HowToDoInJava

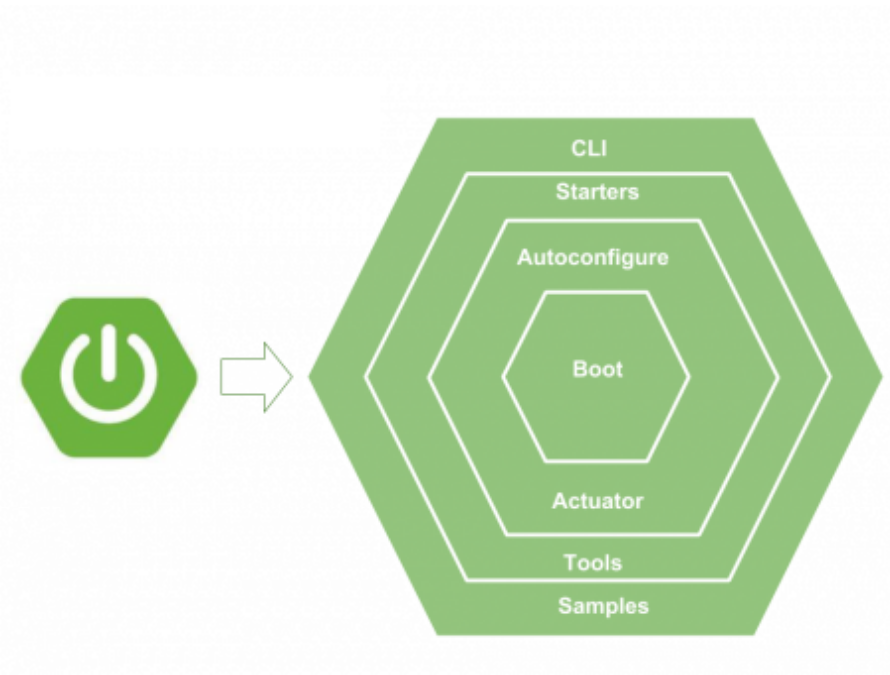
Spring Boot Interview Questions

Last Updated: January 12, 2022 | By: Lokesh Gupta | Interview Questions, Spring Boot | Interview Questions

Spring boot makes application development easier, but we may face some tough **interview questions** when it comes to testing your knowledge on how it all works. This article will help in **preparing for the next job interview**.

Table Of Contents

1. What is Spring Boot? How it is Different from Spring Framework?



Spring boot modules

Spring Boot is a Spring framework module that **provides RAD (Rapid Application Development) features** to the Spring framework with **the help of** starter templates **and *auto-configuration* features** which are very powerful and work flawlessly.

Spring Boot starters take an *opinionated view* of the Spring platform and third-party libraries. It means that as soon as we include any dependency into an application,

spring boot assumes its general purpose and automatically configures the most used classes in the library as spring beans **with sensible defaults**.

For example, if we create a WebMVC application, only including the maven dependency `spring-boot-starter-web` brings all jars/libraries used for building web, including RESTful, applications using Spring WebMVC. It also includes Tomcat as the **default embedded server**.

It also **provides a range of non-functional features** such as embedded servers, security, metrics, health checks, and externalized configuration out of the box without extra configurations.

Suppose we have to identify the difference between the Spring framework and Spring boot. In that case, we can say that Spring Boot is an extension of the Spring framework, which eliminated the boilerplate configurations required for setting up a working production-ready application.

It takes an opinionated view of the Spring and third-party libraries imported into the project and configures the behavior for us.

2. Advantages and Disadvantages of Spring Boot?

Advantages:

The two best advantages of spring boot are **simplified & version-conflict-free dependency management through the starte**r POMs and **opinionated auto-configuration** of most commonly used libraries and behaviors.

The embedded jars enable package the web applications as jar files that we can run anywhere.

The actuator module provides HTTP endpoints to access application internals like detailed performance metrics, health status, etc.

