**88Spring Boot Interview Question**

**1. Intro Spring boot**

* Spring boot is an open – source, spring module which provides support for RAD (Rapid Application Development) & gives a platform for developing stand-alone & production-ready applications.

**2. Spring Vs Spring Boot**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Basis** | **Spring** | **Spring Boot** |
| 1. | Intro | Spring framework is a Java EE framework that is used to build applications. | Spring boot framework is mainly used to develop REST APIs. |
| 2. | Key feature | The key feature of Spring framework is **Dependency Injection (DI)** | The key feature of Spring Boot is **AutoConfiguration** |
| 3. | Why it’s used | Its goal is to make Java EE development easier, allowing developers to be more productive. | Spring boot provides the RAD feature to the Spring framework for rapid application development. |
| 4. | Type of Application development | Spring framework helps to create a loosely coupled application. | Spring boot helps to create a standalone application. |
| 5. | Server dependency | We need to set up the servers explicitly to test the Spring Project. | Spring boot offers built – in or embedded servers such as Tomcat, Jetty & Undertow. |
| 6. | Deployment descriptor | To run a spring application, a deployment descriptor is required. | To run Spring boot application, no deployment descriptor is required. |
| 7. | Boilerplate code | Spring framework requires too many lines of code even for minimal tasks. | Because of Auto configuration, boilerplate code can be avoided which reduces time & increase productivity. |
| 8. | In – memory database support | Spring framework doesn’t provide support for the in – memory database. | Spring boot provides support for the in – built database such as H2. |
| 9. | Configurations | In the Spring framework, we have to build configurations manually. | In Spring boot, there are default configurations that allow faster bootstrapping. |
| 10. | Dependencies | Spring framework requires a number of dependencies to create a web app. | Spring boot can get an application working with just one dependency. |
| 11. | XML configuration | In the Spring framework, XML configuration is required | No need for XML configuration in Spring boot |
| 12. | Testing | Hard due to large amount of source code. | Easier due to the reduced amount of source code. |
| 13. | CLI Tools | Doesn’t provide any CLI tool for developing & testing applications | It provides a CLI tool for developing & testing Spring boot applications. |
| 14. | Plugins | Doesn’t provide any plugin for maven, Gradle etc. | Provide build tool plugins for Maven & Gradle. The plugins offer a variety of features like the packaging of executable jars. |
| 15. | HTTP Authentication | Spring requires both the standard  **Spring-security-web** & **spring-security-config** dependencies to setup security in an application. | Spring boot only need to define the dependency of **spring-boot-starter-security** as this will automatically add all the relevant dependencies to the classpath. |

**Note**:

* In – memory databases rely on system memory as opposed to disk space for storage of data. Because memory access is faster than disk access, these databases are naturally faster.
* We can only use an in – memory database in applications or scenarios where data doesn’t need to be persisted or for the purpose of executing tests faster.

**3. Advantages of using Spring boot for application development**

1. Spring Boot helps to create stand-alone applications which can be started using **java.jar** (Doesn’t require configuring WAR files).
2. Embed Tomcat, Jetty or Undertow directly. You don’t need to deploy WAR files.
3. It provides opinionated ‘starter’ POMs to simplify your Maven configuration.
4. Auto-Configuration: Provides a way to automatically configure an application based on the dependencies present on the classpath.

**4. Ways to create spring boot projects.**

* Any IDE like eclipse or IntelliJ
* Spring boot CLI (tool that can be downloaded from official Spring site)
* Spring Initializer (web tool provided by Spring official site)

**5. What is Spring Boot dependency management**

* Spring boot manages dependencies & configuration automatically. You don’t need to specify version for any of those dependencies.
* Spring boot upgrades all dependencies automatically when we upgrade Spring boot.

**6. What are the Spring boot starters dependency descriptors**

* Starters are a set of convenient dependency descriptors which we can include in our application.
* Spring boot provides built – in starters which makes development easier & rapid.
* Instead of manually describing the dependency of our application, we simply specify the starter descriptor for the kind of application we want to build.
* For e.g., If we want to get started using Spring & JPA for database access, just include the

