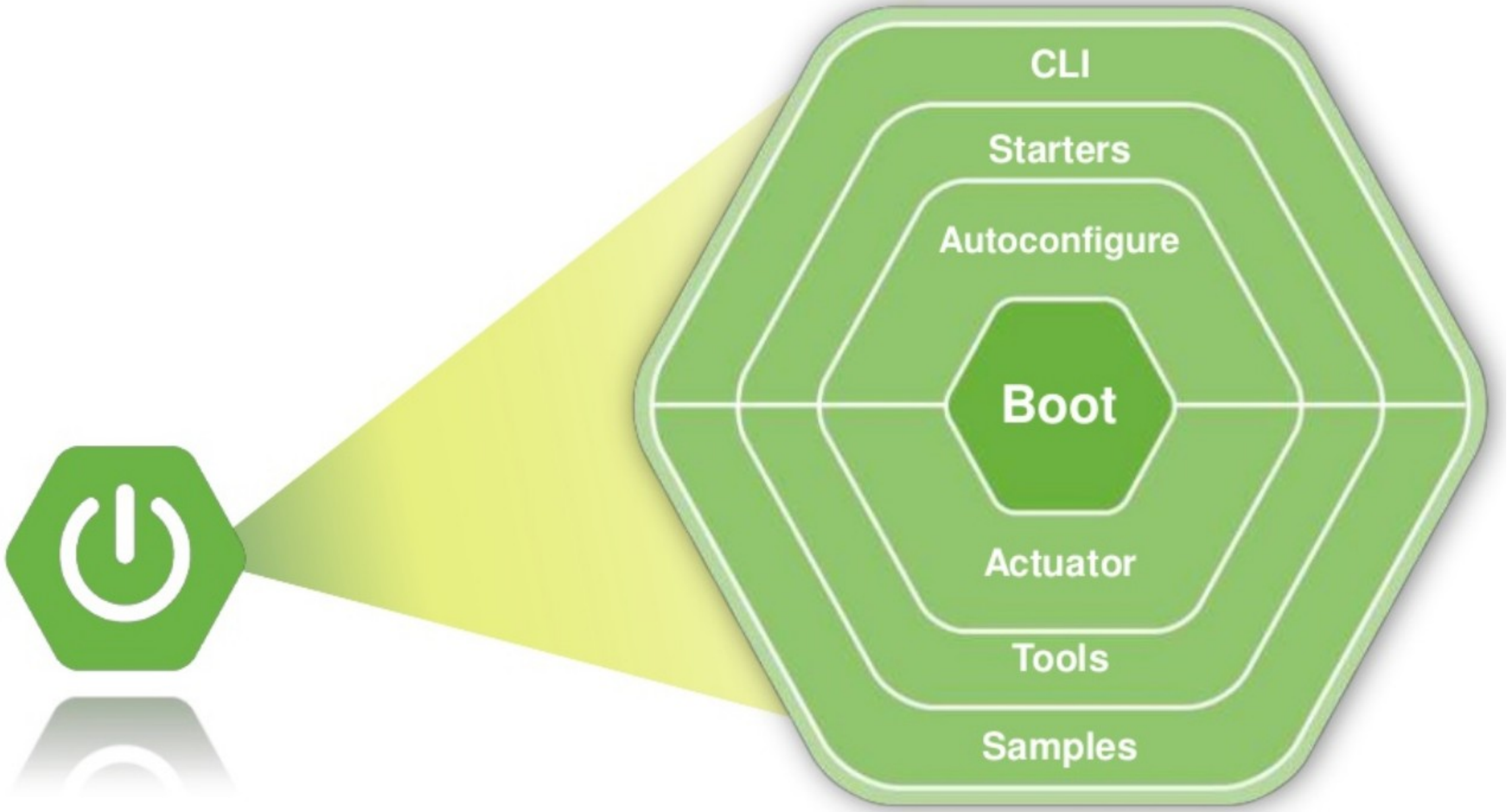
Goals

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application

Why Spring-Boot

- **It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.**
- **It provides a powerful batch processing and manages REST endpoints.**
- **In Spring Boot, everything is auto configured; no manual configurations are needed.**
- **It offers annotation-based spring application**
- **Eases dependency management**
- **It includes Embedded Servlet Container**

Spring-Boot Modules



How does it work?

- Spring Boot automatically configures your application based on the dependencies you have added to the project by using @EnableAutoConfiguration annotation.
- For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.
- The entry point of the spring boot application is the class contains @SpringBootApplication annotation and the main method.
- Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation.

Spring-Boot starter

Spring Boot Starters are the dependency descriptors.