Disadvantages:

On the disadvantages side, they are very few. Still, many developers may see the **transitive dependencies included with starter poms as a burden** to deployment packaging.

Also, its *auto-configuration feature may enable many such features that we may never use in the application lifecycle. They* will sit there all the time, initialized and fully configured. It may cause some unnecessary resource utilization.

3. How Can We Set Up a Spring Boot Application With Maven?

The easiest and recommended way to set up a new Spring boot application is using the ***Spring Initializr*** **tool**. It ensures that we are using the latest artifact versions, automatically. The tool can generate the project with various configurations such as Java version, Maven or Gradle build, jar or war packaging, etc. We can also select all the features our application needs, such as Web, JPA, H2 etc.

Another way to create a project is to **Create New Maven Project wizard in IDEs** such as Eclipse or IntelliJ. After creating the maven project, we need to include the *spring-boot-starter-parent* dependency in the *pom.xml* file, if the tool didn't included it already.

4. What is AutoConfiguration? How to Enable or Disable a Certain Configuration?

Spring boot autoconfiguration scans the classpath, finds the libraries in the classpath and then attempts to guess the best default configuration for them, and finally configure all such beans.

Autoconfiguration tries to be as intelligent as possible and backs away as we define more of our own custom configuration. Autoconfiguration is always applied after user-defined custom beans have been registered.

Autoconfiguration works with the help of **@Conditional** annotations such as `@ConditionalOnBean` and `@ConditionalOnClass`.

For example, look at `AopAutoConfiguration` class. If classpath scanning finds *EnableAspectJAutoProxy*, *Aspect*, *Advice* and *AnnotatedElement* classes and `spring.aop.auto=false` is not present in the properties file, then Spring boot will configure the Spring AOP module for us.

```
@Configuration
@ConditionalOnClass({ EnableAspectJAutoProxy.class,
            Aspect.class,
            Advice.class,
            AnnotatedElement.class })
@ConditionalOnProperty(prefix = "spring.aop",
            name = "auto",
            havingValue = "true",
            matchIfMissing = true)
public class AopAutoConfiguration
{
    //code
}
```

To **enable an autoconfiguration**, just importing the correct starter dependency is enough. Everything else works as discussed above.

To **disable an autoconfiguration**, use the *exclude* attribute of the *@EnableAutoConfiguration* annotation. For instance, this code snippet disables the *DataSourceAutoConfiguration*:

```
@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)
public class MyConfiguration { }
```

5. What are Starter Dependencies?

Spring Boot starters are maven templates that contain a **collection of all the relevant transitive dependencies that are needed to start a particular functionality**.

For example, If we want to create a Spring WebMVC application, we would have included all required dependencies ourselves in a traditional setup. It leaves the *chances of version conflict* which ultimately results in *ClassCastException*.

With Spring boot, to create a WebMVC application, all we need to import is `spring-boot-starter-web` dependency. Transitively, this starter brings in all other required dependencies to build a web application, for example, *spring-webmvc*, *spring-web*, *hibernate-validator*, *tomcat-embed-core*, *tomcat-embed-el*, *tomcat-embed-websocket*, *jackson-databind*, *jackson-datatype-jdk8*, *jackson-datatype-jsr310* and *jackson-module-parameter-names*.

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

6. List down Important Annotations?

The most commonly used and important spring boot annotations are as below:

- `@ComponentScan` – enables component scanning in application classpath.
- `@EnableAutoConfiguration` – enables auto-configuration mechanism that attempts to guess and configure beans that you are likely to need.
- `@Configuration` – indicates that a class declares one or more @Bean methods to generate bean definitions and service requests for those beans at runtime.
- `@SpringBootApplication` – is a composite annotation composed on above 3 annotations. This enables auto-configuration mechanism, enable component scanning and register extra beans in the context.

