1. Spring Boot
2. Spring Core
3. Spring MVC
4. Spring Jdbc
5. Spring Test
6. Junit jupiter
7. Mockito
8. Mock MVC
9. Logback
10. Jackson
11. SQL
12. Apache Maven
13. Swagger
14. Hibernate validator
15. Bootstrap
16. Thymeleaf

Spring Boot

Q1. What is Spring Boot and What Are Its Main Features?

Spring Boot is essentially a framework for rapid application development built on top of the Spring Framework. With its auto-configuration and embedded application server support, combined with the extensive documentation and community support it enjoys, Spring Boot is one of the most popular technologies in the Java ecosystem as of date.

Here are a few salient features:

- Starters – a set of dependency descriptors to include relevant dependencies at a go
- Auto-configuration – a way to automatically configure an application based on the dependencies present on the classpath
- Actuator – to get production-ready features such as monitoring
- Security
- Logging

Q2. What Are the Differences Between Spring and Spring Boot?

The Spring Framework provides multiple features that make the development of web applications easier. These features include dependency injection, data binding, aspect-oriented programming, data access, and many more.

Over the years, Spring has been growing more and more complex, and the amount of configuration such application requires can be intimidating. This is where Spring Boot comes in handy – it makes configuring a Spring application a breeze.

Essentially, while Spring is unopinionated, **Spring Boot takes an opinionated view of the platform and libraries, letting us get started quickly.**

Here are two of the most important benefits Spring Boot brings in:

- Auto-configure applications based on the artifacts it finds on the classpath
- Provide non-functional features common to applications in production, such as security or health checks

Please check one of our other tutorials for a detailed comparison between vanilla Spring and Spring Boot.

Q3. How Can We Set up a Spring Boot Application With Maven?

We can include Spring Boot in a Maven project just like we would any other library. However, the best way is to inherit from the *spring-boot-starter-parent* project and declare dependencies to Spring Boot starters. Doing this lets our project reuse the default settings of Spring Boot.

Inheriting the *spring-boot-starter-parent* project is straightforward – we only need to specify a *parent* element in *pom.xml*:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0.RELEASE</version>
</parent>
```

We can find the latest version of *spring-boot-starter-parent* on Maven Central.

Using the starter parent project is convenient, but not always feasible. For instance, if our company requires all projects to inherit from a standard

POM, we can still benefit from Spring Boot's dependency management using a custom parent.

Q4. What is Spring Initializr?

Spring Initializr is a convenient way to create a Spring Boot project.

We can go to the Spring Initializr site, choose a dependency management tool (either Maven or Gradle), a language (Java, Kotlin or Groovy), a packaging scheme (Jar or War), version and dependencies, and download the project.

This **creates a skeleton project for us** and saves setup time so that we can concentrate on adding business logic.

Even when we use our IDE's (such as STS or Eclipse with STS plugin) new project wizard to create a Spring Boot project, it uses Spring Initializr under the hood.

Q5. What Spring Boot Starters Are Available out There?

Each starter plays a role as a one-stop-shop for all the Spring technologies we need. Other required dependencies are then transitively pulled in and managed in a consistent way.

All starters are under the *org.springframework.boot* group and their names start with *spring-boot-starter-*. **This naming pattern makes it easy to find starters, especially when working with IDEs that support searching dependencies by name.**

At the time of this writing, there are more than 50 starters at our disposal. The most commonly used are:

- *spring-boot-starter:* core starter, including auto-configuration support, logging, and YAML
- *spring-boot-starter-aop:* starter for aspect-oriented programming with Spring AOP and AspectJ
- *spring-boot-starter-data-jpa:* starter for using Spring Data JPA with Hibernate
- *spring-boot-starter-security:* starter for using Spring Security
- *spring-boot-starter-test:* starter for testing Spring Boot applications
- *spring-boot-starter-web:* starter for building web, including RESTful, applications using Spring MVC

For a complete list of starters, please see this repository.

To find more information about Spring Boot starters, take a look at Intro to Spring Boot Starters.

Q6. How to Disable a Specific Auto-Configuration?

If we want to disable a specific auto-configuration, we can indicate it using the *exclude* attribute of the *@EnableAutoConfiguration* annotation. For instance, this code snippet neutralizes *DataSourceAutoConfiguration*:

```
// other annotations

@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)

public class MyConfiguration { }
```

If we enabled auto-configuration with the *@SpringBootApplication* annotation — which has *@EnableAutoConfiguration* as a meta-annotation — we could disable auto-configuration with an attribute of the same name:

```
// other annotations

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

public class MyConfiguration { }
```

We can also disable an auto-configuration with the *spring.autoconfigure.exclude* environment property. This setting in the *application.properties* file does the same thing as before:

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

Q7. How to Register a Custom Auto-Configuration?

To register an auto-configuration class, we must have its fully-qualified name listed under the *EnableAutoConfiguration* key in the *META-INF/spring.factories* file:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.baeldung.autoconfigure.CustomAutoConfiguration
```

If we build a project with Maven, that file should be placed in the *resources/META-INF* directory, which will end up in the mentioned location during the *package* phase.

Q8. How to Tell an Auto-Configuration to Back Away When a Bean Exists?

To instruct an auto-configuration class to back off when a bean is already existent, we can use the *@ConditionalOnMissingBean* annotation. The most noticeable attributes of this annotation are:

- *value:* The types of beans to be checked
- *name:* The names of beans to be checked

When placed on a method adorned with *@Bean*, the target type defaults to the method's return type:

```
@Configuration

public class CustomConfiguration {

    @Bean

    @ConditionalOnMissingBean

    public CustomService service() { ... }

}
```

Q9. How to Deploy Spring Boot Web Applications as Jar and War Files?

Traditionally, we package a web application as a WAR file, then deploy it into an external server. Doing this allows us to arrange multiple applications on the same server. During the time that CPU and memory were scarce, this was a great way to save resources.