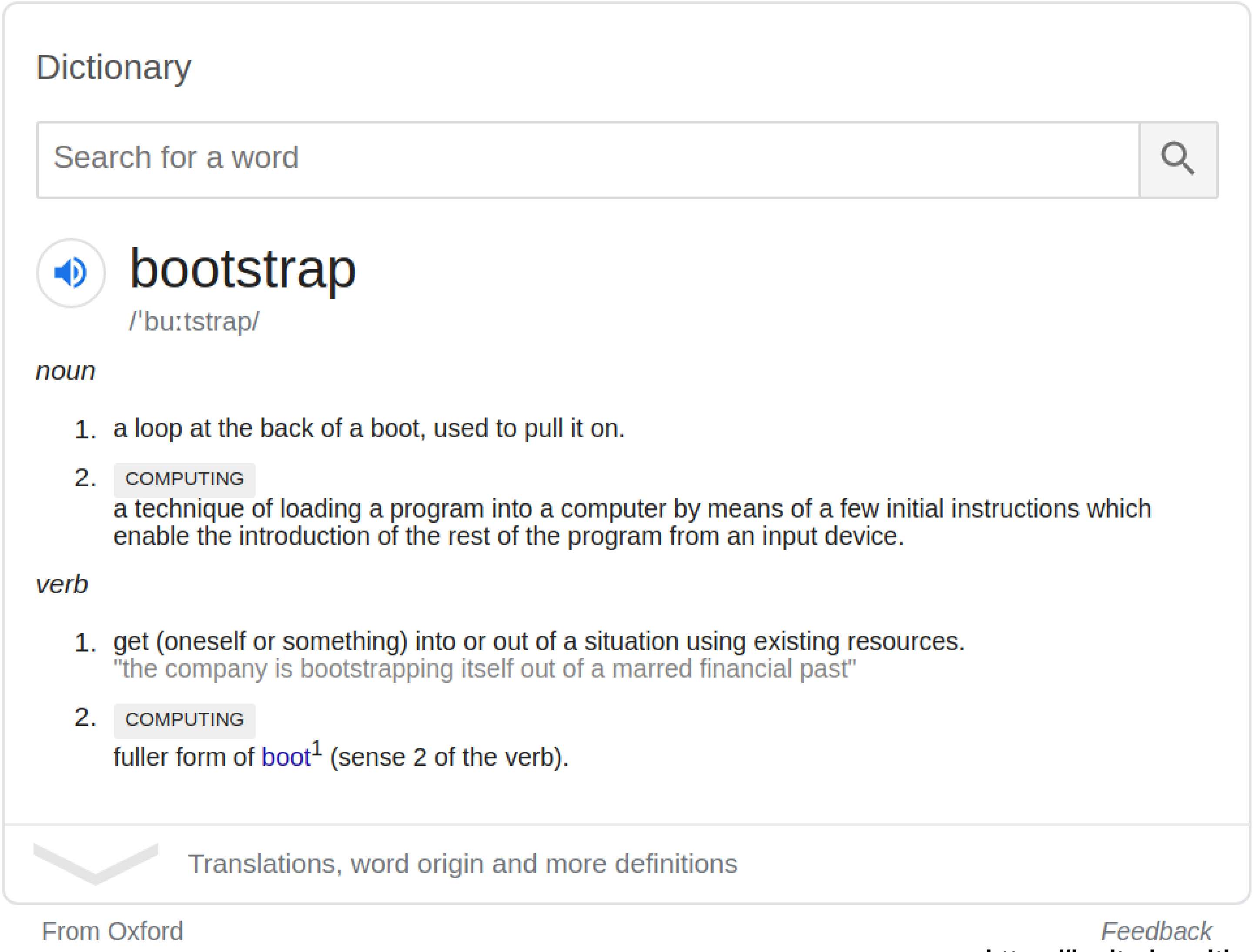
Handling dependency management is a difficult task for big projects. Spring Boot resolves this problem by providing a set of dependencies for developers' convenience.

Spring Boot provides a number of starters that allow us to add jars in the classpath. Spring Boot built-in starters make development easier and rapid.

In the Spring Boot Framework, all the starters follow a similar naming pattern: spring-boot-starter-*, where * denotes a particular type of application.

e.g. JPA -> spring-boot-starter-data-jpa
JMS -> spring-boot-starter-data-jms

Bootstrapping of Spring Application using Spring-Boot



Dictionary

Search for a word

bootstrap

/ˈbuːtstrap/

noun

1. a loop at the back of a boot, used to pull it on.
2. COMPUTING
   a technique of loading a program into a computer by means of a few initial instructions which enable the introduction of the rest of the program from an input device.

verb

1. get (oneself or something) into or out of a situation using existing resources.
   "the company is bootstrapping itself out of a marred financial past"
2. COMPUTING
   fuller form of boot^1 (sense 2 of the verb).

Translations, word origin and more definitions

From Oxford

Feedback

Bootstrapping of Spring Application using Spring-Boot

There are few ways to bootstrap spring-boot application

- **Using Maven**
- **Using Spring Initializrs**

Refer to https://start.spring.io/ for Spring Initializr.