- `@ImportAutoConfiguration` – imports and apply only the specified auto-configuration classes. We should use this when we don't want to enable the default autoconfiguration.
- `@AutoConfigureBefore, @AutoConfigureAfter, @AutoConfigureOrder` – shall be used if the configuration needs to be applied in a specific order (before of after).
- `@Conditional` – annotations such as *@ConditionalOnBean*, *@ConditionalOnWebApplication* or *@ConditionalOnClass* allow to register a bean only when the condition meets.

7. How to Create a REST API?

- Creating REST APIs is part of *Sping WebMVC* module that is imported via *spring-boot-starter-web* dependency.
- With the web module, we get the annotations for creating REST APIs such as *@RestController*, *@GetMapping*, *@PostMapping* etc.
- We *start with defining the REST resource model* that is generally a POJO object with necessary fields and accessor methods.
- Next, we write the *REST controllers that handle the requests coming to resource mapping URLs*. At this step, we connect to autowired DAO and other components to fetch the data from backend systems and return the response along with appropriate response codes.
- Also, we can use the annotations, such as `@ControllerAdvice`, to create a *central exception handling mechanism*.
- Optionally, based on requirements, we can plug-in additional functionalities such as *request validations* and *HATEOAS*.

8. Difference between @RestController and @Controller annotations?

The `@Controller` annotation serves as a specialization of `@Component`, allowing for implementation classes to be auto-detected through classpath scanning. The `@RequestMapping` annotated methods, in the controller class, act as request handlers for the mapped URLs.

If we need to return raw JSON or XML from the @Controller class methods, we need to annotate the method with `@ResponseBody` that indicates a method return value should be bound to the web response body.

The `@RestController` is a composite annotation that is a combination of @Controller and @ResponseBody. When we annotate a class with @RestController, all the handler methods automatically have @ResponseBody annotation applied. So all handler methods automatically return the raw JSON/XML response bodies.

```
@RestController = @Controller + @ResponseBody
```

9. What is the difference between @RequestMapping and @GetMapping?

The *@GetMapping* is a specialized composed version of *@RequestMapping* annotation that is used on handler methods for HTTP GET APIs.

```
@GetMapping = @RequestMapping(method = { RequestMethod.GET })
```

Similarly, *@PostMapping* annotation maps HTTP POST requests onto specific handler methods. It is a shorter form to write *@RequestMapping(method = RequestMethod.POST)}*.

10. How do you add a Filter to an application?

To create a filter, we simply need to implement the `javax.servlet.Filter` interface. Also, for Spring to recognize a filter, we need to define it as a bean with the *@Component* annotation.

Based on application needs, we override the filter methods *init()*, *doFilter()* and *destroy()*.

```
@Component
public class TraceLoggingFilter implements Filter {

    @Override
    public void doFilter(
      ServletRequest request,
      ServletResponse response,
      FilterChain chain) throws IOException, ServletException {

      //...
    }
}
```

To apply the filter on URL patterns, we need to register it as *FilterRegistrationBean*.

```
@Bean
public FilterRegistrationBean<TraceLoggingFilter> tracingFilter()
{
    FilterRegistrationBean<TraceLoggingFilter> filterBean
        = new FilterRegistrationBean<>();
    filterBean.setFilter(new TraceLoggingFilter());
    filterBean.addUrlPatterns("/*");
    filterBean.setOrder(1);
    return filterBean;
}
```

11. Explaing Embedded Server Support in Spring Boot?

Spring boot applications include embedded servers as part of *spring-boot-starter-web* dependency and configure ***Tomcat as the default embedded server***. It means that we can run a web application from the command prompt without setting up any complex server infrastructure.

If we want, we can exclude Tomcat and include any other embedded server. Or we can exclude the server environment altogether. It is all configuration-based.

For example, the below configuration **excludes Tomcat and includes jetty** as the embedded server.

```
<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>
```

12. Can We Disable the Default Web Server?

Yes, we can disable all embedded servers from a spring boot application. We need to look into the maven dependency tree and exclude the Tomcat library from whichever dependency is including it.