**spring-boot-starter-data-jpa** dependency in our project.

|  |  |  |
| --- | --- | --- |
| **No.** | **Name** | **Description** |
| 1. | spring-boot-starter | Core starter, including auto – configuration support, logging & YAML |
| 2. | spring-boot-starter-aop | Starter for aspect-oriented programming with Spring AOP & AspectJ |
| 3. | spring-boot-starter-actuator | Starter for using Spring boot’s Actuator which provides production ready features to help you monitor & manage your application. |
| 4. | spring-boot-starter-batch | Starter for using Spring Batch |
| 5. | spring-boot-starter-cache | Starter for using Spring framework’s caching support |
| 6. | spring-boot-starter-data-jdbc | Starter for using Spring Data JDBC |
| 7. | spring-boot-starter-data-jpa | Starter for using Spring Data JPA with Hibernate |
| 8. | spring-boot-starter-data-rest | Starter for exposing Spring Data repositories over REST using Spring Data REST |
| 9. | spring-boot-starter-hateoas | Starter for building hypermedia-based RESTful web application with Spring MVC & Spring HATEOAS |
| 10. | spring-boot-starter-jdbc | Starter for using JDBC with the HikariCP connection pool |
| 11. | spring-boot-starter-json | Starter for reading & writing json |
| 12. | spring-boot-starter-security | Starter for using Spring security |
| 13. | spring-boot-starter-web | Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container |
| 14. | spring-boot-starter-web-services | Starter for using Spring web services |
| 15. | spring-boot-starter-tomcat | Starter for using Tomcat as the embedded servlet container. Default servlet container started used by **spring-boot-starter-web** |
| 16. | spring-boot-starter-logging | Starter for logging using Logback. Default logging starter |

**7. Features of Spring Boot**

* **Spring Boot CLI** (Command Line Interface) – This allows to run & test Spring boot applications from command prompt.
* **Starter Dependency** – With the help of this feature, spring boot aggregates common dependencies together & eventually improves productivity.
* **Auto-configuration** – This helps in loading the default configuration according to the project you’re working on. In this way, unnecessary WAR files can be avoided.
* **Spring Actuator** – Spring boot uses actuator to provide “Management Endpoints” which helps the developer in going through the application internals, Metrics etc.
* **Logging & Security** – This ensures that all the application made using Spring boot are properly secured without any hassle.

**8. Important Spring Boot Annotations**

**1. @SpringBootApplication (@SpringBootConfiguration + @EnableAutoConfiguration + @ComponentScan)**

* This single annotation can be used to enable all 3 features

1. **@EnableAutoConfiguration**

* enables Spring boot’s auto-configuration mechanism.
* Spring boot auto – configuration attempts to automatically configure the application based on the jar dependencies that you have added i.e., it autoconfigures the beans present in the Java classpath.
* For e.g., if HSQLDB is on classpath, & you have not manually configured any database connection beans, then Spring Boot auto – configures an in-memory database.
* We can use the exclude attribute of @EnableAutoConfiguration to disable any specific auto-configuration classes.

@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

1. **@ComponentScan**

* It scans the current package & all sub-packages for injectable beans.
* Enable **@Component** scan on the package where the application is located.

1. **@Configuration**

* As per spring documentation, @SpringBootConfiguration is just an alternative to the Spring standard @Confiuguration. The only difference between the two is that the @SpringBootConfiguration allows the configuration to be found automatically.
* Allow to register extra beans in the context or import additional configuration classes.

**2. @Component**

* @Component annotation allows Spring to automatically detect our custom beans

@Component **public** **class** **OutsideScopeExample** { }

* This annotation indicates that this is a component managed by the Spring runtime.
* There are some scenarios where we won’t be able to create spring bean using @Component annotation because may be the source code comes from a third-party source, & we’re unable to add the @Component annotation there.
* In this case we can use @Bean.

**3. @Bean**

* @Bean is used by spring to gather beans at runtime, but it’s not used at the class level. Instead, we annotate methods with @Bean so than Spring can store the method’s result as a Spring bean.

@Bean **public** BeanExample **beanExample** () {**return** **new** **BeanExample** ();}

**@Component** Vs **@Bean**

|  |  |  |
| --- | --- | --- |
| **No.** | **@Component** | **@Bean** |
| 1. | @Component is a class level annotation. | @Bean is a method level annotation. |
| 2. | @Component can be used for classes which are on our classpath i.e., inside our module. | @Bean is mostly used to create bean for outside class (present in 3rd party API) |
| 3. | @Component is compatible with Spring’s auto – detection. | @Bean requires manual class instantiation. |
| 4. | If any class is annotated with @Component, it will be automatically detected by using classpath scan. | It is used to explicitly declare a single bean, rather than letting spring do it automatically. |
| 5. | We can’t create bean if class is outside spring container. | Bean can be created even class is outside the spring container. |
| 6. | It works without @Configuration annotation. | It works only when class is also annotated with @Configuration |

