Use @Value annotation.

Sometimes we want beans to be able to communicate among themselves. This is typically done by Sender bean having access to Receiver bean, and then sender sends a message to receiver by using the reference to the receiver. This creates a tight coupling between sender and receiver. Sender is aware of the receiver.
When using IoC POJOs can communicate by interface than by implementation. This model helps reduce coupling. However it is efficient only when the sender has to communicate with 1 receiver. When the sender has to communicate with multiple receivers it has to call the receivers 1 by 1.
In Spring 1 component can send a message to another component by publishing an event without knowing who the receiver is. Receiver is also not aware of the sender. Receiver typically listens to an event. It can listen to multiple events from different senders. In this way, sender and receiver are loosely coupled.
First you must create an event, by extending ApplicationEvent class. It is an abstract class to prevent instantiation thus disallowing clients to create instances of ApplicationEvent class, prohibiting them from publishing generic event. Spring notifies a listener of all events, so you must filter the events yourself. If you use generics then Spring delivers the message to listener that listens specifically that event.

Create and event by extending ApplicationEvent class. Give it a constructor that accepts an object that describes the event and pass that object to super class ApplicationEvent. Application class is abstract and does not have default constructor, so subclasses are forced to provide a 1 or more arg constructor.

```
public class CheckoutEvent extends ApplicationEvent
{
    private Date checkoutTime;
    public CheckoutEvent(Object source, Date checkoutTime)
    {
        super(source);
        this.checkoutTime = checkoutTime;
    }…
}
```

Create an event publisher by implanting `ApplicationEventPublisherAware` interface. It encapsulates the even publishing mechanism. Override the `setApplicationEventPublisher` reference. Spring will take care of calling it with an ApplicationEventPublisher object.

In the event publisher method just create the event and publish it by using calling publishEvent(event) on `applicationEventPublisher` instance.

```
@Component(value = "checkoutEventPublisher")
public class CheckoutEventPublisher implements ApplicationEventPublisherAware
{
    private ApplicationEventPublisher applicationEventPublisher;
    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher
applicationEventPublisher)
    {
        this.applicationEventPublisher = applicationEventPublisher;
    }
    public void checkout(Object shoppingCart)
    {
        System.out.println("Checking out");
```

```
        CheckoutEvent checkoutEvent = new CheckoutEvent(shoppingCart, new Date());
        applicationEventPublisher.publishEvent(checkoutEvent);
    }
}
```

Create a listener by implementing ApplicationListener interface and overriding onApplicationEvent(ApplicationEvent) method. Add the code for processing the event in it.
You must create a bean for the listener, else it won't be notified.

Generic listener (Can listen to only 1 event)

```
@Component
public class CheckoutEventListener implements  ApplicationListener<CheckoutEvent>
{
    @Override
    public void onApplicationEvent(CheckoutEvent checkoutEvent) {

        System.out.println("Checkout event occured at " +
checkoutEvent.getCheckoutTime());
    }
}
```

Raw listener(Can listen to several events from several publishers)

```
@Component
public class CheckoutEventListener implements  ApplicationListener
{
    @Override
    public void onApplicationEvent(ApplicationEvent event)
    {
        if(event instanceof CheckoutEvent)
            System.out.println("Checkout event occured at " +
((CheckoutEvent)event).getCheckoutTime());
    }
}
```

Is init-method called before constructor or after, why to use init-method when you can have that code in constructor.
Init method is called after constructor runs. Destroy-method is called when context is closed.

Java config classes are decorated with @Configuration annotation. It tells Spring that the decorated class contains bean definitions. It will look for methods annotated with @Bean. The java methods create and return and bean instance. They return a bean with same name as method name. Alternatively, you can explicitly specify bean name in @Bean annotation with the name attribute. E.g. @Bean(name="myBean") creates a bean with name myBean. If you explicitly specify bean name then, method name is ignored for bean creation.

Table 3.5. Feature Matrix

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Automatic `BeanPostProcessor` registration	No	Yes

Feature	BeanFactory	ApplicationContext
Automatic BeanFactoryPostProcessor registration	No	Yes
Convenient MessageSource access (for i18n)	No	Yes
ApplicationEvent publication	No	Yes

If you are using BeanFactory then a fair amount of support provided by Spring AOP and Spring transactions will not take effect (without requiring some extra steps on your part), although there would be nothing wrong with the configuration. BeanFactory is only recommended for Applet where memory consumption might be critical, a few extra kilobytes make the difference, otherwise it is always recommended to use ApplicationContext.