Another way to disable all embedded servers is by using the property *spring.main.web-application-type* to *none* in *application.properties* file.

```
spring.main.web-application-type=none
```

A similar configuration can be done using the Java configuration when starting the application by setting the application type to *NONE*.

```
SpringApplication application = new SpringApplication(MainApplication
application.setWebApplicationType(WebApplicationType.NONE);
application.run(args);
```

13. How to Enable Debug Logging?

The easiest way to enable debug logging is setting it through properties file:

```
debug=true

# For trace level logging
# trace=true
```

We can pass the **debug** flag during application startup as well.

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

14. How to Check all the Environment Properties in the Application?

The easiest way to list down all the properties in an application is by including the **actuator module** and accessing the URL endpoint **/env**.

Do not forget to enable the endpoint in properties configuration.

```
management.endpoints.web.exposure.include=env
```

15. How do we Define and Load Properties?

Spring boot provides many ways to set up and access properties in an application.

- Typically, spring boot reads and applies all configurations from **application.properties** file from the classpath.
- To specify a different file name or location, use the startup argument `spring.config.location`. For example, `--spring.config.location=config/*.properties` will load the properties files from the *config* folder.
- The ***@PropertySource*** annotation is a convenient mechanism for adding property sources. Note that it is a repeatable annotation so we can apply this annotation multiple times in a class. In the event of a property name collision, the last source read takes precedence.

```
@Configuration
@PropertySource("classpath:foo.properties")
@PropertySource("classpath:bar.properties")
public class AppProperties {
    //...
}
```

- We can use ***@Value*** annotation to inject a property directly into a field.

```
@Value( "${jdbc.url}" )
private String jdbcUrl;
```

- To load the test-specific properties file, the most straightforward way is to use the `@TestPropertySource` annotation.
- To load the profile-specific properties files, `application-{environment}.properties` file in the *src/main/resources* directory, and then set a Spring profile with the same *environment* name.

16. How to Connect to the Database using JPA?

To work with JPA-based repositories, first, we need to include *spring-boot-starter* and *spring-boot-starter-data-jpa* dependencies. These starters have all the autoconfiguration and hibernate-related dependencies.

Spring Boot configures ***Hibernate* as the default JPA provider,** so the application will automatically have all necessary beans such as *entityManagerFactory*.

For connecting with a database, we need to specify the datasource configuration in the *application.properties* file.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=sa
spring.datasource.url=jdbc:mysql://localhost:3306/testDb?createDataba
```

Also, note that spring boot2 uses **HikariCP as the default connection pool**. So if you need to change the connection pool, configure the respective properties.

17. Why we use Spring Boot Maven Plugin?

The plugin provides Spring Boot support in Maven, letting us **package executable jar or war archives** and **run an application in-place**. To use it, we must use Maven 3.2 (or later).

The plugin provides several goals to work with a Spring Boot application:

- `spring-boot:repackage`: create a jar or war file that is auto-executable. It can replace the regular artifact or can be attached to the build lifecycle with a separate classifier.
- `spring-boot:run`: run your Spring Boot application with several options to pass parameters to it.

- `spring-boot:start` and `stop`: integrate your Spring Boot application to the `integration-test` phase so that the application starts before it.
- `spring-boot:build-info`: generate a build information that can be used by the Actuator.

18. How to Package an Application as Executable .jar or .war File?

Executable jars (sometimes called"**fat jar**") are archives containing the compiled classes and all of the jar dependencies that the application needs to run.

To **create an executable jar**, we shall add `spring-boot-maven-plugin` in `pom.xml`. By default, this plugin package the application as `.jar` file only.

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

The first logical step to create a war file is to declare the packaging type '**war**' in *pom.xml* file.

The second thing is set the scope of embedded server dependency to '*provided*' because server dependencies will be provided by the application server where we will deploy the war file.

```
<packaging>war</packaging>

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

```
    <scope>provided</scope>
</dependency>
```

19. How to Configure Logging in Spring Boot?

Spring Boot uses Commons Logging for all logging internal to the framework and thus it is a mandatory dependency. For other logging needs, Spring boot supports default configuration for Java Util Logging, Log4J2, and Logback.