**4. @Controller** Vs **@RestController**

|  |  |  |
| --- | --- | --- |
| **No.** | **@Controller** | **@RestController** |
| 1. | @Controller is used to mark class as Spring MVC controller | @RestController annotation is a special controller used in REST API, & it’s combination of @Controller & @ResponseBody annotation. |
| 2. | In @Controller, we need to use @ResponseBody on every handler method. | In @RestController, we don’t need to use @ResponseBody on every handler method. |
| 3. | In @Controller, we can return a view in Spring Web MVC | In @RestController, we can’t return a view. |
| 4. | It was added to Spring 2.5 version | It was added to Spring 4.0 version |

* We typically use **@Controller** in combination with a **@RequestMapping** annotation for request handling method & we use **@ResponseBody** with response object that enables automatic serialization of the return object into the **HttpResponse**.
* When we use Spring MVC to build a web application, it’s built using Model-View-Controller pattern.
* Model View Controller is a standard design pattern to build user interface based applications.
* The Model is the part of our application that deals with databases and underlying data that is represented.
* The View is what is rendered to the user.
* The Controller contains the business logic of the application.

|  |
| --- |
| **@Controller**  **@RequestMapping** ("books")  public class SimpleBookController {  **@GetMapping**("/{id}", **produces** = "application/json")  public **@ResponseBody** Book getBook (**@PathVariable** int id) {  return findBookById (id);  }  private Book findBookById (int id) {  // ...  }  } |

* **@RestController** is a specialized version of the controller. It includes the **@Controller** & **@ResponseBody** annotations & as a result, simplifies the controller implementation.
* The controller is annotated with the **@RestController** annotation; therefore, the **@ResponseBody** isn’t required. Every request handling method of the controller class automatically serializes return objects into **HttpResponse**.

|  |
| --- |
| **@RestController**  **@RequestMapping** ("books-rest")  public class SimpleBookRestController {    **@GetMapping** ("/{id}", **produces** = "application/json")  public Book getBook **(@PathVariable** int id) {  return findBookById(id);  }  private Book findBookById (int id) {  // ...  }  } |

**5. @RequestMapping**

* **@RequestMapping** annotation is used to **map HTTP requests to handler methods of MVC & REST controllers**.
* @RequestMapping annotation can be applied to **class – level & method – level** in a controller.
* The class-level annotation maps a specific request path or pattern onto a controller.
* The method-level annotation is applied to make mappings more specific to handler methods.
* In Spring MVC applications, the DispatcherServlet (Front Controller) is responsible for routing incoming HTTP requests to handler methods of controllers. When configuring Spring MVC, we need to specify the mappings between the requests & handler methods. To configure the mapping of web requests, we use the @RequestMapping annotation.

e.g1: **@RequestMapping – by Path**

|  |
| --- |
| **@RequestMapping** (**value** = "/ex/foos", **method** = RequestMethod.GET)  **@ResponseBody**  public String getFoosBySimplePath () {  return "Get some Foos";  }  To test out this mapping run: **curl -i** [**http://localhost:8080/spring-rest/ex/foos**](http://localhost:8080/spring-rest/ex/foos) |

e.g2: **@RequestMapping – the HTTP Method**

|  |
| --- |
| **@RequestMapping** (**value** = "/ex/foos", **method** = POST)  **@ResponseBody**  public String postFoos () {  return "Post some Foos";  }  To test out this mapping run: **curl -i -X POST** [**http://localhost:8080/spring-rest/ex/foos**](http://localhost:8080/spring-rest/ex/foos) |

e.g3: **@RequestMapping with the headers Attribute**

|  |
| --- |
| **@RequestMapping (value** ="/ex/foos", **headers** ="key=val"**, method** =GET**)**  **@ResponseBody**  public String getFoosWithHeader () {  return "Get some Foos with Header";  }  To test out this mapping run: **curl -i -H** "key: val"[**http://localhost:8080/spring-rest/ex/foos**](http://localhost:8080/spring-rest/ex/foos)  **@RequestMapping** (**value** = "/ex/foos", **headers** = {"key1=val1", "key2=val2"}, **method** = GET)  **@ResponseBody**  public String getFoosWithHeaders () {  return "Get some Foos with Header";  }  To test out this mapping run:  **curl -i -H** "key1: val1" -H "key2: val2**"** [**http://localhost:8080/spring-rest/ex/foos**](http://localhost:8080/spring-rest/ex/foos) |