Check highlighted
NonUniqueBeanDefinitionFoundException when multiple beans of same type are found in App Context or multiple beans are made @Primary.
When @Primary and @Qualiifer are both used and @Primary is a different bean while @Qualifier points to a different bean, then @Primary will get priority.

```
@Autowired
	@Qualifier(value = "dateGen")
	PrefixGenerator prefixGenerator;

	@Bean("dateGen")
	//@Primary
	public DatePrefixGenerator datePrefixGenerator() {
		return new DatePrefixGenerator();
	}

	@Bean("intGen")
	@Primary
	public IntegerPrefixGenerator integerPrefixGenerator() {
		return new IntegerPrefixGenerator();
	}

System.out.println((context.getBean(PrefixGenerator.class)).generatePrefix());
Output: 1
```
--> @ org.springframework.context.annotation.Primary

@Target(value={METHOD, TYPE})
@Retention(value=RUNTIME)
@Documented
@Inherited
Indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency. If exactly one 'primary' bean exists among the candidates, it will be the autowired value.
This annotation is semantically equivalent to the <bean> element's primary attribute in Spring XML.
May be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean.
Example

```
@Component
public class FooService {

    private FooRepository fooRepository;

    @Autowired
    public FooService(FooRepository fooRepository) {
        this.fooRepository = fooRepository;
    }
}

@Component
public class JdbcFooRepository {

    public JdbcFooService(DataSource dataSource) {
        // ...
    }
}

@Primary
@Component
public class HibernateFooRepository {

    public HibernateFooService(SessionFactory sessionFactory) {
        // ...
    }
}
```

Because HibernateFooRepository is marked with @Primary, it will be injected preferentially over the jdbc-based variant assuming both are present as beans within the same Spring application context, which is often the case when component-scanning is applied liberally.
Note that using @Primary at the class level has no effect unless component-scanning is being used. If a @Primary-annotated class is declared via XML, @Primary annotation metadata is ignored, and <bean primary="true|false"/> is respected instead.
Since:
3.0

The JSR 250 or the common annotations for Java platform defines the @Resource annotation to autowire POJO references by name. The JSR-330 or the standard annotation for injection defines the @Inject annotation to autowire POJO references by type. The @Autowired annotation belongs to org.springframework.beans.factory.annotation package, specifically to the Spring framework. It can be only used within the context of Spring. Soon after Spring added @Autowired annotation, Java language itself standardized with various annotations to fulfill the purpose of @Autowired annotation. These annotations are @Resource which belongs to javax.annotation package and @Inject which belongs to javax.inject package.

@Resource annotation provides the same functionality as putting together @autowired and @Qualifier.

```
@Resource(name="integerPrefixGenerator")
```

```
      PrefixGenerator prefixGenerator;
Same as
@Autowired
@Qualifier(name=" integerPrefixGenerator")


@Resource(name="integerPrefixGenerator")
```
→
The JNDI name of the resource. For field annotations, the default is the field name. For method annotations, the default is the JavaBeans property name corresponding to the method. For class annotations, there is no default and this must be specified.

The first step to create autowiring by name using @Inject annotation is to create a custom annotation to identify both the POJO injection class and POJO injection point.
Notice the custom annotation makes use of the @Qualifier annotation. This annotation is different from the
one used with Spring's @Qualifier annotation, as this last class belongs to the same Java package as the @Inject
annotation (i.e., javax.inject)
Once the custom annotation is done, it's necessary to decorate the POJO injection class that generates the bean
instance, which in this case is the DatePrefixGenerator class.

```
package com.apress.springrecipes.sequence;
...
@DatePrefixAnnotation
public class DatePrefixGenerator implements PrefixGenerator {
...
}
```

Finally, the POJO attribute or injection point is decorated with the same custom annotation to qualify the POJO
and eliminate any ambiguity.

```
package com.apress.springrecipes.sequence;
import javax.inject.Inject;

public class SequenceGenerator {
@Inject @DataPrefixAnnotation
private PrefixGenerator prefixGenerator;
...
}
```

Java.lang.annotation.RetentionPolicy valid values
RetentionPolicy.SOURCE - Annotations are to be discarded by the compiler.
RetentionPolicy.CLASS - Annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time. This is the default behavior.
RententionPolicy.RUNTIME - Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively.

If you're going to do name based autowiring, the @Resource annotation offers the simplest syntax. For autowiring
by class type, all three annotations are as straightforward to use because all three require a single annotation.

Init-method or initMethod is called after the constructor runs. Destory-method or destroyMethod is called when context is closed.

Difference between constructor and setter injection

1. Partial initialization of beans is possible with setter injection because you don't need to pass values for all properties, they will get their default values. Partial initialization is not possible with constructor injection, you need to pass all constructor parameters.
2. Default constructor is used to create object in setter based injection, while parameterized constructor is used to create object with constructor based injection.
3. In setter based injection, order in which we pass the properties, does not matter, but it matters in cons..
4. If object of type A depends on object of Type B and you use constructor injection you will get ObjectCurrentlyInCreationException if the dependent object is being created. After all constructors are used to create objects. Spring can resolve circular dependencies through setter injection because objects are constructed before setter methods are invoked.
5. Setter injection will override values of constructor based injection, because object is created first.

Built in events:

1. ContextRefreshedEvents
2. ContextStartedEvents: when context is started with start() method.
3. ContextStoppedEvent: …stop() method
4. ContextClosedEvent
5. RequestHandledEvent: This is a web specific event telling all beans that an HTTP request has been serviced.

18) How do you turn on annotation based autowiring?

To enable `@Autowired`, you have to register `AutowiredAnnotationBeanPostProcessor`, and you can do it in two ways.

1. Include `<context:annotation-config >` in bean configuration file.

```
<beans>
    <context:annotation-config />
</beans>
```

2. Include `AutowiredAnnotationBeanPostProcessor` directly in bean configuration file.

```
<beans>
    <bean
class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostP
rocessor"/>
</beans>
```

19) Explain @Required annotation with example?

In a production-scale application, there may be hundreds or thousands of beans declared in the IoC container, and the dependencies between them are often very complicated. One of the shortcomings of setter injection is that it's very hard for you to check if all required properties have been set or not. To overcome this problem, you can set "**dependency-check**" attribute of <bean> and set one of four attributes i.e. none, simple, objects or all (none is default option).

In real life application, you will not be interested in checking all the bean properties configured in your context files. Rather you would like to check if particular set of properties have been set or not in some specific beans only. Spring's dependency checking feature using "**dependency-check**" attribute, will not able to help you in this case. So solve this problem, you can use `@Required` annotation.
To Use the `@Required` annotation over setter method of bean property in class file as below:

```
public class EmployeeFactoryBean extends AbstractFactoryBean<Object>
{
    private String designation;

    public String getDesignation() {
        return designation;
    }

    @Required
    public void setDesignation(String designation) {
        this.designation = designation;
    }

    //more code here
}
```

`RequiredAnnotationBeanPostProcessor` is a spring bean post processor that checks if all the bean properties with the `@Required` annotation have been set. To enable this bean post processor for property checking, you must register it in the Spring IoC container.

```
<bean
class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostPr
ocessor" />
```

If any properties with `@Required` have not been set, a `BeanInitializationException` will be thrown by this bean post processor.

@PostConstruct and @PreDestory are similar to init-method and destroy-method

Spring framework provides following **4 ways for controlling life cycle events** of bean:

- InitializingBean (`afterPropertiesSet()`) and DisposableBean (`destroy`) callback interfaces
- Other Aware interfaces for specific behavior
- Custom init() and destroy() methods in bean configuration file
- @PostConstruct and @PreDestroy annotations

```
post construct
afterPropertiesSet
Context closed.Mon Jul 31 18:00:09 IST 2017
pre destroy
Disposable bean destroy
```

ClassPathXmlApplicationContext
Standalone XML application context, taking the context definition files from the class path, interpreting plain paths as class path resource names that include the package path (e.g. "mypackage/myresource.txt"). Useful for test harnesses as well as for application contexts embedded within JARs.
The config location defaults can be overridden via `AbstractRefreshableConfigApplicationContext.getConfigLocations()`, Config locations can

either denote concrete files like "/myfiles/context.xml" or Ant-style patterns like "/myfiles/*-context.xml" (see the `AntPathMatcher` javadoc for pattern details).
Note: In case of multiple config locations, later bean definitions will override ones defined in earlier loaded files. This can be leveraged to deliberately override certain bean definitions via an extra XML file.
This is a simple, one-stop shop convenience ApplicationContext. Consider using the `GenericApplicationContext` class in combination with an `XmlBeanDefinitionReader` for more flexible context setup.