When added directly or transitively, *spring-boot-starter-logging* module configures the **default logging with Logback and SLF4J**.

The **default logging uses a console logger with a log level set to *DEBUG***, which we can change in the custom *logback.xml* file.

To use Log4j2, we must exclude *spring-boot-starter-logging* module and import *spring-boot-starter-log4j2* module. The custom configuration can be done in the *log4j2.xml* file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

20. What is Spring Actuator? What are its Advantages?

Spring boot's actuator module allows us to monitor and manage application usages in the production environment, without coding and configuration for any of them. This monitoring and management information is exposed via REST like endpoint URLs.

The simplest way to enable the features is to add a dependency to the `spring-boot-starter-actuator` starter pom file.

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

The actuator module includes several built-in endpoints and lets us add our own. Further, each individual endpoint can be *enabled* or *disabled* as well.

Some of the important and widely used actuator endpoints are given below:

Endpoint	`/env`
Usage	Returns list of properties in the current environment
Endpoint	`/health`
Usage	Returns application health information.
Endpoint	`/auditevents`
Usage	Returns all auto-configuration candidates and the reason why they 'were' or 'were not' applied.
Endpoint	`/beans`
Usage	Returns a complete list of all the Spring beans in your application.
Endpoint	`/trace`
Usage	Returns trace logs (by default the last 100 HTTP requests).
Endpoint	`/dump`

Usage	It performs a thread dump.
Endpoint	`/metrics`
Usage	It shows several useful metrics information like JVM memory used, system CPU usage, open files, and much more.

21. What are Relaxed Bindings?

Spring Boot uses some **relaxed rules for resolving configuration property names** such that we can write a simple property name in multiple ways.

For example, a simple property `log.level.my-package` can be written in the following ways and all are correct and will be resolved by framework for its value based on property source.

```
log.level.my-package = debug        //Kebab case
log.level.my_package = debug        //Underscore notation
log.level.myPackage = debug         //Camel case
LOG.LEVEL.MY-PACKAGE = debug        //Upper case format
```

Following is a list of the relaxed binding rules per property source.

Property Source	Properties Files
Types Allowed	Camel case, kebab case, or underscore notation
Property Source	YAML Files
Types Allowed	Camel case, kebab case, or underscore notation
Property Source	Environment Variables
Types Allowed	Upper case format with an underscore as the delimiter. _ should not be used within a property name

Property Source	System properties
Types Allowed	Camel case, kebab case, or underscore notation

22. How to Perform Unit Testing and Integration Testing?

Typically any software application is divided into different modules and components. When one such component is tested in isolation, it is called unit testing.

Unit tests do not verify whether the application code works with external dependencies correctly. It focuses on a single component and mocks all dependencies this component interacts with.

We can **perform unit testing help of specialized annotations** such as :

- `@JdbcTest` – can be used for a typical jdbc test when a test focuses only on jdbc-based components.
- `@JsonTest` – It is used when a test focuses only on JSON serialization.
- `@RestClientTest` – is used to test REST clients.
- `@WebMvcTest` – used for Spring MVC tests with configuration relevant to only MVC tests.

Integration tests can put the whole application in scope or only certain components – based on what is being tested. They may need to require resources like database instances and hardware to be allocated for them. Though these interactions can be mocked out as well to improve the test performance.

In integration testing, we shall focus on testing complete request processing from controller to persistence layer.

The `@SpringBootTest` annotation helps in writing integration tests. It starts the embedded server and fully initializes the application context. We can inject the dependencies in the test class using *@Autowired* annotation.

We can also provide test specific beans configuration using *nested @Configuration class* or explicit `@TestConfiguration` classes.

It also registers a `TestRestTemplate` and/or `WebTestClient` bean for use in web tests.

```
@SpringBootTest(classes = SpringBootDemoApplication.class,
        webEnvironment = WebEnvironment.RANDOM_PORT)
public class EmployeeControllerIntegrationTests
{
    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    //tests
}
```

23. What are Spring Profiles?

We can assume profiles as the various runtime environments where we will deploy the application, and we expect the application to behave differently. For example, *localhost*, *dev*, *test* and *prod*.