e.g4: **@RequestMapping Consumes & Produces**

|  |
| --- |
| **@RequestMapping** (**value** = "/ex/foos", **method** = GET, **headers** = "Accept=application/json")  **@ResponseBody**  public String getFoosAsJsonFromBrowser () {  return "Get some Foos with Header Old";  }  To test out this mapping run:  **curl -H** "Accept: application/json, text/html"[**http://localhost:8080/spring-rest/ex/foos**](http://localhost:8080/spring-rest/ex/foos)   * This way of defining the **Accept** header is not recommended as it uses **contains** instead of **equal**, so a request like above where there can be multiple accept is valid. * Above limitation is resolved. Starting from Spring 3.1, we can use **produces** & **consumes** attributes for this purpose.   **@RequestMapping** (**value** = "/ex/foos", **method** = RequestMethod.GET,  **produces** = "application/json")  **@ResponseBody**  public String getFoosAsJsonFromREST () {  return "Get some Foos with Header New";  }  To test out this mapping run:  **curl -H** "Accept: application/json"[**http://localhost:8080/spring-rest/ex/foos**](http://localhost:8080/spring-rest/ex/foos) |

e.g5: **@RequestMapping with PathVariable**

* Parts of the mapping URI can be bound to variables via the **@PathVariable** annotation.

|  |
| --- |
| **@RequestMapping** (**value** = "/ex/foos/{id}", **method** = GET)  **@ResponseBody**  public String getFoosBySimplePathWithPathVariable (**@PathVariable** String id) {  return "Get a specific Foo with id=" + id;  }  To test out this mapping run: **curl** [**http://localhost:8080/spring-rest/ex/foos/1**](http://localhost:8080/spring-rest/ex/foos/1)  **Multiple @PathVariable**  **@RequestMapping** (**value** = "/ex/foos/{fooid}/bar/{barid}", **method** = GET)  **@ResponseBody**  public String getFoosBySimplePathWithPathVariables (**@PathVariable** long fooid, **@PathVariable** long barid) {  return "Get a specific Bar with id=" + barid + " from a Foo with id=" + fooid;  }  To test out this mapping run**: curl** [**http://localhost:8080/spring-rest/ex/foos/1/bar/2**](http://localhost:8080/spring-rest/ex/foos/1/bar/2)  **@PathVariable with Regex**   * We can use Regex to restrict the mapping to only accept some specific value for path variable   **@RequestMapping** (**value** = "/ex/bars/{numericId:[\\d]+}", **method** = GET)  **@ResponseBody**  public String getBarsBySimplePathWithPathVariable (**@PathVariable** long numericId) {  return "Get a specific Bar with id=" + numericId;  }  To test out this mapping run**:**  **curl** [**http://localhost:8080/spring-rest/ex/bars/1**](http://localhost:8080/spring-rest/ex/bars/1) **// this will work**  **curl** [**http://localhost:8080/spring-rest/ex/bars/abc**](http://localhost:8080/spring-rest/ex/bars/abc) **// this won’t work** |

e.g6: **@RequestMapping with Request Parameters**

* We can use @RequestParam annotation for mapping of URL parameters.
* @RequestParam annotation tells our Spring MVC framework that the value for the handler method argument will be available as a part of a request parameter made with the request inside @RequestMapping.

|  |
| --- |
| **@RequestMapping** (**value** = "/ex/bars", **method** = GET)  **@ResponseBody**  public String getBarBySimplePathWithRequestParam (**@RequestParam("id")** long id) {  return "Get a specific Bar with id=" + id;  }  To test out this mapping run: **curl** [**http://localhost:8080/spring-rest/ex/bars?id=100**](http://localhost:8080/spring-rest/ex/bars?id=100)  **Or curl** -i -d id=100[**http://localhost:8080/spring-rest/ex/bars**](http://localhost:8080/spring-rest/ex/bars) |

|  |
| --- |
| One more way:  **@RequestMapping** (value = "/ex/bars", params = { "id", "second" }, method = GET)  **@ResponseBody**  public String getBarBySimplePathWithExplicitRequestParams (**@RequestParam("id")** long id) {  return "Narrow Get a specific Bar with id=" + id;  }  To test out this mapping run:  **curl** <http://localhost:8080/spring-rest/ex/bars?id=100&second=something> |

e.g7: **New Request Mapping Shortcuts**

* Spring framework 4.3 introduced a few new HTTP mapping annotations, all based on @RequestMapping:
* **@GetMapping** is same as **@RequestMapping** (**value** = "/get/{id}", **method** = RequestMethod.GET)
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping
* These new annotations can improve the readability & reduce the verbosity of the Code.