However, things have changed. Computer hardware is fairly cheap now, and the attention has turned to server configuration. A small mistake in configuring the server during deployment may lead to catastrophic consequences.

Spring tackles this problem by providing a plugin, namely *spring-boot-maven-plugin*, to package a web application as an executable JAR. To include this plugin, just add a *plugin* element to *pom.xml*:

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

With this plugin in place, we'll get a fat JAR after executing the *package* phase. This JAR contains all the necessary dependencies, including an embedded server. Thus, we no longer need to worry about configuring an external server.

We can then run the application just like we would an ordinary executable JAR.

Notice that the *packaging* element in the *pom.xml* file must be set to *jar* to build a JAR file:

```
<packaging>jar</packaging>
```

If we don't include this element, it also defaults to *jar*.

In case we want to build a WAR file, change the *packaging* element to *war*:

```
<packaging>war</packaging>
```

And leave the container dependency off the packaged file:

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

After executing the Maven *package* phase, we'll have a deployable WAR file.

Q10. How to Use Spring Boot for Command Line Applications?

Just like any other Java program, a Spring Boot command line application must have a *main* method. This method serves as an entry point, which invokes the *SpringApplication#run* method to bootstrap the application:

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class);
        // other statements
    }
}
```

The *SpringApplication* class then fires up a Spring container and auto-configures beans.

Notice we must pass a configuration class to the *run* method to work as the primary configuration source. By convention, this argument is the entry class itself.

After calling the *run* method, we can execute other statements as in a regular program.

Q11. What Are Possible Sources of External Configuration?

Spring Boot provides support for external configuration, allowing us to run the same application in various environments. **We can use properties files, YAML files, environment variables, system properties, and command-line option arguments to specify configuration properties.**

We can then gain access to those properties using the *@Value* annotation, a bound object via the *@ConfigurationProperties* annotation, or the *Environment* abstraction.

Q12. What Does it Mean that Spring Boot Supports Relaxed Binding?

Relaxed binding in Spring Boot is applicable to the type-safe binding of configuration properties.

With relaxed binding, **the key of a property doesn't need to be an exact match of a property name.** Such an environment property can be written in camelCase, kebab-case, snake_case, or in uppercase with words separated by underscores.

For example, if a property in a bean class with the *@ConfigurationProperties* annotation is named *myProp*, it can be bound to any of these environment properties: *myProp*, *my-prop*, *my_prop*, or *MY_PROP*.

Q13. What is Spring Boot Devtools Used For?

Spring Boot Developer Tools, or DevTools, is a set of tools making the development process easier. To include these development-time features, we just need to add a dependency to the *pom.xml* file:

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

The *spring-boot-devtools* module is automatically disabled if the application runs in production. The repackaging of archives also excludes this module by default. Hence, it won't bring any overhead to our final product.

By default, DevTools applies properties suitable to a development environment. These properties disable template caching, enable debug logging for the web group, and so on. As a result, we have this sensible development-time configuration without setting any properties.

Applications using DevTools restart whenever a file on the classpath changes. This is a very helpful feature in development, as it gives quick feedback for modifications.

By default, static resources, including view templates, don't set off a restart. Instead, a resource change triggers a browser refresh. Notice this can only happen if the LiveReload extension is installed in the browser to interact with the embedded LiveReload server that DevTools contains.

For further information on this topic, please see Overview of Spring Boot DevTools.

Q14. How to Write Integration Tests?

When running integration tests for a Spring application, we must have an *ApplicationContext*.

To make our life easier, Spring Boot provides a special annotation for testing – *@SpringBootTest*. This annotation creates an *ApplicationContext* from configuration classes indicated by its *classes* attribute.

In case the *classes* attribute isn't set, Spring Boot searches for the primary configuration class. The search starts from the package containing the test up until it finds a class annotated with *@SpringBootApplication* or *@SpringBootConfiguration*.

For detailed instructions, check out our tutorial on testing in Spring Boot.

Q15. What Is Spring Boot Actuator Used For?

Essentially, Actuator brings Spring Boot applications to life by enabling production-ready features. **These features allow us to monitor and manage applications when they're running in production**.

Integrating Spring Boot Actuator into a project is very simple. All we need to do is to include the *spring-boot-starter-actuator* starter in the *pom.xml* file:

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

Spring Boot Actuator can expose operational information using either HTTP or JMX endpoints. Most applications go for HTTP, though, where the identity of an endpoint and the */actuator* prefix form a URL path.

Here are some of the most common built-in endpoints Actuator provides:

- *env:* Exposes environment properties
- *health:* Shows application health information
- *httptrace:* Displays HTTP trace information
- *info:* Displays arbitrary application information
- *metrics:* Shows metrics information
- *loggers:* Shows and modifies the configuration of loggers in the application
- *mappings:* Displays a list of all *@RequestMapping* paths

Please refer to our Spring Boot Actuator tutorial for a detailed rundown.

Q16. Which Is a Better Way to Configure a Spring Boot Project – Using Properties or YAML?

YAML offers many advantages over properties files, such as:

- More clarity and better readability
- Perfect for hierarchical configuration data, which is also represented in a better, more readable format
- Support for maps, lists, and scalar types
- Can include several profiles in the same file (since Spring Boot 2.4.0, this is possible for properties files too)

However, writing it can be a little difficult and error-prone due to its indentation rules.

For details and working samples, please refer to our Spring YAML vs Properties tutorial.

Q17. What Are the Basic Annotations that Spring Boot Offers?

The primary annotations that Spring Boot offers reside in its *org.springframework.boot.autoconfigure* and its sub-packages. Here are a couple of basic ones:

- *@EnableAutoConfiguration* – to make Spring Boot look for auto-configuration beans on its classpath and automatically apply them.
- *@SpringBootApplication* – used to denote the main class of a Boot Application. This annotation combines *@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan* annotations with their default attributes.

Spring Boot Annotations offers more insight into the subject.

Q18. How Can You Change the Default Port in Spring Boot?

We can change the default port of a server embedded in Spring Boot using one of these ways:

- using a properties file – we can define this in an *application.properties* (or *application.yml*) file using the property *server.port*
- programmatically – in our main *@SpringBootApplication* class, we can set the *server.port* on the *SpringApplication* instance

using the command line – when running the application as a jar file, we can set the server.port as a java command argument:
java -jar -Dserver.port=8081 myspringproject.jar

-

Q19. Which Embedded Servers does Spring Boot Support, and How to Change the Default?

As of date, **Spring MVC supports Tomcat, Jetty, and Undertow**. Tomcat is the default application server supported by Spring Boot's *web* starter.

Spring WebFlux supports Reactor Netty, Tomcat, Jetty, and Undertow with Reactor Netty as default.

In Spring MVC, to change the default, let's say to Jetty, we need to exclude Tomcat and include Jetty in the dependencies:

```
<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>
```

Similarly, to change the default in WebFlux to UnderTow, we need to exclude Reactor Netty and include UnderTow in the dependencies.

"Comparing embedded servlet contains in Spring Boot" contains more details on the different embedded servers we can use with Spring MVC.

Q20. Why Do We Need Spring Profiles?

When developing applications for the enterprise, we typically deal with multiple environments such as Dev, QA, and Prod. The configuration properties for these environments are different.

For example, we might be using an embedded H2 database for Dev, but Prod could have the proprietary Oracle or DB2. Even if the DBMS is the same across environments, the URLs would definitely be different.

To make this easy and clean, **Spring has the provision of profiles, to help separate the configuration for each environment**. So that instead of maintaining this programmatically, the properties can be kept in separate files such as *application-dev.properties* and *application-prod.properties*. The default *application.properties* points to the currently active profile using *spring.profiles.active* so that the correct configuration is picked up.

Spring Profiles gives a comprehensive view of this topic.

Spring Core

Q1. What Is Spring Framework?

Spring is the most broadly used framework for the development of Java Enterprise Edition applications. The core features of Spring can be used in developing any Java application.

We can use its extensions for building various web applications on top of the Jakarta EE platform, or we may just use its dependency injection provisions in simple standalone applications.

Q2. What Are the Benefits of Using Spring?

Spring targets to make Jakarta EE development easier. Here are the advantages of using it:

- Lightweight: there is a slight overhead of using the framework in development
- Inversion of Control (IoC): Spring container takes care of wiring dependencies of various objects, instead of creating or looking for dependent objects
- Aspect Oriented Programming (AOP): Spring supports AOP to separate business logic from system services

- IoC container: it manages Spring Bean life cycle and project specific configurations
- MVC framework: that is used to create web applications or RESTful web services, capable of returning XML/JSON responses
- Transaction management: reduces the amount of boiler-plate code in JDBC operations, file uploading, etc., either by using Java annotations or by Spring Bean XML configuration file
- Exception Handling: Spring provides a convenient API for translating technology-specific exceptions into unchecked exceptions

Q3. What Spring Sub-Projects Do You Know? Describe Them Briefly.

- Core – a key module that provides fundamental parts of the framework, like IoC or DI
- JDBC – this module enables a JDBC-abstraction layer that removes the need to do JDBC coding for specific vendor databases
- ORM integration – provides integration layers for popular object-relational mapping APIs, such as JPA, JDO, and Hibernate
- Web – a web-oriented integration module, providing multipart file upload, Servlet listeners, and web-oriented application context functionalities
- MVC framework – a web module implementing the Model View Controller design pattern
- AOP module – aspect-oriented programming implementation allowing the definition of clean method-interceptors and pointcuts

Q4. What Is Dependency Injection?

Dependency Injection, an aspect of Inversion of Control (IoC), is a general concept stating that you do not create your objects manually but instead describe how they should be created. An IoC container will instantiate required classes if needed.

For more details, please refer here.

Q5. How Can We Inject Beans in Spring?

A few different options exist:

- Setter Injection
- Constructor Injection
- Field Injection

The configuration can be done using XML files or annotations.

For more details, check this article.

Q6. Which Is the Best Way of Injecting Beans and Why?

The recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. Constructor injection allows injecting values to immutable fields and makes testing easier.

Q7. What Is the Difference Between Beanfactory and Applicationcontext?

BeanFactory is an interface representing a container that provides and manages bean instances. The default implementation instantiates beans lazily when getBean() is called.

ApplicationContext is an interface representing a container holding all information, metadata, and beans in the application. It also extends the BeanFactory interface but the default implementation instantiates beans eagerly when the application starts. This behavior can be overridden for individual beans.

For all differences, please refer to the reference.

Q8. What Is a Spring Bean?

The Spring Beans are Java Objects that are initialized by the Spring IoC container.

Q9. What Is the Default Bean Scope in Spring Framework?

By default, a Spring Bean is initialized as a singleton.

Q10. How to Define the Scope of a Bean?

To set Spring Bean's scope, we can use @Scope annotation or “scope” attribute in XML configuration files. There are five supported scopes:

- singleton
- prototype
- request
- session
- global-session

For differences, please refer here.

Q11. Are Singleton Beans Thread-Safe?

No, singleton beans are not thread-safe, as thread safety is about execution, whereas the singleton is a design pattern focusing on creation. Thread safety depends only on the bean implementation itself.

Q12. What Does the Spring Bean Lifecycle Look Like?

First, a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some initialization to get it into a usable state. After that, when the bean is no longer required, it will be removed from the IoC container.

The whole cycle with all initialization methods is shown on the image (source):

Spring MVC

Q1. Why Should We Use Spring MVC?

Spring MVC implements a clear separation of concerns that allows us to develop and unit test our applications easily.

The concepts like:

- Dispatcher Servlet
- Controllers
- View Resolvers
- Views, Models
- *ModelAndView*
- Model and Session Attributes

are completely independent of each other, and they are responsible for one thing only.

Therefore, **MVC gives us quite big flexibility**. It's based on interfaces (with provided implementation classes), and we can configure every part of the framework by using custom interfaces.

Another important thing is that we aren't tied to a specific view technology (for example, JSP), but we have the option to choose from the ones we like the most.

Also, **we don't use Spring MVC only in web applications development but in the creation of RESTful web services as well**.

Q2. What Is the Role of the *@Autowired* Annotation?

The *@Autowired* annotation can be used with fields or methods for injecting a bean by type. This annotation allows Spring to resolve and inject collaborating beans into your bean.

For more details, please refer to the tutorial about *@Autowired* in Spring.

Q3. Explain a Model Attribute

The *@ModelAttribute* annotation is one of the most important annotations in Spring MVC. **It binds a method parameter or a method return value to a named model attribute and then exposes it to a web view**.

If we use it at the method level, it indicates the purpose of that method is to add one or more model attributes.

On the other hand, when used as a method argument, it indicates the argument should be retrieved from the model. When not present, we should first instantiate it and then add it to the model. Once present in the model, we should populate the arguments fields from all request parameters that have matching names.

More about this annotation can be found in our article related to the *@ModelAttribute* annotation.

Q4. Explain the Difference Between *@Controller* and *@RestController*?

The main difference between the *@Controller* and *@RestController* annotations is that **the *@ResponseBody* annotation is automatically included in the *@RestController***. This means that we don't need to annotate our handler methods with the *@ResponseBody*. We need to do this in a *@Controller* class if we want to write response type directly to the HTTP response body.

Q5. Describe a *PathVariable*

We can use the *@PathVariable* annotation as a handler method parameter in order to extract the value of a URI template variable.

For example, if we want to fetch a user by id from the *www.mysite.com/user/123*, we should map our method in the controller as */user/{id}*:

```
@RequestMapping("/user/{id}")
public String handleRequest(@PathVariable("id") String userId, Model map) {}
```

The *@PathVariable* has only one element named *value*. It's optional and we use it to define the URI template variable name. If we omit the value element, then the URI template variable name must match the method parameter name.

It's also allowed to have multiple *@PathVariable* annotations, either by declaring them one after another:

```
@RequestMapping("/user/{userId}/name/{userName}")
public String handleRequest(@PathVariable String userId,
  @PathVariable String userName, Model map) {}
```

or putting them all in a *Map<String, String>* or *MultiValueMap<String, String>*:

```
@RequestMapping("/user/{userId}/name/{userName}")
public String handleRequest(@PathVariable Map<String, String> varsMap, Model map) {}
```

Q6. Validation Using Spring MVC

Spring MVC supports JSR-303 specifications by default. We need to add JSR-303 and its implementation dependencies to our Spring MVC application. Hibernate Validator, for example, is one of the JSR-303 implementations at our disposal.

JSR-303 is a specification of the Java API for bean validation, part of Jakarta EE and JavaSE, which ensures that properties of a bean meet specific criteria, using annotations such as *@NotNull*, *@Min*, and *@Max*. More about validation is available in the Java Bean Validation Basics article.

Spring offers the *@Validator* annotation and the *BindingResult* class. The *Validator* implementation will raise errors in the controller request handler method when we have invalid data. Then we may use the *BindingResult* class to get those errors.

Besides using the existing implementations, we can make our own. To do so, we create an annotation that conforms to the JSR-303 specifications first. Then, we implement the *Validator* class. Another way would be to implement Spring's *Validator* interface and set it as the validator via *@InitBinder* annotation in *Controller* class.

To check out how to implement and use your own validations, please see the tutorial regarding Custom Validation in Spring MVC.

Q7. What are the *@RequestBody* and the *@ResponseBody*?

The *@RequestBody* annotation, used as a handler method parameter, binds the HTTP Request body to a transfer or a domain object. Spring automatically deserializes incoming HTTP Request to the Java object using Http Message Converters.

When we use the *@ResponseBody* annotation on a handler method in the Spring MVC controller, it indicates that we'll write the return type of the method directly to the HTTP response body. We'll not put it in a *Model*, and Spring won't interpret as a view name.

Please check out the article on *@RequestBody* and *@ResponseBody* to see more details about these annotations.

Q8. Explain *Model*, *ModelMap* and *ModelAndView*?

The *Model* interface defines a holder for model attributes. **The *ModelMap* has a similar purpose, with the ability to pass a collection of values**. It then treats those values as if they were within a *Map*. We should note that in *Model* (*ModelMap*) we can only store data. We put data in and return a view name.

On the other hand, **with the *ModelAndView*, we return the object itself**. We set all the required information, like the data and the view name, in the object we're returning.

You can find more details in the article on *Model, ModelMap, and ModelView*.

Q9. Explain *SessionAttributes* and *SessionAttribute*

The *@SessionAttributes* annotation is used for storing the model attribute in the user's session. We use it at the controller class level, as shown in our article about the Session Attributes in Spring MVC:

```
@Controller
@RequestMapping("/sessionattributes")
@SessionAttributes("todos")
public class TodoControllerWithSessionAttributes {

   @GetMapping("/form")
   public String showForm(Model model,
     @ModelAttribute("todos") TodoList todos) {
```

```
        // method body
        return "sessionattributesform";
    }

    // other methods
}
```

In the previous example, the model attribute '*todos*' will be added to the session if the *@ModelAttribute* and the *@SessionAttributes* have the same name attribute.

If we want to retrieve the existing attribute from a session that is managed globally, we'll use *@SessionAttribute* annotation as a method parameter:

```
@GetMapping
public String getTodos(@SessionAttribute("todos") TodoList todos) {
    // method body
    return "todoView";
}
```

Q10. What is the Purpose of *@EnableWebMVC*?

The *@EnableWebMvc* annotation's purpose is to enable Spring MVC via Java configuration. It's equivalent to *<mvc: annotation-driven>* in an XML configuration. This annotation imports Spring MVC Configuration from *WebMvcConfigurationSupport*. It enables support for *@Controller*-annotated classes that use *@RequestMapping* to map incoming requests to a handler method.

You can learn more about this and similar annotations in our Guide to the Spring *@Enable* Annotations.

Q11. What Is *ViewResolver* in Spring?

The *ViewResolver* enables an application to render models in the browser – without tying the implementation to a specific view technology – **by mapping view names to actual views**.

For more details about the *ViewResolver*, have a look at our Guide to the ViewResolver in Spring MVC.

Q12. What is the *BindingResult*?

***BindingResult* is an interface from *org.springframework.validation* package that represents binding results. We can use it to detect and report errors in the submitted form**. It's easy to invoke — we just need to ensure that we put it as a parameter right after the form object we're validating. The optional *Model* parameter should come after the *BindingResult*, as it can be seen in the custom validator tutorial:

```
@PostMapping("/user")
public String submitForm(@Valid NewUserForm newUserForm,
  BindingResult result, Model model) {
    if (result.hasErrors()) {
        return "userHome";
```

```
    }
    model.addAttribute("message", "Valid form");
    return "userHome";
}
```

When Spring sees the *@Valid* annotation, it'll first try to find the validator for the object being validated. Then it'll pick up the validation annotations and invoke the validator. Finally, it'll put found errors in the *BindingResult* and add the latter to the view model.

Q13. What is a Form Backing Object?

The form backing object or a Command Object is just a POJO that collects data from the form we're submitting.

We should keep in mind that it doesn't contain any logic, only data.

To learn how to use form backing object with the forms in Spring MVC, please take a look at our article about Forms in Spring MVC.

Q14. What Is the Role of the *@Qualifier* Annotation?

It is used simultaneously with the *@Autowired* annotation to avoid confusion when multiple instances of a bean type are present.

Let's see an example. We declared two similar beans in XML config:

```
<bean id="person1" class="com.baeldung.Person" >
    <property name="name" value="Joe" />
</bean>
<bean id="person2" class="com.baeldung.Person" >
    <property name="name" value="Doe" />
</bean>
```

When we try to wire the bean, we'll get an *org.springframework.beans.factory.NoSuchBeanDefinitionException.* To fix it, we need to use *@Qualifier* to tell Spring about which bean should be wired:

```
@Autowired
@Qualifier("person1")
private Person person;
```

Q15. What Is the Role of the *@Required* Annotation?

The *@Required* annotation is used on setter methods, and it indicates that the bean property that has this annotation must be populated at configuration time. Otherwise, the Spring container will throw a *BeanInitializationException* exception.

Also, *@Required* differs from *@Autowired* – as it is limited to a setter, whereas *@Autowired* is not. *@Autowired* can be used to wire with a constructor and a field as well, while *@Required* only checks if the property is set.

Let's see an example:

```
public class Person {
    private String name;

    @Required
    public void setName(String name) {
        this.name = name;
    }
}
```

Now, the *name* of the *Person* bean needs to be set in XML config like this:

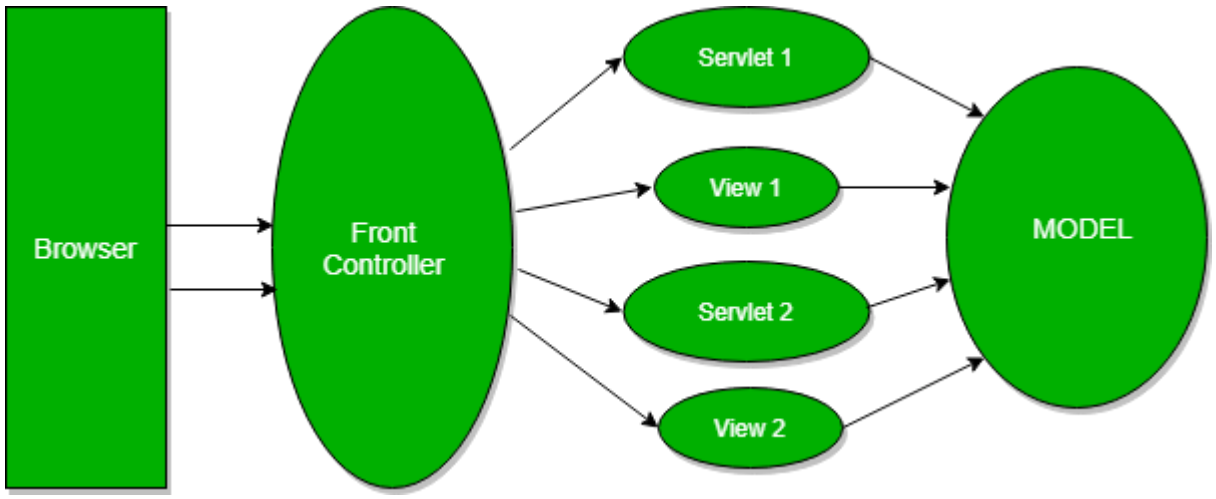
```
<bean id="person" class="com.baeldung.Person">
    <property name="name" value="Joe" />
</bean>
```

Please note that ***@Required* doesn't work with Java based *@Configuration* classes by default**. If you need to make sure that all your properties are set, you can do so when you create the bean in the *@Bean* annotated methods.

Q16. Describe the Front Controller Pattern

In the Front Controller pattern, all requests will first go to the front controller instead of the servlet. It'll make sure that the responses are ready and will send them back to the browser. This way we have one place where we control everything that comes from the outside world.

The front controller will identify the servlet that should handle the request first. Then, when it gets the data back from the servlet, it'll decide which view to render and, finally, it'll send the rendered view back as a response:

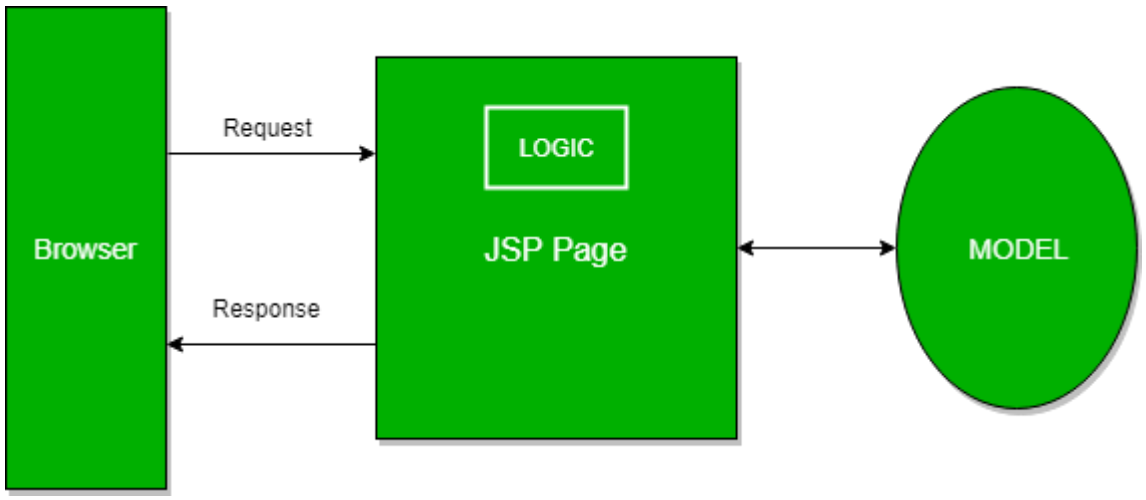


To see the implementation details, please check out our Guide to the Front Controller Pattern in Java.

Q17. What Are Model 1 and Model 2 Architectures?

Model 1 and Model 2 represent two frequently used design models when it comes to designing Java Web Applications.

In Model 1, a request comes to a servlet or JSP where it gets handled. The servlet or the JSP processes the request, handles business logic, retrieves and validates data, and generates the response:



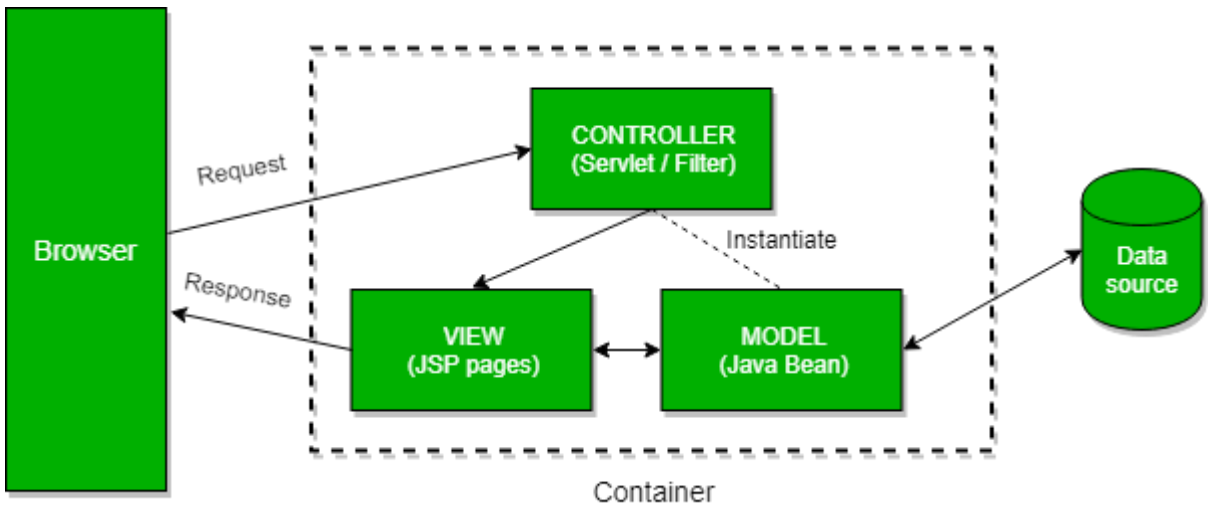
Since this architecture is easy to implement, we usually use it in small and simple applications.

On the other hand, it isn't convenient for large-scale web applications. The functionalities are often duplicated in JSPs where business and presentation logic are coupled.

The Model 2 is based on the Model View Controller design pattern and it separates the view from the logic that manipulates the content.

Furthermore, we can distinguish three modules in the MVC pattern: the model, the view, and the controller. The model is representing the dynamic data structure of an application. It's responsible for the data and business logic manipulation. The view is in charge of displaying the data, while the controller serves as an interface between the previous two.

In Model 2, a request is passed to the controller, which handles the required logic in order to get the right content that should be displayed. The controller then puts the content back into the request, typically as a JavaBean or a POJO. It also decides which view should render the content and finally passes the request to it. Then, the view renders the data:



Junit

1. What is Testing?

Testing is the process of checking the functionality of the application whether it is working as per requirements.

2. What is Unit Testing?

Unit testing is the testing of single entity (class or method). Unit testing is very essential to every software company to give a quality product to their customers.

3. What is JUnit?

JUnit is a Regression Testing Framework used by developers to implement unit testing in Java and accelerate programming speed and increase the quality of code.

JUnit is the testing framework, it is used for unit testing of Java code.

JUnit = Java + Unit Testing

4. What is Manual testing?

Executing the test cases manually without any tool support is known as manual testing.

5. What is Automated testing?

Taking tool support and executing the test cases by using automation tool is known as automation testing.

6. What is the difference between manual testing and automated testing?

Manual testing is performed by Human, so it is time-consuming and costly. Automated testing is performed by testing tools or programs, so it is fast and less costly.

Give some disadvantages of manual testing.

Following are some disadvantages of manual testing:

- The testing is very time consuming and is very tiring.
- The testing demands a very big investment in the human resources.
- The testing is less reliable
- The testing cannot be programmed.

7. What are important features of JUnit?

Import features of JUnit are:

- It is an open source framework.
- Provides Annotation to identify the test methods.
- Provides Assertions for testing expected results.
- Provides Test runners for running tests.

- JUnit tests can be run automatically and they check their own results and provide immediate feedback.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if test is going fine and it turns red when a test fails.
- Go through the JUnit Video to get clear understanding of JUnit.

8. How to install JUnit?

Installation steps for JUnit :

Download the latest version of JUnit, referred to below as junit.zip.

Unzip the junit.zip distribution file to a directory referred to as %JUNIT_HOME%.

Add JUnit to the classpath −

set CLASSPATH=%CLASSPATH%;%JUNIT_HOME%\junit.jar

Test the installation by running the sample tests distributed with JUnit (sample tests are located in the installation directory directly, not the junit.jar file).

Then simply type −

java org.junit.runner.JUnitCore org.junit.tests.AllTests

All the tests should pass with an “OK” message. If the tests don’t pass, verify that junit.jar is in the CLASSPATH.

9. What is a Unit Test Case?

A Unit Test Case is a part of code which ensures that the another part of code (method) works as expected. A formal written unit test case is characterized by a known input and by an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post condition.

10. Why does JUnit only report the first failure in a single test?

Reporting multiple failures in a single test is generally a sign that the test does too much and it is too big a unit test. JUnit is designed to work best with a number of small tests. It executes each test within a separate instance of the test class. It reports failure on each test.

11. In Java, assert is a keyword. Won’t this conflict with JUnit’sassert() method?

JUnit 3.7 deprecated assert() and replaced it with assertTrue(), which works exactly the same way. JUnit 4 is compatible with the assert keyword. If you run with the -ea JVM switch, assertions that fail will be reported by JUnit.

12. When are the unit tests pertaining to JUnit written in Developmental Cycle?

The unit tests are written before the development of the application. It is so because by writing the check before coding, it assists the coders to write error-free codes which further boosts the viability of the form.

13. Shed light on the variety of JUnit classes and make a proper list of them

The JUnit classes are essential classes that are usually utilized in testing and writing the JUnits. Here is the list of the critical JUnit test classes.

Test Suite: It is also known as a composition of various tests

Assert: It is a set of assertive procedures used to design an application

Test Result: It is associated with the collection of results while executing a test case

Test Case: It is that kind of a JUnit class which is related to various fixtures. It also has the ability to run on a variety of tests

Mockito

1) What is mockito?

Mockito is a **JAVA-based** library used for **unit testing** applications. This open-source library plays an important role in automated unit tests for the purpose of **test-driven development** or **behavior-driven development.** It uses a mock interface to add dummy functionality in the unit testing. It also uses Java reflection to create mock objects for an interface to test it.

2) List some benefits of Mockito?

Some of the benefits of Mockito are,

- It has support for return values.
- It supports exceptions.
- It has support for the creation of mock using annotation.
- There is no need to write mock objects on your own with Mockito.
- The test code is not broken when renaming interface method names or reordering parameters.

3) what is mocking in testing?

Mocking in Mockito is the way to test the functionality of the class. The mock objects do the mocking process of real service in isolation. These mock objects return the dummy data corresponding to the dummy input passed to the function. The mocking process does not require you to connect to the database or file server to test functionality.

4) What is difference between Assert and Verify in mockito?

Both statements are used to add validations to the test methods in the test suites but they differ in the following.

The Assert command is used to validate critical functionality. If this validation fails, then the execution of that test method is stopped and marked as failed.

In the case of **Verify** command, the test method continues the execution even after the failure of an assertion statement. The test method will be marked as failed but the execution of remaining statements of the test method is executed normally.

5) List some limitations of Mockito?

Some limitations of the mockito are,

- It cannot mock constructors or static methods.
- It requires Java version 6 plus to run.
- It also cannot mock equals(), hashCode() methods.
- VM mocking is only possible on VMs that are supported by Objenesis.

Download Free : Mockito Interview Questions PDF

6) What is use of mock() method in Mockito?

The Mock() method is used to create and inject the mocked instances. It gives you boilerplate assignments to work with these instances. The other way of creating the instances is using the **@mock** annotations.

7) What is ArgumentCaptor in Mockito?

ArgumentCaptor is a class that is used to capture the argument values for future assertions. This class is defined in the **org.mockito** package and can be imported from it.

Some of the methods present in this class are

- capture(),
- getValue(),
- getAllValues(), and ArgumentCaptor <U> forClass.

8) What is Hamcrest ?

Hamcrest is a framework used for writing customized assertion **matchers** in the Java programming language. It allows the match rules to be defined declaratively. This makes the **hamcrest** valuable in **UI validation**, **data filtering**, **writing flexible tests**, etc. It can also be used with mock objects by using adaptors. Hamcrest can also e used with **JUnit** and **TestNG**.

9) List some Mockito Annotations?

Some of the Mockito annotations are,

- @Mock - It is used to create and inject mocked instances.
- @Spy - It is used to create a real object and spy on the real object.
- @Captor - It is used to create an ArgumentCaptor.
- @InjectMocks - It is used to create an object of a class and insert its dependencies.

10) What is PowerMock?

PowerMock is a Java framework for unit testing purposes. This framework extends from other mock libraries with more powerful capabilities. It uses custom classloader and bytecode manipulation for **mocking the static methods**, **constructors**, **final classes**, **private methods**, and more. It normally lets you test the code that is regarded as untestable.

11) What is EasyMock?

EasyMock is a framework for creating mock objects as it uses Java reflection to create it for a given interface. It relieves the user of hand-writing mock objects as it uses a dynamic mock object generator.

Some other perks you get with EasyMock are

- exception support,
- return value support,
- refactoring scale,
- annotation support and order check support.

SQL

1. What is DBMS?

A Database Management System (DBMS) is a program that controls creation, maintenance and use of a database. DBMS can be termed as File Manager that manages data in a database rather than saving it in file systems.

2. What is RDBMS?

RDBMS stands for Relational Database Management System. RDBMS store the data into the collection of tables, which is related by common fields between the columns of the table. It also provides relational operators to manipulate the data stored into the tables.

Example: SQL Server.

3. What is SQL?

SQL stands for Structured Query Language , and it is used to communicate with the Database. This is a standard language used to perform tasks such as retrieval, updation, insertion and deletion of data from a database.

Standard SQL Commands are Select.

4. What is a Database?

Database is nothing but an organized form of data for easy access, storing, retrieval and managing of data. This is also known as structured form of data which can be accessed in many ways.

Example: School Management Database, Bank Management Database.

5. What are tables and Fields?

A table is a set of data that are organized in a model with Columns and Rows. Columns can be categorized as vertical, and Rows are horizontal. A table has specified number of column called fields but can have any number of rows which is called record.

Example:.

Table: Employee.

Field: Emp ID, Emp Name, Date of Birth.

Data: 201456, David, 11/15/1960.

6. What is a primary key?

A primary key is a combination of fields which uniquely specify a row. This is a special kind of unique key, and it has implicit NOT NULL constraint. It means, Primary key values cannot be NULL.

7. What is a unique key?

A Unique key constraint uniquely identified each record in the database. This provides uniqueness for the column or set of columns.

A Primary key constraint has automatic unique constraint defined on it. But not, in the case of Unique Key.

There can be many unique constraint defined per table, but only one Primary key constraint defined per table.

8. What is a foreign key?

A foreign key is one table which can be related to the primary key of another table. Relationship needs to be created between two tables by referencing foreign key with the primary key of another table.

9. What is a join?

This is a keyword used to query data from more tables based on the relationship between the fields of the tables. Keys play a major role when JOINs are used.

10. What are the types of join and explain each?

There are various types of join which can be used to retrieve data and it depends on the relationship between tables.

- **Inner Join.**

Inner join return rows when there is at least one match of rows between the tables.

- **Right Join.**

Right join return rows which are common between the tables and all rows of Right hand side table. Simply, it returns all the rows from the right hand side table even though there are no matches in the left hand side table.

- **Left Join.**

Left join return rows which are common between the tables and all rows of Left hand side table. Simply, it returns all the rows from Left hand side table even though there are no matches in the Right hand side table.

- **Full Join.**

Full join return rows when there are matching rows in any one of the tables. This means, it returns all the rows from the left hand side table and all the rows from the right hand side table.

11. What is normalization?

Normalization is the process of minimizing redundancy and dependency by organizing fields and table of a database. The main aim of Normalization is to add, delete or modify field that can be made in a single table.

12. What is Denormalization.

DeNormalization is a technique used to access the data from higher to lower normal forms of database. It is also process of introducing redundancy into a table by incorporating data from the related tables.

13. What are all the different normalizations?

The normal forms can be divided into 5 forms, and they are explained below -.

- **First Normal Form (1NF):.**

This should remove all the duplicate columns from the table. Creation of tables for the related data and identification of unique columns.

- **Second Normal Form (2NF):.**

Meeting all requirements of the first normal form. Placing the subsets of data in separate tables and Creation of relationships between the tables using primary keys.

- **Third Normal Form (3NF):.**

This should meet all requirements of 2NF. Removing the columns which are not dependent on primary key constraints.

- **Fourth Normal Form (4NF):.**

Meeting all the requirements of third normal form and it should not have multi- valued dependencies.

14. What is a View?

A view is a virtual table which consists of a subset of data contained in a table. Views are not virtually present, and it takes less space to store. View can have data of one or more tables combined, and it is depending on the relationship.

15. What is an Index?

An index is performance tuning method of allowing faster retrieval of records from the table. An index creates an entry for each value and it will be faster to retrieve data.

16. What are all the different types of indexes?

There are three types of indexes -.

- **Unique Index.**

This indexing does not allow the field to have duplicate values if the column is unique indexed. Unique index can be applied automatically when primary key is defined.

- **Clustered Index.**

This type of index reorders the physical order of the table and search based on the key values. Each table can have only one clustered index.

- **NonClustered Index.**

NonClustered Index does not alter the physical order of the table and maintains logical order of data. Each table can have 999 nonclustered indexes.

17. What is a Cursor?

A database Cursor is a control which enables traversal over the rows or records in the table. This can be viewed as a pointer to one row in a set of rows. Cursor is very much useful for traversing such as retrieval, addition and removal of database records.

18. What is a relationship and what are they?

Database Relationship is defined as the connection between the tables in a database. There are various data basing relationships, and they are as follows:.

- One to One Relationship.
- One to Many Relationship.
- Many to One Relationship.
- Self-Referencing Relationship.

19. What is a query?

A DB query is a code written in order to get the information back from the database. Query can be designed in such a way that it matched with our expectation of the result set. Simply, a question to the Database.

20. What is subquery?

A subquery is a query within another query. The outer query is called as main query, and inner query is called subquery. SubQuery is always executed first, and the result of subquery is passed on to the main query.