Spring profiles allow us to map our beans to different profiles. And **based on the profile, only mapped beans will be activated** and other beans will be deactivated.

To create a new profile, we can use the ***@Profile*** annotation. In the given example, we have configured two profiles *localhost* and *non-localhost* environments for datasource configuration.

```
@Profile("localhost")
public class LocalhostDatasourceConfig {
   //...
}
```

```
@Profile("!localhost")
public class DatasourceConfig {
   //...
}
```

To activate a profile, we can pass the *Spring.profiles.active* property during the application startup. This property can also be defined using the system property in the respective machines.

```
java -jar app.jar -Dspring.profiles.active=localhost
```

We can specify the **default profile using the property *spring.profiles.default***.

24. What Is Spring Boot DevTools Used For?

The Spring boot dev tools module provides many useful developer features for improving the development experience such as caching static resources, automatic restarts, live reload, global settings and running the remote applications.

To enable dev tools, add the `spring-boot-devtools` dependency in the build file.

Read the linked article to know the complete list of features offered by the dev tools module.

25. How to enable Hot Deployment and Live Reload on browser?

Most modern IDEs support hot-swapping of bytecode, and most code changes should reload cleanly with no side effects. Additionally, the `spring-boot-devtools` **module includes support for automatic application restarts whenever files on the classpath change**.

By default, any entry on the classpath that points to a folder is monitored for changes. Note that certain resources, such as static assets and view templates, do not need to restart the application.

The `spring-boot-devtools` module includes an **embedded LiveReload server** that can be used to trigger a browser refresh when a resource is changed. *LiveReload browser extensions* are freely available for Chrome, Firefox and Safari from *livereload.com*.

To enable/disable LiveReload server, change value of `spring.devtools.livereload.enabled` property to `true` (default value) or `false`.

26. What is Cross-Site Request Forgery attack?

CSRF stands for *Cross-Site Request Forgery* or **session riding**. It targets an end-user to, unknowingly, execute unwanted actions on a web application in which they are currently authenticated.

The unwanted actions are generally the form of URL requests that may happen either by clicking on injected links by the bad actor or by image URLs that do not need even a click.

In a Spring application, CSRF protection is **enabled by default**. We can disable it using the following spring **HttpSecurity** interface configuration.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
      .csrf().disable();
}
```

27. Explain CORS in Spring Boot?

CORS (*Cross-origin resource sharing*) allows a webpage to request additional resources into the browser from other domains e.g. fonts, CSS or static images from CDN. **CORS helps in serving web content from multiple domains** into browsers that usually have the same-origin security policy.

In Spring, `@CrossOrigin` annotation marks the annotated method or type as permitting cross-origin requests. If applied to a controller, all the handler methods permit the cross-origin requests.

```
@CrossOrigin(origins = "*", allowedHeaders = "*")
@Controller
public class HomeController
{
    //
}

//or

@Controller
public class HomeController
{
    @CrossOrigin(origins = "*", allowedHeaders = "*")
    @GetMapping(path="/")
    public String homeInit(Model model) {
        return "home";
    }
}
```

To enable CORS for the whole application, use `WebMvcConfigurer` to add `CorsRegistry`.

```
@Configuration
@EnableWebMvc
public class CorsConfiguration implements WebMvcConfigurer
{
    @Override
    public void addCorsMappings(CorsRegistry registry) {
```

```
            registry.addMapping("/**")
                    .allowedMethods("GET", "POST");
        }
    }
```

28. How to enable HTTPS/SSL support in Spring boot?

The SSL support in spring boot project can be added via `application.properties` and by adding the below entries.