FileSystemXmlApplicationContext
Standalone XML application context, taking the context definition files from the file system or from URLs, interpreting plain paths as relative file system locations (e.g. "mydir/myfile.txt"). Useful for test harnesses as well as for standalone environments.
NOTE: Plain paths will always be interpreted as relative to the current VM working directory, even if they start with a slash. (This is consistent with the semantics in a Servlet container.) **Use an explicit "file:" prefix to enforce an absolute file path.**
This is a simple, one-stop shop convenience ApplicationContext. Consider using the `GenericApplicationContext` class in combination with an `XmlBeanDefinitionReader` for more flexible context setup.

XmlWebApplicationContext – WebApplicationContext implementation which takes its configurations from XML files, understood by XMLBeanDefinitionReader. By deafault it reads configuration from "WEB-INF/applicationContext.xml" for the root context and "/WEB-INF/test-servlet.xml" for a context with the namespace test-servlet.xml (like DispatcherServlet with servlet-name test)
The defaults can be overriden by contextConfigLocation context-param of ContextLoader and servlet init-param of FrameWorkServlet (DispatcherServlet extends FrameworkServlet)
Config Locations can either denote concrete files like "/WEB-INF/contex.xml" or ANT style patterns like /WEB-INF/*-context.xml

```
<context:component-scan base-package="com.apress.springrecipes.shop.config" />
<context:component-scan base-package="com.apress.springrecipes.shop">
<context:include-filter type="regex" expression="com\.apress\.
springrecipes\.shop\..*ShoppingCart.*" />
</context:component-scan>
```

Spring transaction propagation behaviors

1. REQUIRED – If a transaction is in progress then the current method must run within the same transaction. If a transaction does not exist then create a new transaction and current method must run in its own transaction. Default.
2. REQUIRES_NEW – If a transaction is in progress then suspend it. Create a new transaction and current method must run in it's own transaction.
3. SUPPORTS – If a transaction is in progress the current method must run in it. It is not necessary to that current method must run in a transaction.
4. NOT_SUPPORTED – The current method must not run within a transaction, if an existing transaction is in progress, it must be suspended.
5. MANDATORY – The current method must always run in a transaction, if there is not transaction in progress an exception will be thrown.
6. NEVER – The current method must never run within a transaction, if there is a transaction in progress then an exception will be thrown.

7. NESTED – If a transaction is in progress then the current method must run within a nested transaction of this transaction (supported by JDBC 3.0 save point feature). This feature is unique to Spring, whereas previous propagation types have analogues in Java EE specification. This is useful in batch processing, in which you have a long running process (imagine processing 1 million records) and you want to chunk the commits on the batch. So you commit every 10,000 records. If something goes wrong, you roll back the nested transaction and you've lost only 10,000 records' worth of work (as opposed to the entire 1 million).

Problems caused by concurrent transaction can be categorized into 4 types:

1. Dirty read - For 2 transactions T1 and T2, T1 reads field that is written by T2 but not yet committed, so later if T2 rollbacks, the field read by T2 will be temporary and invalid.
2. Non-repeatable read - For 2 transactions T1 and T2, T1 reads a field, then T2 updates the field. Later if T2 reads the field again, value will be different.
3. Phantom read - For 2 transactions T1 and T2, T1 reads some rows from a table, then T2 inserts new rows in the table, when T1 reads the rows again, there will be additional rows.
4. Lost updates - For 2 transactions T1 and T2, they both select a row for update, and based on the state of the row update it. Thus one overwrites the other when the second transaction should have waited for the first one to commit before performing its selection.

In theory, transactions should be completely isolated from each other (i.e., serializable) to avoid all the mentioned problems. However, this isolation level will have great impact on performance, because transactions have to run in serial order. In practice, transactions can run in lower isolation levels in order to improve performance. A transaction's isolation level can be specified by the isolation transaction attribute. Spring supports five isolation levels, as shown in Table 11-10. These levels are defined in the org.springframework.transaction.TransactionDefinition interface.