Bootstrapping with Maven

Couple of steps in maven will make your application as spring-boot application.

1. Add below parent dependency to your pom.xml

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
</parent>
```

2. Add below dependency to your pom.xml

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-<subsystem></
artifactId>
</dependency>
```

3. Make sure your build section in pom.xml contains below plugin

```
<plugins>
        <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
</plugins>
```

4. Write your Main class as like below code-snippet

```
@SpringBootApplication
public class SpringBootLoggingApplication {


   public static void main(String[] args) {
      SpringApplication.run(
                SpringBootLoggingApplication.class, args);
   }


}
```

Logging

Spring Boot uses Commons Logging for all internal logging but leaves the underlying log implementation open. Default configurations are provided for Java Util Logging, Log4J2, and Logback. In each case, loggers are pre-configured to use console output with optional file output also available.

By default, if you use the “Starters”, Logback is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.
Below are the different types of logging in Spring Boot:

- Zero config logging
- Logback configuration logging
- Log4j2 Configuration Logging
- Log4j2 Without SLF4J
- Logging with Lombok
- @Slf4j and @CommonsLog
- @Log4j2
- JANSI on Windows
- Beware of Java Util Logging

How to use Logback Configuration Logging

- **Add one of the below log configuration file classpath (src/main/resources)**

logback.xml	logback-spring.xml
logback.groovy	logback-spring.groovy

- **Sample Configuration**

```
<appender name="STDFILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logs/spring-boot-logging-app.log</file>
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
        <Pattern>
            %d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level %logger{36}.%M - %msg%n
        </Pattern>
    </encoder>


    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>
            logs/archived/spring-boot-logging-app.%d{dd-MM-yyyy}.log
        </fileNamePattern>
        <maxHistory>10</maxHistory>
        <totalSizeCap>100MB</totalSizeCap>
    </rollingPolicy>
</appender>

<root level="info">
    <appender-ref ref="STDFILE"/>
</root>
```

- If we configure logging in logback.xml as well as application.properties then application.properties will take preference for logging

What is mean by Embedded Tomcat

- **For a long time, we (Java developers) used to ship our applications as war (Web ARchive) and ear (Enterprise ARchive) files.**
- **We used to deploy these builds(WARs/EARs) on application servers (like Tomcat, WildFly, WebSphere, EAP etc.) already up and running in production environment.**
- **For the last couple of years, developers around the world started changing this deployment strategy.**
- **Instead of shipping applications that had to be deployed on running servers, they started shipping applications that contain the server inside the bundle.**
- **That is, they started creating jar (Java ARchive) files that are executable and that starts and manage the server programmatically.**

- **When we create an application deployable, we would embed the server (for example, tomcat) inside the deployable.**

Need of Embedded Tomcat

- ✓ We can start new instance of application using just a single command
- ✓ All dependencies of the application are declared explicitly in the application code.
- ✓ The responsibility of Application execution isn’t spread across different teams.
- ✓ Also, as this approach fits perfectly in the microservices architecture.

Embedded tomcat configuration

Embedded tomcat provides lot of default properties and also allows us to customize it for expected smooth execution.

Embedded tomcat allows us to configure below properties:

- **Server address & port**
- **Server connection properties**
- **Error Handling properties**
- **Enable/Disable HTTPS**
- **Access logging**

Embedded tomcat configuration

Server address & port

```
server.address=my_custom_ip
server.port=80
```

Server connection properties

```
server.connection-timeout=10s
server.max-http-header-size=8KB

server.tomcat.accept-count=100
server.tomcat.max-connections=10000
server.tomcat.max-threads=200
server.tomcat.min-spare-threads=10
server.tomcat.max-swallow-size=2MB
server.tomcat.max-http-post-size=2MB
```

Error handling

```
server.error.include-exception=false
server.error.include-stacktrace=never
server.error.path=/error
server.error.whitelabel.enabled=true
```