|  |
| --- |
| **@GetMapping** ("/get")  public **@ResponseBody** ResponseEntity<String> get () {  return new ResponseEntity<String> ("GET Response", HttpStatus.OK);  }  **@PostMapping** ("/post")  public **@ResponseBody** ResponseEntity<String> post () {  return new ResponseEntity<String> ("POST Response", HttpStatus.OK);  }  **@PutMapping** ("/put")  public **@ResponseBody** ResponseEntity<String> put () {  return new ResponseEntity<String> ("PUT Response", HttpStatus.OK);  }  **@DeleteMapping** ("/delete")  public **@ResponseBody** ResponseEntity<String> delete () {  return new ResponseEntity<String> ("DELETE Response", HttpStatus.OK);  }  **@PatchMapping** ("/patch")  public **@ResponseBody** ResponseEntity<String> patch () {  return new ResponseEntity<String> ("PATCH Response", HttpStatus.OK);  } |

**6. @Service**

* It is also a specialization of @Component annotation.
* @Service annotation can be applied only to classes.
* In an application, the business logic resides within the service layer so we use the @Service annotation to indicate that a class belongs to that layer.

**7. @Repository**

1. It is also a specialization of @Component annotation.

* @Repository annotation can be applied only to classes.

1. @Repository annotation is used to indicate that the class provides the mechanism for storage, retrieval, update, delete & search operation on objects.

**8. @Autowired**

1. The @Autowired annotation marks a constructor, Setter method, Properties & Config () method as to be autowired i.e., ‘injecting beans’ (Objects) at runtime by Spring dependency injection mechanism.
2. The @SpringBootApplication which is a combination of @Configuration, @EnableAutoConfiguration & @ComponentScan scans all the components or services & other configuration files included in the base & child packages. This will register them in Spring context & inject the beans at runtime using @Autowired.

**9. @ConfigurationProperties**

* Class level annotation
* The @ConfigurationProperties annotation specifies to Spring that this class contains some external configuration properties for our app.
* The message input that we have specified to @ConfigurationProperties(“message”) tells Spring that it should take all properties that start with messages. Like message.greeting, message.name etc. & try to inject them into this required bean.

10. @EnableAsync

* Class level annotation
* It turns on Spring’s ability to run asynchronous methods in a background thread pool.
* Rather than instantiating threads directly, this @EnableAsync uses an executor & we can specify the executor that it should use using the taskExecutor() method.
* One more important point is in our main method when we call run() method, we need to add .close() as well so that as soon as the application will complete its execution, it will close

|  |
| --- |
| **import** java.util.concurrent.Executor;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.scheduling.annotation.EnableAsync;  **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  @SpringBootApplication  @EnableAsync  **public** **class** AsynchronousMethodsDemoApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(AsynchronousMethodsDemoApplication.**class**, args).close();  }  **public** Executor taskExecutor() {  ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();  executor.setCorePoolSize(2);  executor.setMaxPoolSize(2);  executor.setQueueCapacity(500);  executor.setThreadNamePrefix("GithubLookup-");  executor.initialize();    **return** executor;  }  } |

@Async

* Method level annotation
* This @Async annotation tells our spring boot application that the code for this method needs to be run on a separate thread. It shouldn’t be on the main thread, it should be on a background thread.

11. @EnableScheduling

* Class level annotation
* It tells spring that this particular application will run tasks at scheduled intervals.
* This annotation will bring up a task executor to run the scheduled task on a background thread, not on the main thread.

**@Scheduled**

* Method level annotation
* Any method with @Scheduled annotation, Spring will invoke that method at pre-specified times based on a schedule.
* We can specify arguments to @Scheduled annotation to customize the rate etc.

e.g., @Scheduled(**fixedRate**=3000) means the method will be invoked every 3 seconds or 3000 milliseconds.

* **fixedRate** used when each invocation of the method is an independent process i.e., subsequent invocation doesn’t depend on previous invocations.
* If there’s exactly one background thread running the schedule tasks i.e., unless the previous invocation of the method is complete, the next invocation will not be started, no matter what value you used for **fixedRate**.
* Instead of fixedRate, we can use fixedDelay which allows us to configure the delay between the completion of previous execution of the method & start of the next execution.

14. @Value

15. @ConfigurationProperties

16. @Profile

18 @CrossOrigin (CORS support)

**9. REST API**