Isolation levels supported by Spring

1. DEFAULT – Use the default isolation level of the underlying database. For most databases the default level is READ_COMMITTED.
2. READ_UNCOMMITTED – Allows a transaction to read uncommitted changes made by other transactions. The dirty read, phantom read, non-repeatable read problems may occur.
3. READ_COMMITTED – Allows a transaction to read only those changes that have been committed by other transactions. The dirty read problem can be avoided but phantom read and non-repeatable read problem will still occur.
4. REPEATABLE_READ – Ensures that a transaction can read identical values from fields multiple times. For the duration of this transaction, updates made by other transactions to this field will be prohibited. The dirty read and non-repeatable read problems can be avoided but phantom read problem will still occur.
5. SERIALIZABLE - Ensures that a transaction can read identical rows from a table multiple times. For the duration of this transaction, inserts, updates, and deletes made by other transactions to this table are prohibited. All the concurrency problems can be avoided, but the performance will be low.

These isolation levels are supported by the underlying database and not by any application framework. Not all database engines support these isolation levels. You can change isolation level by calling setTransactionIsolation on java.sql.Connection interface.

20. Which DI would you suggest Constructor-based or setter-based DI?

You can use both Constructor-based and Setter-based Dependency Injection. The best solution is using constructor arguments for mandatory dependencies and setters for optional dependencies.

27. Explain Bean lifecycle in Spring framework

The spring container finds the bean's definition from the XML file and instantiates the bean.
Spring populates all of the properties as specified in the bean definition.
If the bean implements `BeanNameAware` interface, spring passes the bean's id to `setBeanName()` method.
If Bean implements `BeanFactoryAware` interface, spring passes the `beanfactory` to `setBeanFactory()` method.
If there are any bean `BeanPostProcessors` associated with the bean, Spring calls `postProcesserBeforeInitialization()` method.
Initialize the bean.
If the bean implements `IntializingBean`, its `afterPropertySet()` method is called. If the bean has init method declaration, the specified initialization method is called.
If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.
Now the bean is ready to use by the application.
If the bean implements `DisposableBean`, it will call the `destroy()` method.

A transaction is a unit of work that has ACID (atomic, consistent, isolated and durable) properties. Atomic means that the changes all happen or nothing happens. If money is debited from an account and credited to another account, a transaction ensures that either both the debit and credit complete or neither completes. Consistent implies that the changes leave the data in a consistent state. Isolated implies that changes do not interfere with other changes. Durable implies that once the changes are committed, they stay committed.

WebApplicationContext

The `WebApplicationContext` is an extension of the plain `ApplicationContext` that has some extra features necessary for web applications. It differs from a normal `ApplicationContext` in that it is capable of resolving themes, and that it knows which servlet it is associated with.

29. What are inner beans in Spring?

When a bean is only used as a property of another bean it can be declared as an inner bean. Spring's XML-based configuration metadata provides the use of `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements of a bean definition, in order to define the so-called inner bean. **Inner beans are always anonymous and they are always scoped as prototypes.**

28. Which are the important beans lifecycle methods? Can you override them?

There are two important bean lifecycle methods. The first one is `setup` which is called when the bean is loaded in to the container. The second method is the `teardown` method which is called when the bean is unloaded from the container.
The `bean` tag has two important attributes (`init-method` and `destroy-method`) with which you can define your own custom initialization and destroy methods. There are also the correspondive annotations(`@PostConstruct` and `@PreDestroy`).

30. How can you inject a Java Collection in Spring?

Spring offers the following types of collection configuration elements:
The <list> type is used for injecting a list of values, in the case that duplicates are allowed.

The <set> type is used for wiring a set of values but without any duplicates.
The <map> type is used to inject a collection of name-value pairs where name and value can be of any type.
The <props> type can be used to inject a collection of name-value pairs where the name and value are both Strings.

Q14. 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:
?
1@Configuration
2@Import({MainConfig.class, SchedulerConfig.class})
3public class AppConfig {

ApplicationContext context = new ClassPathXmlApplicationContext("spring-all.xml");
<import resource="main.xml"/>
<import resource="scheduler.xml"/>

Q17. 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
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
Q19. 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:
?
1@Autowired
2ServletContext servletContext;
3
4@Autowired
5ServletConfig servletConfig;

Q20. 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:

```
?
1public class Person {
2   private String name;
3
4   @Required
5   public void setName(String name) {
6       this.name = name;
7   }
8}
```

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

```
?
1<bean id="person" class="com.baeldung.Person">
2   <property name="name" value="Joe" />
3</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.

Q21. 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 this tutorial.

Q22. 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:

```
?
1<bean id="person1" class="com.baeldung.Person" >
2   <property name="name" value="Joe" />
3</bean>
4<bean id="person2" class="com.baeldung.Person" >
5   <property name="name" value="Doe" />
6</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:

```
?
1@Autowired
2@Qualifier("person1")
3private Person person;
```

Q23. How to handle exceptions in Spring MVC environment?