```
application.properties

server.port=8443
server.ssl.key-alias=selfsigned_localhost_sslserver
server.ssl.key-password=changeit
server.ssl.key-store=classpath:ssl-server.jks
server.ssl.key-store-provider=SUN
server.ssl.key-store-type=JKS
```

Please share with us any more *spring boot interview questions*, you have encountered in past.

Happy Learning !!

Recommended Reading:

1. Spring Interview Questions with Answers
2. Spring AOP Interview Questions
3. Spring WebMVC Interview Questions
4. Java String Interview Questions with Answers
5. Core Java Interview Questions
6. Java HashMap Interview Questions
7. Java Collections Interview Questions
8. Real Java Interview Questions asked in Oracle
9. Spring Boot – Changing Context Path
0. Spring boot – CommandLineRunner interface example

Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*

Leave a Comment

Name *

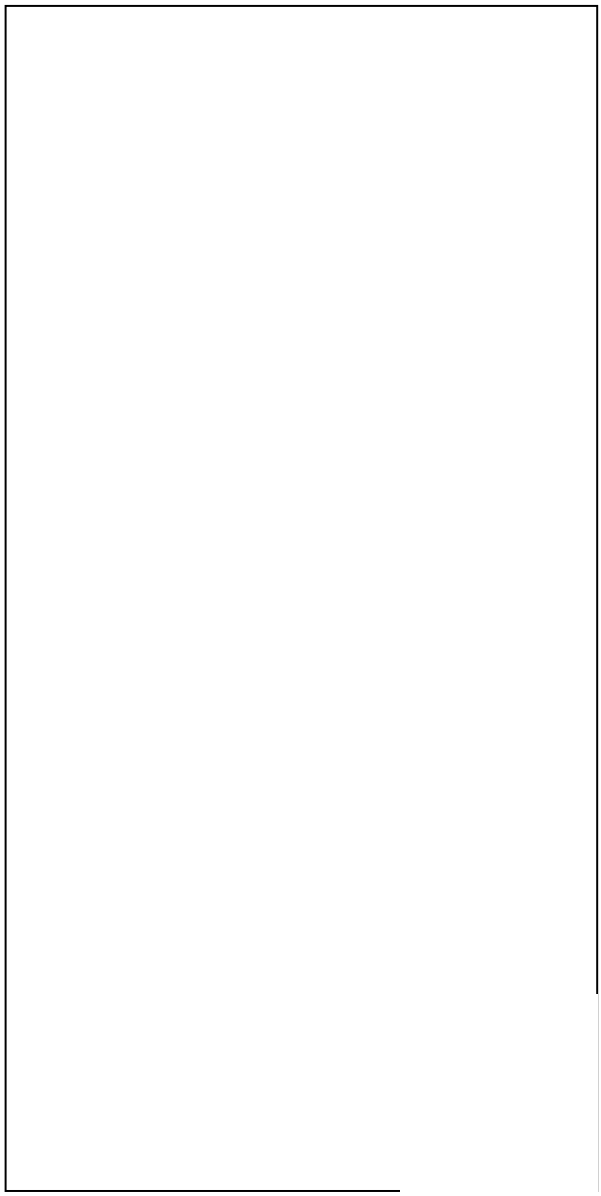
Email *

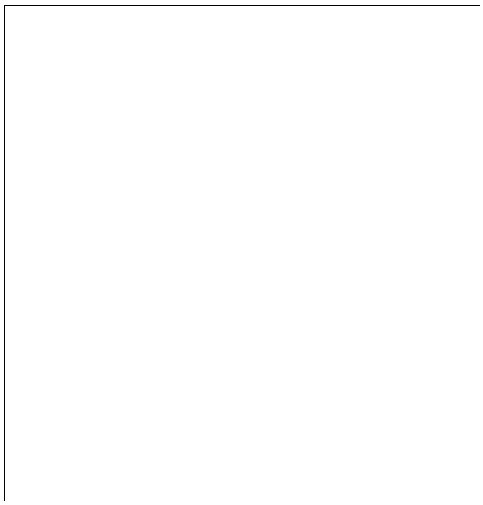
Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...





HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial