Features Introduced in Spring 3?

- Java 5 Support
- Spring Expression Language (SpEL)
- IoC enhancements/Java based bean metadata are introduced - capabilities have been migrated into the core framework, such as @Configuration, @Bean, @Primary, @DependsOn, @Lazy, @Import and @Value.
- Declarative model validation using constraint annotations - such as @NotNull and @Max(23).
- Comprehensive REST support in Spring MVC
- Java EE 6 support -
    - Many of the Java EE 6 features such as
        - JPA 2.0
        - JSF 2.0
        - they work on non-EE 6 containers like Tomcat and J2EE 1.4 app servers.

- JSR 330 support - The javax.inject annotations introduced by JSR 330 are now supported.
- Annotation-based formatting - Bean fields can be automatically formatted and converted using annotations such as
    - @DateFimeFormat(iso=ISO.DATE) and
    - @NumberFormat(style=Style.CURRENCY).
- Support for embedded databases - org.springframework.jdbc.datasource.embedded package (HSQL, H2, and Derby).
- Object to XML Mapping(OXM)- The Object-to-XML (OXM) mapping functionality from the Spring Web Services project has been moved into the Core Spring Framework (org.springframework.oxm).
- Support for Hibernate 4.x (Spring 3.1)
- Support for Servlet 3 based asynchronous request processing (Spring3.1) -
    - The Spring MVC programming model now provides explicit Servlet 3 async support. @RequestMapping methods can return one of:
        - java.util.concurrent.Callable to complete processing in a separate thread managed by a task executor within Spring MVC.
        - org.springframework.web.context.request.async. DeferredResult to

Features or Enhancements introduced in Spring 4.0:

- Removed Deprecated Packages and Methods
- Java 8 Support
- Java EE 6 and 7 - Java EE version 6 or above is now considered the baseline for Spring Framework 4, with the JPA 2.0 and Servlet 3.0 specifications being of particular relevance. It is possible to run your application in Servlet 2.5, but it is recommended to use Servlet 3.0 environment.
- Groovy Bean Definition DSL
- Core Container Improvements:
- There have been several general improvements to the core container:
    - Spring now treats generic types as a form of qualifier when injecting Beans.
        - For example, if you are using a Spring Data Repository you can now easily inject a specific implementation: @Autowired Repository<Customer> customerRepository.
    - If you use Spring's meta-annotation support, you can now develop custom annotations that expose specific attributes from the source annotation.
    - Beans can now be Ordered when they are autowired into lists and arrays. Both the @Ordered annotation and Ordered interface are supported.
    - The @Lazy annotation can now be used on injection points, as well as @Bean definitions.
    - The @Description annotation has been added for developers using Java-based configuration.
    - A generalized model for conditionally filtering beans has been added via the @Conditional annotation. This is similar to @Profile but allows for user-defined strategies to be developed.
    - CGLIB-based proxy classes no longer require a default constructor. Support is provided via the objenesis library which is repackaged inline and distributed as part of the Spring Framework. With this strategy, no constructor at all is being invoked for proxy instances anymore.
    - There is managed time zone support across the framework now, e.g. on LocaleContext.
- General Web Improvements:
    - Deployment to Servlet 2.5 servers remains an option, but Spring Framework 4.0 is now focused primarily on Servlet 3.0+ environments. If you are using the Spring MVC Test Framework you will need to ensure that a Servlet 3.0 compatible JAR is in your test

complete processing at a later time from a thread not known to Spring MVC — for example, in response to some external event (JMS, AMQP, etc.)
        - org.springframework.web.context.request.async.AsyncTask to wrap a Callable and customize the timeout value or the task executor to use.
- Spring MVC Test framework (Spring 3.1)
- Refined Java SE 7 / OpenJDK 7 support (Spring 3.2)
- Support for Portlet 2.0 (JSR-286): Spring 3.x supports Spring Webmvc portlet 2.0 with the features Ajax support, Events, and Securing portlets.

Features introduced in Spring 5:

- Baseline upgrades, for e.g:
    - Servlet 3.1
    - JMS 2.0
    - JPA 2.1
    - JAX-RS 2.0
    - Bean Validation 1.1
    - Hibernate 5
    - Jackson 2.6
    - EhCache 2.10
    - JUnit 5
    - Tiles 3
    - Tomcat 8.5+
    - Jetty 9.4+
    - WildFly 10+
    - Netty 4.1+
    - Undertow 1.4+
- JDK 9 runtime compatibility -
    - Spring 5 release has been very well aligned with JDK 9 release dates. The goal is for Spring Framework 5.0 to go GA right after JDK 9's GA. Spring 5.0 release candidates are already supporting Java 9 on classpath as well as modulepath.
- Usage of JDK 8 features -
    - Spring 5 has baseline version 8, so it uses many new features of Java 8 and 9 as well.
    - Java 8 default methods in core Spring interfaces
    - Internal code improvements based on Java 8 reflection enhancements
    - Use of functional programming in the framework code – lambdas and streams
- Reactive programming support
- A functional web framework
- Kotlin support
- Dropped some support

classpath.
    - In addition to the WebSocket support mentioned later, the following general improvements have been made to Spring's Web modules:
        - You can use the new @RestController annotation with Spring MVC applications, removing the need to add @ResponseBody to each of your @RequestMapping methods.
        - The AsyncRestTemplate class has been added, allowing non-blocking asynchronous support when developing REST clients.
        - Spring now offers comprehensive timezone support when developing Spring MVC applications.
- WebSocket, SockJS, and STOMP Messaging
    - A new spring-websocket module provides comprehensive support for WebSocket-based, two-way communication between client and server in web applications. It is compatible with JSR-356, the Java WebSocket API, and in addition provides SockJS-based fallback options (i.e. WebSocket emulation) for use in browsers that don't yet support the WebSocket protocol (e.g. Internet Explorer < 10).
    - A new spring-messaging module adds support for STOMP as the WebSocket sub-protocol to use in applications along with an annotation programming model for routing and processing STOMP messages from WebSocket clients. As a result an @Controller can now contain both @RequestMapping and @MessageMapping methods for handling HTTP requests and messages from WebSocket-connected clients. The new spring-messaging module also contains key abstractions from the Spring Integration project such as Message, MessageChannel, MessageHandler, and others to serve as a foundation for messaging-based applications.
- Testing Improvements
    - In addition to pruning of deprecated code within the spring-test module, Spring Framework 4.0 introduces several new features for use in unit and integration testing.
    - Almost all annotations in the spring-test module (e.g., @ContextConfiguration, @WebAppConfiguration, @ContextHierarchy, @ActiveProfiles, etc.) can now be used as meta-annotations to create custom composed annotations and reduce configuration duplication across a test suite.

- Active bean definition profiles can now be resolved programmatically, simply by implementing a custom ActiveProfilesResolver and registering it via the resolver attribute of @ActiveProfiles.
- A new SocketUtils class has been introduced in the spring-core module which enables you to scan for free TCP and UDP server ports on localhost. This functionality is not specific to testing but can prove very useful when writing integration tests that require the use of sockets, for example tests that start an in-memory SMTP server, FTP server, Servlet container, etc.
- As of Spring 4.0, the set of mocks in the org.springframework.mock.web package is now based on the Servlet 3.0 API. Furthermore, several of the Servlet API mocks (e.g., MockHttpServletRequest, MockServletContext, etc.) have been updated with minor enhancements and improved configurability.

What is Inversion of Control?

- Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework. It's most often used in the context of object-oriented programming.
- By contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.

The advantages of this architecture are:

- decoupling the execution of a task from its implementation
- making it easier to switch between different implementations
- greater modularity of a program
- greater ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts
- Inversion of Control can be achieved through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

What is Dependency Injection?

- Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies.
- The act of connecting objects with other objects, or "injecting" objects into other objects, is done by an assembler rather than by the objects themselves.
- Here's how you would create an object dependency in traditional programming:

```
public class Store {
   private Item item;
   public Store() {
      item = new ItemImpl1();
   }
}
```

- In the example above, we need to instantiate an implementation of the Item interface within the Store class itself.
- By using DI, we can rewrite the example without specifying the implementation of Item that we want:

```
public class Store {
   private Item item;
   public Store(Item item) {
      this.item = item;
   }
```

- Both IoC and DI are simple concepts, but have deep implications in the way we structure our systems, so they're well worth understanding well.

The Spring IoC Container:

- An IoC container is a common characteristic of frameworks that implement IoC.

Constructor-Based Dependency Injection:

- In the Spring framework, the IoC container is represented by the interface ApplicationContext. The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their lifecycle.
- The Spring framework provides several implementations of the ApplicationContext interface — ClassPathXmlApplicationContext and FileSystemXmlApplicationContext for standalone applications, and WebApplicationContext for web applications.
- In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.
- Here's one way to manually instantiate a container:

```
ApplicationContext context
  = new
ClassPathXmlApplicationContext("applicationContext.
xml");
```

- To set the item attribute in the example above, we can use metadata. Then, the container will read this metadata and use it to assemble beans at runtime.
- Dependency Injection in Spring can be done through constructors, setters or fields.

Setter-Based Dependency Injection:

- For setter-based DI, the container will call setter methods of our class, after invoking a no-argument constructor or no-argument static factory method to instantiate the bean. Let's create this configuration using annotations:

```
@Bean
public Store store() {
   Store store = new Store();
   store.setItem(item1());
   return store;
}
```

- We can also use XML for the same configuration of beans:

```
<bean id="store" class="org.baeldung.store.Store">
   <property name="item" ref="item1" />
</bean>
```

- Constructor-based and setter-based types of injection can be combined for the same bean. The Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones.

Autowiring Dependencies:

- Wiring allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.
- In the case of constructor-based dependency injection, the container will invoke a constructor with arguments each representing a dependency we want to set.
- Spring resolves each argument primarily by type, followed by name of the attribute and index for disambiguation. Let's see the configuration of a bean and its dependencies using annotations:

```
@Configuration
public class AppConfig {
   @Bean
   public Item item1() {
      return new ItemImpl1();
   }
   @Bean
   public Store store() {
      return new Store(item1());
   }
}
```

- The @Configuration annotation indicates that the class is a source of bean definitions. Also, we can add it to multiple configuration classes.
- The @Bean annotation is used on a method to define a bean. If we don't specify a custom name, the bean name will default to the method name.
- For a bean with the default singleton scope, Spring first checks if a cached instance of the bean already exists and only creates a new one if it doesn't. If we're using the prototype scope, the container returns a new bean instance for each method call.
- Another way to create the configuration of the beans is through XML configuration:

```
<bean id="item1"
class="org.baeldung.store.ItemImpl1" />
<bean id="store" class="org.baeldung.store.Store">
   <constructor-arg type="ItemImpl1" index="0"
name="item" ref="item1" />
</bean>
```

Field-Based Dependency Injection:

- In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation:

```
public class Store {
   @Autowired
   private Item item;
}
```

- While constructing the Store object, if there's no constructor or setter method to inject the Item bean, the container will use reflection to inject Item into Store.
- We can also achieve this using XML configuration.
- This approach might look simpler and cleaner but is

- There are four modes of autowiring a bean using an XML configuration:
    - no: the default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies
    - byName: autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set
    - byType: similar to the byName autowiring, only based on the type of the property. This means Spring will look for a bean with the same type of the property to set. If there's more than one bean of that type, the framework throws an exception.
    - constructor: autowiring is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments

- For example, let's autowire the item1 bean defined above by type into the store bean:

```
@Bean(autowire = Autowire.BY_TYPE)
public class Store {
   private Item item;
   public setItem(Item item){
      this.item = item;
   }
}
```

- We can also inject beans using the @Autowired annotation for autowiring by type:

```
public class Store {
   @Autowired
   private Item item;
}
```

- If there's more than one bean of the same type, we can use the @Qualifier annotation to reference a bean by name:

```
public class Store {
   @Autowired
   @Qualifier("item1")
   private Item item;
}
```

- Now, let's autowire beans by type through XML configuration:

```
<bean id="store" class="org.baeldung.store.Store"
autowire="byType"> </bean>
```

- Next, let's inject a bean named item into the item property of store bean by name through XML:

```
<bean id="item" class="org.baeldung.store.ItemImpl1"
/>
```

not recommended to use because it has a few drawbacks such as:

- This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection
- It's really easy to keep adding multiple dependencies using this approach. If you were using constructor injection having multiple arguments would have made us think that the class does more than one thing which can violate the Single Responsibility Principle.

Lazy Initialized Beans:

- By default, the container creates and configures all singleton beans during initialization. To avoid this, you can use the lazy-init attribute with value true on the bean configuration:

```
<bean id="item1"
class="org.baeldung.store.ItemImpl1" lazy-init="true"
/>
```

- As a consequence, the item1 bean will be initialized only when it's first requested, and not at startup. The advantage of this is faster initialization time, but the trade-off is that configuration errors may be discovered only after the bean is requested, which could be several hours or even days after the application has already been running.

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 Java EE platform, or we may just use its dependency injection provisions in simple standalone applications.

What are the benefits of using Spring?

Spring targets to make Java 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

```
<bean id="store" class="org.baeldung.store.Store"
autowire="byName">
</bean>
```

- We can also override the autowiring by defining dependencies explicitly through constructor arguments or setters.

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.

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.

- 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

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.

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.

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.

What is a Spring Bean?

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

What is the default bean scope in Spring framework?

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

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:

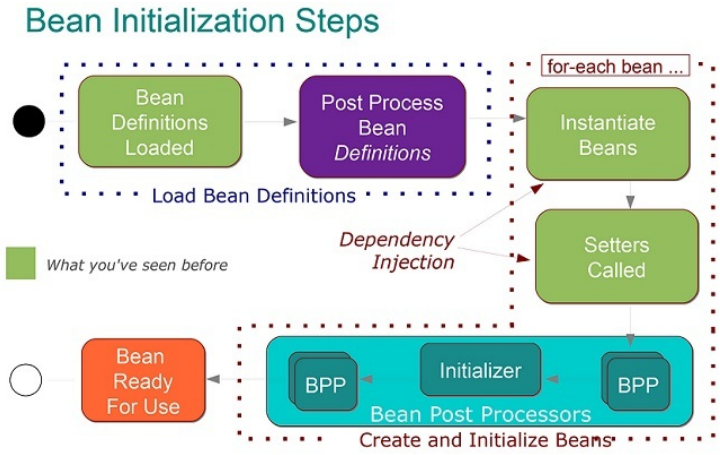
- Singleton:
  - Scopes a single bean definition to a single object instance per Spring IoC container.
- Prototype:
  - Scopes a single bean definition to any number of object instances.
- Request:
  - Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
- Session:

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.

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.



What is the Spring Java-Based Configuration?

- It's one of the ways of configuring Spring-based applications in a type-safe manner. It's an alternative to the XML-based configuration.

What is Spring Security?

- Spring Security is a separate module of the Spring framework that focuses on providing authentication and authorization methods in Java applications. It also takes care of most of the common security vulnerabilities such as CSRF attacks.
- To use Spring Security in web applications, you can get started with a simple annotation: @EnableWebSecurity.

What is Spring Boot?

- Spring Boot is a project that provides a pre-configured set of frameworks to reduce boilerplate configuration so that you can have a Spring application up and running with the smallest amount of code.

Name some of the Design Patterns used in the Spring Framework?

- Singleton Pattern: Singleton-scoped beans
- Factory Pattern: Bean Factory classes
- Prototype Pattern: Prototype-scoped beans
- Adapter Pattern: Spring Web and Spring MVC

    - Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.
- Global-session:
    - Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

Can we have multiple Spring configuration files in one project?

- Yes, in large projects, having multiple Spring configurations is recommended to increase maintainability and modularity.
- You can load multiple Java-based configuration files:
  @Configuration
  @Import({MainConfig.class, SchedulerConfig.class})
  public class AppConfig {
- Or load one XML file that will contain all other configs:
  ApplicationContext context = new ClassPathXmlApplicationContext("spring-all.xml");
- And inside this XML file you'll have:
  <import resource="main.xml"/>
  <import resource="scheduler.xml"/>

How does the scope Prototype work?

- Scope prototype means that every time you call for an instance of the Bean, Spring will create a new instance and return it.
- This differs from the default singleton scope, where a single object instance is instantiated once per Spring IoC container.

- Proxy Pattern: Spring Aspect Oriented Programming support
- Template Method Pattern: JdbcTemplate, HibernateTemplate, etc.
- Front Controller: Spring MVC DispatcherServlet
- Data Access Object: Spring DAO support
- Model View Controller: Spring MVC

What is a Controller in Spring MVC?

- Simply put, all the requests processed by the DispatcherServlet are directed to classes annotated with @Controller. Each controller class maps one or more requests to methods that process and execute the requests with provided inputs.
- If you need to take a step back, we recommend having a look at the concept of the Front Controller in the typical Spring MVC architecture.

What is Spring JDBCTemplate class and how to use it?

- The Spring JDBC template is the primary API through which we can access database operations logic that we're interested in:
    - creation and closing of connections
    - executing statements and stored procedure calls
    - iterating over the ResultSet and returning results
    - To use it, we'll need to define the simple configuration of DataSource:

```
@Configuration
@ComponentScan("org.baeldung.jdbc")
public class SpringJdbcConfig {
   @Bean
   public DataSource mysqlDataSource() {
      DriverManagerDataSource dataSource = new
DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driv
er");
dataSource.setUrl("jdbc:mysql://localhost:3306/sprin
gjdbc");
      dataSource.setUsername("guest_user");
      dataSource.setPassword("guest_password");
      return dataSource;
   }
}
```

What are Aspect, Advice, Pointcut, and JoinPoint in AOP?

- Aspect: a class that implements cross-cutting concerns, such as transaction management
- Advice: the methods that get executed when a specific JoinPoint with matching Pointcut is reached in the application
- Pointcut: a set of regular expressions that are matched with JoinPoint to determine whether Advice needs to be executed or not

How to Get ServletContext and ServletConfig Objects in a Spring Bean?

You can do either by:

- Implementing Spring-aware interfaces. The complete list is available here.
- Using @Autowired annotation on those beans:
  @Autowired
  ServletContext servletContext;

How does the @RequestMapping annotation work?

- The @RequestMapping annotation is used to map web requests to Spring Controller methods. In addition to simple use cases, we can use it for mapping of HTTP headers, binding parts of the URI with @PathVariable, and working with URI parameters and the @RequestParam annotation.

How would you enable transactions in Spring and what are their benefits?

- There are two distinct ways to configure Transactions – with annotations or by using Aspect Oriented Programming (AOP) – each with their advantages.
- The benefits of using Spring Transactions, according to the official docs, are:
    - Provide a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO
    - Support declarative transaction management
    - Provide a simpler API for programmatic transaction management than some complex transaction APIs such as JTA
    - Integrate very well with Spring's various data access abstractions

What is Spring DAO?

- Spring Data Access Object is Spring's support provided to work with data access technologies like JDBC, Hibernate, and JPA in a consistent and easy way.

What is Aspect-Oriented Programming?

- Aspects enable the modularization of cross-cutting concerns such as transaction management that span multiple types and objects by adding extra behavior to already existing code without modifying affected classes.

What is Weaving?

- weaving is a process that links aspects with other application types or objects to create an advised object.
- This can be done at compile time, load time, or at runtime.
- Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

- JoinPoint: a point during the execution of a program, such as the execution of a method or the handling of an exception

What is Spring WebFlux?

- Spring WebFlux is Spring's reactive-stack web framework, and it's an alternative to Spring MVC.
- In order to achieve this reactive model and be highly scalable, the entire stack is non-blocking.

What are the Mono and Flux types?

- The WebFlux framework in Spring Framework 5 uses Reactor as its async foundation.
- This project provides two core types:
  - Mono to represent a single async value, and
  - Flux to represent a stream of async values. They both implement the Publisher interface defined in the Reactive Streams specification.
- Mono implements Publisher and returns 0 or 1 elements:
  public abstract class Mono<T> implements Publisher<T> {...}
- Also, Flux implements Publisher and returns N elements:
  public abstract class Flux<T> implements Publisher<T> {...}
- By definition, the two types represent streams, hence they're both lazy, which means nothing is executed until we consume the stream using the subscribe() method.
- Both types are immutable, therefore calling any method will return a new instance of Flux or Mono.

What are the disadvantages of using Reactive Streams?

The major disadvantages of using reactive streams are:

- Troubleshooting a Reactive application is a bit difficult; be sure to check out our tutorial on debugging reactive streams for some handy debugging tips
- There is limited support for reactive data stores, as traditional relational data stores have yet to embrace the reactive paradigm
- There's an extra learning curve when implementing

Is Spring 5 compatible with older versions of Java?

- In order to take advantage of Java 8 features, the Spring codebase has been revamped. This means older versions of Java cannot be used.
- Hence, the framework requires a minimum of Java 8.

Can we use both Web MVC and WebFlux in the same application?

- As of now, Spring Boot will only allow either Spring MVC or Spring WebFlux, as Spring Boot tries to

What is reactive programming?

- Reactive programming is about non-blocking, event-driven applications that scale with a small number of threads, with back pressure being a key ingredient that aims to ensure producers don't overwhelm consumers.
- The primary benefits of reactive programming are:
  - increased utilization of computing resources on multicore and multi-CPU hardware
  - and increased performance by reducing serialization
- Reactive programming is generally event-driven, in contrast to reactive systems, which are message-driven.
- Thus, using reactive programming does not mean we're building a reactive system, which is an architectural style.
- However, reactive programming may be used as a means to implement reactive systems if we follow the Reactive Manifesto, which is quite vital to understand.
- Based on this, reactive systems have four important characteristics:
  - Responsive: the system should respond in a timely manner
  - Resilient: in case the system faces any failure, it should stay responsive
  - Elastic: reactive systems can react to changes and stay responsive under varying workload
  - Message-driven: reactive systems need to establish a boundary between components by relying on asynchronous message passing

What is the use of WebClient and WebTestClient?

- WebClient is a component in the new Web Reactive framework that can act as a reactive client for performing non-blocking HTTP requests. Being a reactive client, it can handle reactive streams with back pressure, and it can take full advantage of Java 8 lambdas. It can also handle both sync and async scenarios.
- On the other hand, the WebTestClient is a similar class that we can use in tests. Basically, it's a thin shell around the WebClient. It can connect to any server over an HTTP connection. It can also bind directly to WebFlux applications using mock request and response objects, without the need for an HTTP server.

How does Spring 5 integrate with JDK 9 modularity?

- In Spring 5, everything has been modularized, thus we won't be forced to import jars that may not have the functionalities we're looking for.
- Please have a look at our guide to Java 9 modularity for an in-depth understanding of how this technology works.

auto-configure the context depending on the dependencies that exist in its classpath.

- Also, Spring MVC cannot run on Netty. Moreover, MVC is a blocking paradigm and WebFlux is a non-blocking style, therefore we shouldn't be mixing both together, as they serve different purposes.

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.

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.

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.

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.

- Let's see an example to understand the new module functionality in Java 9 and how to organize a Spring 5 project based on this concept.
- To start, let's create a new class that contains a single method to return a String "HelloWorld". We'll place this within a new Java project – HelloWorldModule:

```
package com.hello;
public class HelloWorld {
   public String sayHello(){
      return "HelloWorld";
   }
}
```

- Then let's create a new module:

```
module com.hello {
   export com.hello;
}
```

- Now, let's create a new Java Project, HelloWorldClient, to consume the above module by defining a module:

```
module com.hello.client {
   requires com.hello;
}
```

- The above module will be available for testing now:

```
public class HelloWorldClient {
   public static void main(String[] args){
      HelloWorld helloWorld = new HelloWorld();
      log.info(helloWorld.sayHello());
   }
}
```

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.

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}:

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 JavaEE 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.

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.

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'

```
@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) {}
```

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.

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.

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.

What is the BindingResult?

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";
}
```

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.

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;
```

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:

- 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.

What is the Role of the @Required Annotation?

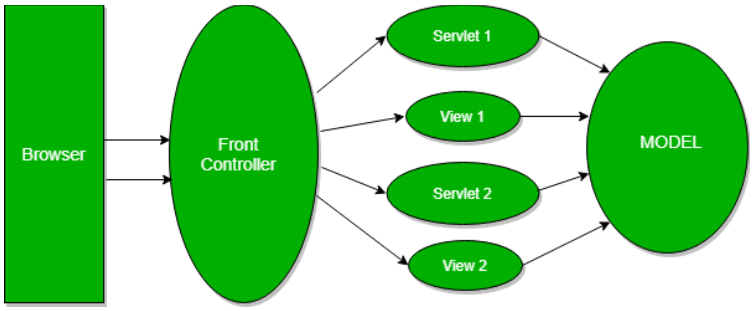
- The @Required annotation is used on setter methods, and it indicates that the bean property that has this annotation must be populated at configuration time. Otherwise, the Spring container will throw a BeanInitializationException exception.
- Also, @Required differs from @Autowired – as it is limited 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



What's the Difference Between @Controller, @Component, @Repository, and @Service Annotations in Spring?

- According to the official Spring documentation, @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.
- Let's take a look at specific use cases of last three:
    - @Controller – indicates that the class serves the role of a controller, and detects @RequestMapping annotations within the class
    - @Service – indicates that the class holds business logic and calls methods in the repository layer
    - @Repository – indicates that the class defines a data repository; its job is to catch platform-specific exceptions and re-throw them as one of Spring's unified unchecked exceptions

What are DispatcherServlet and ContextLoaderListener?

- Simply put, in the Front Controller design pattern, a single controller is responsible for directing incoming HttpRequests to all of an application's other controllers and handlers.
- Spring's DispatcherServlet implements this pattern and is, therefore, responsible for correctly coordinating the HttpRequests to the right handlers.
- On the other hand, ContextLoaderListener starts up and shuts down Spring's root WebApplicationContext. It ties the lifecycle of ApplicationContext to the lifecycle of the ServletContext. We can use it to define shared beans working across different Spring contexts.

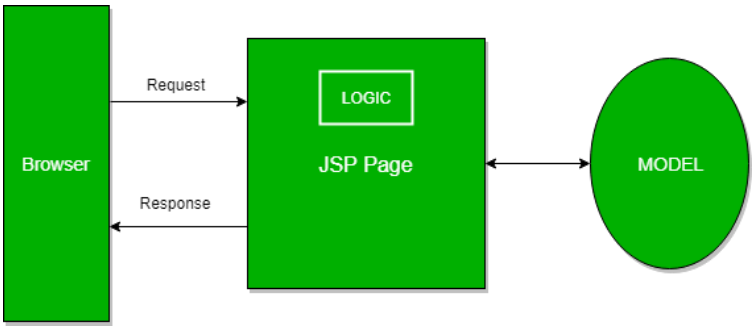
What is a MultipartResolver and When Should We Use It?

- The MultipartResolver interface is used for uploading files. The Spring framework provides one MultipartResolver implementation for use with Commons FileUpload and another for use with Servlet 3.0 multipart request parsing.
- Using these, we can support file uploads in our web

do so when you create the bean in the @Bean annotated methods.

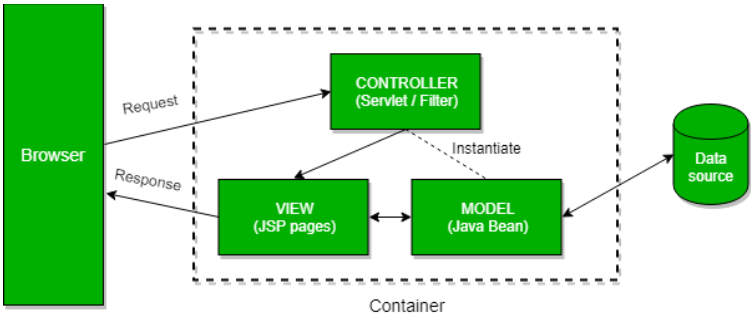
What are Model 1 and Model 2 Architectures?

- 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:



applications.

What is Spring MVC Interceptor and How to Use It?

- Spring MVC Interceptors allow us to intercept a client request and process it at three places – before handling, after handling, or after completion (when the view is rendered) of a request.
- The interceptor can be used for cross-cutting concerns and to avoid repetitive handler code like logging, changing globally used parameters in Spring model, etc.

Explain a Controller Advice

- The @ControllerAdvice annotation allows us to write global code applicable to a wide range of controllers. We can tie the range of controllers to a chosen package or a specific annotation.
- By default, @ControllerAdvice applies to the classes annotated with @Controller (or @RestController). We also have a few properties that we use if we want to be more specific.
- If we want to restrict applicable classes to a package, we should add the name of the package to the annotation:
  @ControllerAdvice("my.package")
  @ControllerAdvice(value = "my.package")
  @ControllerAdvice(basePackages = "my.package")
- It's also possible to use multiple packages, but this time we need to use an array instead of the String.
- Besides restricting to the package by its name, we can do it by using one of the classes or interfaces from that package:
  @ControllerAdvice(basePackageClasses = MyClass.class)
- The 'assignableTypes' element applies the @ControllerAdvice to the specific classes, while 'annotations' does it for particular annotations.
- It's noteworthy to remember that we should use it along with @ExceptionHandler. This combination will enable us to configure a global and more specific error handling mechanism without the need to implement it every time for every controller class.

Exception Handling in Web Applications

- We have three options for exceptions handling in Spring MVC:
  - per exception
  - per controller
  - globally
- If an unhandled exception is thrown during web request processing, the server will return an HTTP 500 response. To prevent this, we should annotate any of

What is an Init Binder?

- A method annotated with @InitBinder is used to customize a request parameter, URI template, and backing/command objects. We define it in a controller and it helps in controlling the request. In this method, we register and configure our custom PropertyEditors, a formatter, and validators.
- The annotation has the 'value' element. If we don't set it, the @InitBinder annotated methods will get called on each HTTP request. If we set the value, the methods will be applied only for particular command/form attributes and/or request parameters whose names correspond to the 'value' element.
- It's important to remember that one of the arguments must be WebDataBinder. Other arguments can be of any type that handler methods support except for command/form objects and corresponding validation result objects.

What Does the @ExceptionHandler Annotation Do?

- The @ExceptionHandler annotation allows us to define a method that will handle the exceptions. We may use the annotation independently, but it's a far better option to use it together with the @ControllerAdvice. Thus, we can set up a global error handling mechanism. In this way, we don't need to write the code for the exception handling within every controller.
- Let's take a look at the example from our article about Error Handling for REST with Spring:

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = {
IllegalArgumentException.class,
      IllegalStateException.class })
    protected ResponseEntity<Object>
handleConflict(RuntimeException ex,
      WebRequest request) {
       String bodyOfResponse = "This should be
application specific";
       return handleExceptionInternal(ex,
bodyOfResponse, new HttpHeaders(),
         HttpStatus.CONFLICT, request);
    }
}
```

- We should also note that this will provide @ExceptionHandler methods to all controllers that throw IllegalArgumentException or IllegalStateException. The exceptions declared with @ExceptionHandler should match the exception used

our custom exceptions with the @ResponseStatus annotation. This kind of exceptions is resolved by HandlerExceptionResolver.

- This will cause the server to return an appropriate HTTP response with the specified status code when a controller method throws our exception. We should keep in mind that we shouldn't handle our exception somewhere else for this approach to work.
- Another way to handle the exceptions is by using the @ExceptionHandler annotation. We add @ExceptionHandler methods to any controller and use them to handle the exceptions thrown from inside that controller. These methods can handle exceptions without the @ResponseStatus annotation, redirect the user to a dedicated error view, or build a totally custom error response.
- We can also pass in the servlet-related objects (HttpServletRequest, HttpServletResponse, HttpSession, and Principal) as the parameters of the handler methods. But, we should remember that we can't put the Model object as the parameter directly.
- The third option for handling errors is by @ControllerAdvice classes. It'll allow us to apply the same techniques, only this time at the application level and not only to the particular controller. To enable this, we need to use the @ControllerAdvice and the @ExceptionHandler together. This way exception handlers will handle exceptions thrown by any controller.

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>2.1.1.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 cannot rely on the Spring Boot starter parent.
- In this case, we can still get the benefits of dependency management with this POM element:

as the argument of the method. Otherwise, the exception resolving mechanism will fail at runtime.

- One thing to keep in mind here is that it's possible to define more than one @ExceptionHandler for the same exception. We can't do it in the same class though since Spring would complain by throwing an exception and failing on startup.
- On the other hand, if we define those in two separate classes, the application will start, but it'll use the first handler it finds, possibly the wrong one.

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..

What Spring Boot starters are available out there?

- Dependency management is a crucial facet of any project. When a project is complex enough, managing dependencies may turn into a nightmare, as there will be too many artifacts involved.
- This is where Spring Boot starters come in handy. 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

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>
      <version>2.1.1.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- Finally, we can add some dependencies to Spring Boot starters, and then we're good to go.

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.

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() { ... }
}
```

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
  - spring-boot-starter-data-jpa: starter for using Spring Data JPA with Hibernate
  - spring-boot-starter-jdbc: starter for using JDBC with the HikariCP connection pool
  - 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

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

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

```
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.

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.
- Here are the most common sources of external configuration:
    - Command-line properties: Command-line option arguments are program arguments starting with a double hyphen, such as –server.port=8080. Spring Boot converts all the arguments to properties and adds them to the set of environment properties.
    - Application properties: Application properties are those loaded from the application.properties file or its YAML counterpart. By default, Spring Boot searches for this file in the current directory, classpath root, or their config subdirectory.
    - Profile-specific properties: Profile-specific properties are loaded from the application-{profile}.properties file or its YAML counterpart. The {profile} placeholder refers to an active profile. These files are in the same locations as, and take precedence over, non-specific property files.

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.

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.

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 an environment 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.

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

- 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.
- Notice if we use JUnit 4, we must decorate the test class with @RunWith(SpringRunner.class).

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:
  - auditevents: Exposes audit events information
  - 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
  - scheduledtasks: Displays the scheduled tasks in your application
  - threaddump: Performs a thread dump

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.

Note: Information gathered in this document has been collected from various sources on Internet.

Sources:

https://www.baeldung.com/

https://www.baeldung.com/spring-interview-questions

https://www.baeldung.com/spring-boot-interview-questions

https://www.baeldung.com/spring-mvc-interview-questions