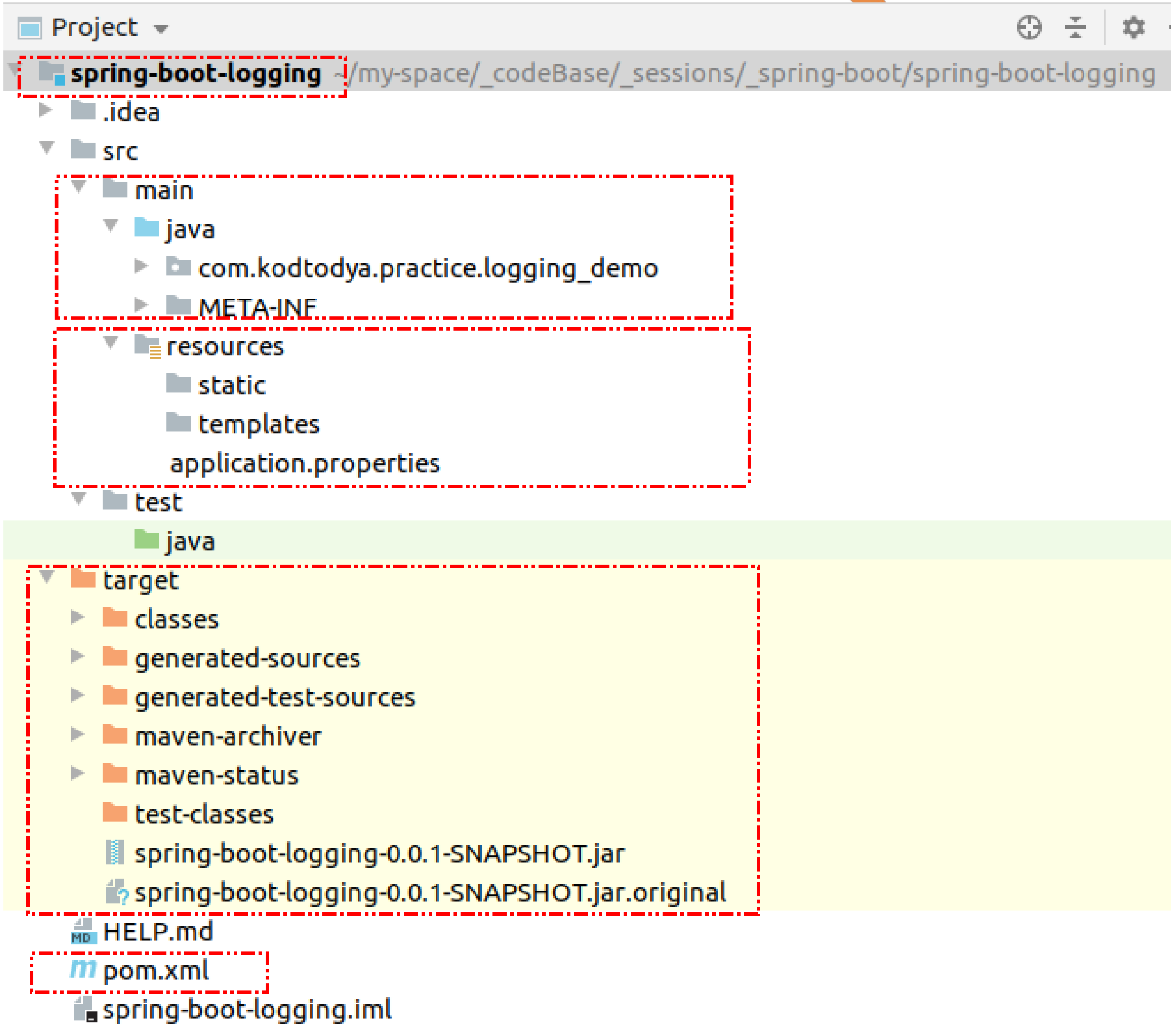
Enable/Disable HTTPS

```
server.ssl.enabled=true
server.ssl.protocol=TLS1.2

server.ssl.key-alias=tomcat
server.ssl.key-store=keystore-path
server.ssl.key-store-type=keystore-type
server.ssl.key-store-provider=provider
server.ssl.key-store-password=some-password

server.ssl.trust-store=store-path
server.ssl.trust-store-type=JKS
server.ssl.trust-store-provider=provider
server.ssl.trust-store-password=some-password
```

Access logging

```
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.directory=logs
server.tomcat.accesslog.file-date-format=yyyy-MM-dd
server.tomcat.accesslog.prefix=access_log
server.tomcat.accesslog.suffix=.log
server.tomcat.accesslog.rotate=true
```

Code Structure



```
Project
spring-boot-logging ~/my-space/_codeBase/_sessions/_spring-boot/spring-boot-logging
  .idea
  src
    main
      java
        com.kodtodya.practice.logging_demo
        META-INF
      resources
        static
        templates
        application.properties
    test
      java
  target
    classes
    generated-sources
    generated-test-sources
    maven-archiver
    maven-status
    test-classes
    spring-boot-logging-0.0.1-SNAPSHOT.jar
    spring-boot-logging-0.0.1-SNAPSHOT.jar.original
  HELP.md
  pom.xml
  spring-boot-logging.iml
```

Let’s revise

- Part – 1 : Fundamentals
  - Service Oriented Architecture (SOA)
  - Micro-services
  - Spring Boot and why Spring boot?
  - Bootstrapping
  - Logging
  - Embedded Tomcat and it’s need
  - Embedded Tomcat configuration
  - Code Structure

Spring Boot – 2.x

Part-2 | Coding time

Course Structure

- **Part – 2 : Coding time**
  - **Runners**
  - **Application Properties**
  - **Spring Boot Annotations**
  - **Actuator**
  - **Interceptor**
  - **Internationalization**
  - **Servlet Filter**
  - **CORS Support**
  - **Spring Boot Exception Handling API**
  - **Scheduling**

Runners

- Get a callback at the SpringApplication startup
- Spring Boot provides two interfaces:

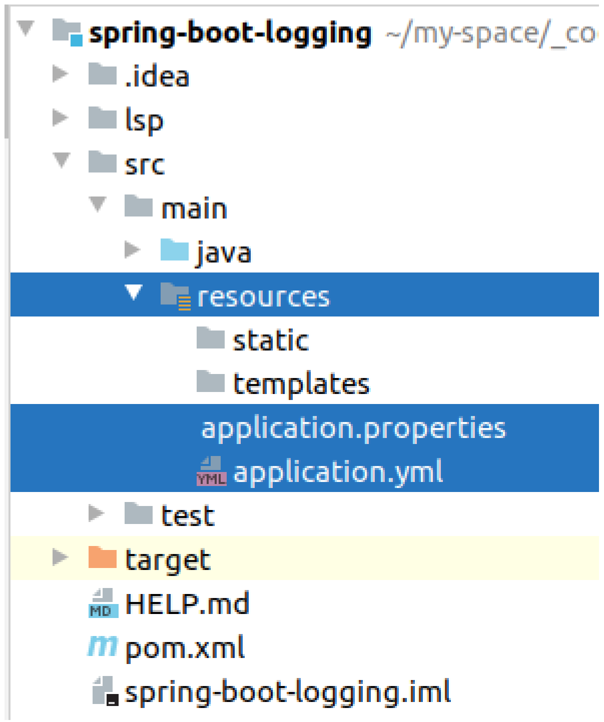
ApplicationRunner	CommandLineRunner

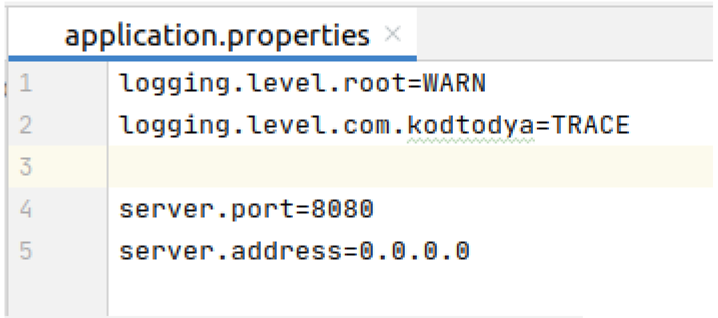
- Execute the code after the Spring Boot application is started.
- Technically there's no difference between the two, both are called at the end of SpringApplication.run().
- There's only one difference; ApplicationRunner.run() is called with ApplicationArguments and CommandLineRunner.run() is called with String[] args.
- ApplicationRunner wraps the raw application arguments & exposes the ApplicationArguments interface, which has many convenient methods to get arguments, like getOptionNames() to return all the arguments' names, getOptionValues() to return the argument value, and raw source arguments with method getSourceArgs().

Application properties

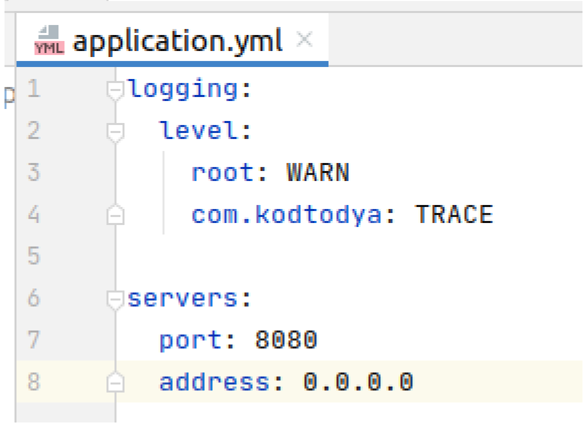
- Spring Boot support dynamic application properties with the help of **application.properties** or **application.yaml** inside **‘src/main/resources’** classpath.

- Spring Boot also supports the externalization of properties file outside of our build. In simple words, you can keep your properties file at network location still use it in application.





```
logging.level.root=WARN
logging.level.com.kodtodya=TRACE

server.port=8080
server.address=0.0.0.0
```



```
logging:
  level:
    root: WARN
    com.kodtodya: TRACE

servers:
  port: 8080
  address: 0.0.0.0
```

Spring core annotations

- @Bean
- @Required
- @Scope
- @Qualifier
- @Autowired
- @ComponentScan
- @Lazy
- @Value
- @DependsOn
- @Lookup
- @Primary

- @Component
- @Service
- @Repository
- @Configuration

- @Profile
- @Import
- @ImportResource
- @PropertySource
- @PropertySources

Spring boot annotations

@SpringBootApplication: This annotation covers @EnableAutoConfiguration, @CommponentScan and @SpringBootConfiguration annotations

@ConditionalOnClass and @ConditionalOnMissingClass: Using these conditions, Spring will only use the marked auto-configuration bean if the class in the annotation's argument is present/absent

```
@Configuration
@ConditionalOnClass(DataSource.class)
class MySQLAutoconfiguration {
    //...
}
```

@ConditionalOnBean and @ConditionalOnMissingBean: We can use these annotations when we want to define conditions based on the presence or absence of a specific bean

```
@Bean
@ConditionalOnBean(name = "dataSource")
ConnectionFactory getConnectionFactory() {
    // ...
}
```

@ConditionalOnProperty: With this annotation, we can make conditions on the values of properties

```
@Bean
@ConditionalOnProperty(
        name = "useJms",
        havingValue = "remote"
)
ConnectionEvent getConnectionEvent() {
    // ...
}
```

Spring boot annotations

@ConditionalOnResource: We can make Spring to use a definition only when a specific resource is present

```
@ConditionalOnResource(resources = "classpath:prod.properties")
Properties additionalProperties() {
    // ...
}
```

@ConditionalExpression: We can use this annotation in more complex situations. Spring will use the marked definition when the SpEL expression is evaluated to true

```
@Bean
@ConditionalOnExpression("${useJms} && ${jms.host == 'localhost'}")
ConnectionFactory getConnectionFactory() {
    // ...
}
```

@Conditional: For even more complex conditions, we can create a class evaluating the custom condition. We use custom condition with this annotation.

```
@Conditional(JmsCondition.class)
Properties additionalProperties() {
    //...
}
```

Spring MVC & REST annotations

- @RequestMapping: marks request handler methods inside @Controller classes & can be configured using:
  - path, or its aliases, name, and value: which URL the method is mapped to
  - method: HTTP methods
  - params: filters requests based on presence, absence, or value of HTTP parameters
  - headers: filters requests based on presence, absence, or value of HTTP headers
  - consumes: which media types the method can consume in the HTTP request body
  - produces: which media types the method can produce in the HTTP response body

- @GetMapping: Specialized version of @RequestMapping for GET request
- @PostMapping: Specialized version of @RequestMapping for POST request
- @PutMapping: Specialized version of @RequestMapping for PUT request
- @DeleteMapping: Specialized version of @RequestMapping for DELETE request
- @PatchMapping: Specialized version of @RequestMapping for PATCH request

Spring MVC & REST annotations

@ResponseBody: binds a method return value to the web response body. It is not interpreted as a view name. It uses HTTP Message converters to convert the return value to HTTP response body, based on the content-type in the request HTTP header.

@PathVariable: indicates that a method parameter should be bound to a URI template variable. If the method parameter is Map<String, String> then the map is populated with all path variable names and values.

@RequestParam: used to bind a web request parameter to a method parameter.

@RequestHeader: binds request header values to method parameters. If the method parameter is Map, MultiValueMap<String, String>, or HttpHeaders then the map is populated with all header names and values.

@RestController: Specialized version of @Controller for creating REST controller

@RequestAttribute: helps us access objects which have been populated or pre-existing global request attributes (outside the controller) on the server-side but during the same HTTP request.

Actuator

- Spring Boot Actuator provides secured HTTP endpoints or JMX beans for monitoring and managing our Spring Boot application. By default, all actuator endpoints are secured.

- Spring Boot Actuators help us in:
  - Monitoring health of our app
  - Gathering metrics
  - Understanding traffic
  - State of our database

- The main benefit of this library is that we can get production grade tools without having to actually implement these features ourselves.

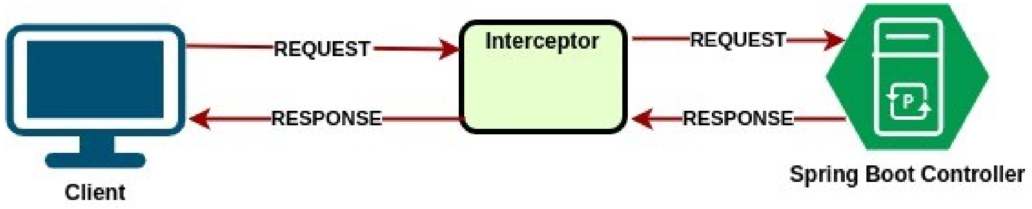
- Adding actuator dependency to project will make several endpoints available for us out of the box. As with most Spring modules, we can easily configure or extend it in many ways.

Actuator

- Spring Boot Actuator provides secured HTTP endpoints or JMX beans for monitoring and managing our Spring Boot application. By default, all actuator endpoints are secured.
- Spring Boot Actuators help us in:
  - Monitoring health of our app
  - Gathering metrics
  - Understanding traffic
  - State of our database
- The main benefit of this library is that we can get production grade tools without having to actually implement these features ourselves.
- Adding actuator dependency to project will make several endpoints available for us out of the box. As with most Spring modules, we can easily configure or extend it in many ways.

Interceptor

- **Interceptor in Spring Boot is used to perform operations under the following situations –**
- **Before sending the request to the controller**
- **Before sending the response to the client**



- **e.g. You can use an interceptor to add the request header before sending the request to the controller and add the response header before sending the response to the client.**
- **To work with interceptor, you need to create @Component class that supports it and it should implement the HandlerInterceptor interface.**
- **The following are the three methods you should know about while working on Interceptors –**
- **preHandle() - This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.**
- **PostHandle() - This is used to perform operations before sending the response to the client.**
- **AfterCompletion() - This is used to perform operations after completing the request and response.**

Internationalization

Internationalization makes our application adaptable to different languages and regions without engineering changes on the source code.

In simple words, Internationalization is a readiness of Localization.

Spring Boot has support for internationalization which helps developer to add multiple Locale/region linguistic support to the website.

Spring boot exception handling API

- In the world of complex data processing, everyone knows what exception is and how we are typically handling it in Java as well as Spring. We have few more ways to handle exceptions in Spring and support in Spring Boot for Exception Handling.

- Default spring validation support
  - Annotate model class with required validation specific annotations such as @NotEmpty, @Email etc.
  - Enable validation of request body by @Valid annotation
- Exception model classes
- Custom ExceptionHandler
- Spring boot exception handling
- REST request validation annotations

Servlet Filter

- Filter is an interface available in javax.servlet package; used for performing filtering task on request to a resource (a servlet or static content), or on the response from a resource, or both.

- In fact it is an object used to intercept the HTTP requests and responses of your application.

- By using filter, we can perform two operations:

  - Before sending the request to the controller
  - Before sending a response to the client

- Specific Implementations:
  - TransactionFilter – to start and commit transactions
  - RequestResponseLoggingFilter – to log requests and responses

- Methods:
- **void init(FilterConfig filterConfig) throws ServletException:** This method tells this filter has been placed into service. It is invoked only one time. If it throws ServletException other tasks will not be performed.
- **void doFilter(ServletRequest servletRequest, ServletResponse ServletResponse, FilterChain filterChain) throws IOException, ServletException:** It is also invoked each time whenever the client sends a request or server sends a response. In this method, we can check our request is proper or not and even we can modify request and response data.

Cross-Origin Resource Sharing (CORS) Support

- **CORS is a W3C specification implemented by most browsers that allows us to specify in a flexible way what kind of cross domain requests are authorized, instead of using some less secured and less powerful hacks like IFrame or JSONP.**

- **It prevents the JavaScript code producing or consuming the requests against different origin.**

- **e.g. your web application is running on 8080 port and by using JavaScript you are trying to consume RESTful web services from 9090 port. Under such situations, you will face the Cross-Origin Resource Sharing security issue on your web browsers.**

- **Two requirements are needed to handle this issue –**
  - **RESTful web services should support the Cross-Origin Resource Sharing.**
  - **RESTful web service application should allow accessing the API(s) from the 8080 port.**

Scheduling

Scheduling is a process of executing the tasks for the specific time period. Spring Boot provides a good support to write a scheduler on the Spring applications.

- **Java Cron Expression: Used to configure the instances of CronTrigger, a subclass of org.quartz.Trigger. The @EnableScheduling annotation is used to enable the scheduler for your application. This annotation should be added into the main Spring Boot application class file.**
- **Fixed Rate: this scheduler is used to execute the tasks at the specific time. It does not wait for the completion of previous task. The values should be in milliseconds.**
- **Fixed Delay: this scheduler is used to execute the tasks at a specific time. It should wait for the previous task completion. The values should be in milliseconds.**

Let’s revise

- **Part – 2 : Coding time**
  - **Runners**
  - **Application Properties**
  - **Spring Boot Annotations**
  - **Actuator**
  - **Interceptor**
  - **Internationalization**
  - **Servlet Filter**
  - **CORS Support**
  - **Spring Boot Exception Handling API**
  - **Scheduling**

Spring Boot – 2.x

Part-3 | Administration

Eureka Server

- Eureka Server is an application that holds the information about all client-service applications.
- Eureka Server is also known as Discovery Server.
- Eureka Server comes with the bundle of Spring Cloud.
- The Eureka server is nothing but a service discovery pattern implementation, where every microservice is registered and a client microservice looks up the Eureka server to get a dependent microservice to get the job done.
- The Eureka Server is a Netflix OSS product, and Spring Cloud offers a declarative way to register and invoke services by Java annotation.

Spring Boot – 2.x

Part-4 | Unit testing

Unit testing

- Unit Testing is a one of the testing done by the developers to make sure individual unit or component functionalities are working fine.
- Mockito and Spring Boot test starter provides very good support to Unit testing.
- For injecting Mockito Mocks into Spring Beans, we need to add the Mockito-core dependency in our build configuration file.
- Maven users can add the following dependency in your pom.xml file.

```xml
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-core</artifactId>
   <version>3.3.3</version>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```

Let’s revise

- **Part – 3 : Administration**

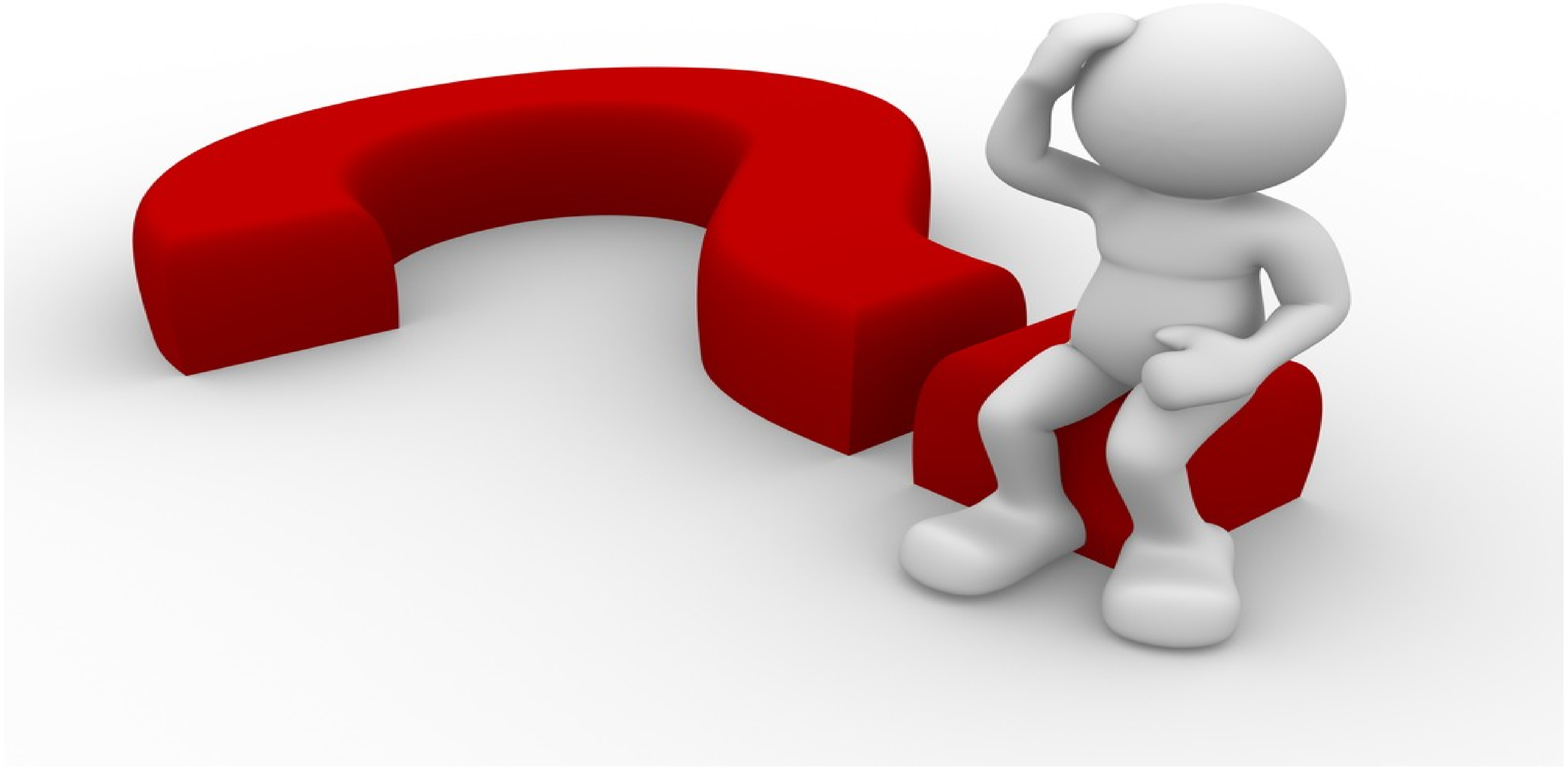
  Eureka Server

- **Part – 4 : Unit testing**
  - **Unit Testcase**

Conclusion

- Part – 1 : Fundamentals
  - Service Oriented Architecture (SOA)
  - Micro-services
  - Spring Boot and why Spring boot?
  - Bootstrapping
  - Logging
  - Embedded Tomcat and it's need
  - Embedded Tomcat configuration
  - Code Structure
- Part – 2 : Coding time
  - Runners
  - Application Properties
  - Spring Boot Annotations
- Part – 2 : Coding time
  - Spring Boot Exception Handling API
  - Actuator
  - Interceptor
  - Servlet Filter
  - CORS Support
  - Internationalization
  - Scheduling
- Part – 3 : Administration
  - Eureka Server
- Part – 4 : Unit testing
  - Unit Testcase

Questions?



kodtodya-talks

Thank you..!!!

https://kodtodya.github.io/talks