There are three ways to handle exceptions in Spring MVC:
Using @ExceptionHandler at controller level – this approach has a major feature – the @ExceptionHandler annotated method is only active for that particular controller, not globally for the entire application
Using HandlerExceptionResolver – this will resolve any exception thrown by the application

Using @ControllerAdvice – Spring 3.2 brings support for a global @ExceptionHandler with the @ControllerAdvice annotation, which enables a mechanism that breaks away from the older MVC model and makes use of *ResponseEntity* along with the type safety and flexibility of *@ExceptionHandler*

Q25. 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.
For details and various implementations, take a look at this series.

@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

Q29. 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.

Q37. What is Weaving?
According to the official docs, 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.

Q36. 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
JoinPoint: a point during the execution of a program, such as the execution of a method or the handling of an exception
Q34. 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.
You can, of course, go more in-depth on persistence, with the entire series discussing persistence in Spring.

Q33. 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

Q27. 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.
@RequestParam used for accessing the values of the query parameters whereas @PathVariable used for accessing the values from the URI template.

```
http://localhost:8080/springmvc/hello/101?param1=10&param2=20
101 is the PathVariable
```

Q5. How can we inject beans in Spring?
A few different options exist:
Setter Injection
Constructor Injection
Field Injection (@Autowired annotation is used above the field)
The configuration can be done using XML files or annotations.

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.**


ssb-business.xml

Spring has the provision for defining multiple application contexts. The org.springframework.context.ApplicationContext is the root context or the parent context for all other contexts.

From the official documentation:

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and is responsible for instatiating, configuring and assembling the aforementioned beans. The container get its instructions on how to instantiate, configure and assemble by reading configuration metadata. The configuration metadata is represented using XML, annotation or Java code.

In the Web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already defined in the root WebApplicationContext. These inherited beans can be overridden in the servlet specific scope, and you can define new scope specific beans local to a given servlet instance.

In the web.xml we can have org.springframework.web.context.ContextLoaderListener which is a spring context listener, which is invoked before the servlets are created.

```
<listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/spring/application-context.xml</param-value>
</context-param>
```

This listener looks for a context-param named contextConfigLocation in the web.xml. If it finds one, then the xml meta data file will be scanned for the beans that belong to the root application context. If it doesn't find any, then the listener will look for a file named applicationContext.xml in the class path for the configuration meta data.

Similarly the DispatcherServlet, which is configured in the web.xml will look for an init-param named contextConfigLocation in the servlet definition as shown below.

DispatcherServlet configuration:

```
<servlet>
   <servlet-name>dispatcher</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>WEB-INF/spring/servlet-context.xml</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
```

If the init-param is not configured, then a file named dispatcher-servlet.xml will be searched for in the class path. Tha file name being searched for is made up of the servlet name – in our case 'dispatcher' – and a string '-servlet.xml' as suffix.

Note:

The child context i.e. the MVC context will have access to the beans in the parent or root context. But not vice versa.

So, in both the cases i.e. the ContextLoaderListener or in case of DispatcherServlet if the parameters are not found in the web.xml and the corresponding files applicationContext.xml or YOUR SERVLET NAME-servlet.xml are not found in the class path then a **FileNotFoundException** will be thrown and the application won't start up.

Usually the model classes and other beans that belong to the entire application are configured as beans in the applicationContext.xml. And the classes annotated with @Controller and the beans that are related to the MVC layer are configured in the DispatcherServlet configuration file.

Most of the tutorials on web have Spring MVC configuration with a single file included in both the parent context as well as in the child context because of lack of understanding of how Spring works. Due to this all the beans configured in the corresponding xml file will be instantiated twice including any of those beans defined for connection pooling, in which case connection pooling will also be a problem.

If we have only one file with spring beans configuration meta data, then we can configure only that xml file in DispatcherServlet's web context, which is the only mandatory context for an MVC web application, in which case we can exclude the configuration for the servlet context listener ContextLoaderListener in the web.xml.

DispatcherServlet extends FrameworkServlet.
Class FrameworkServlet
All Implemented Interfaces:
Serializable, Servlet, ServletConfig, Aware, ApplicationContextAware, EnvironmentAware, EnvironmentCapable
Direct Known Subclasses:
DispatcherServlet
Base servlet for Spring's web framework. Provides integration with a Spring application context, in a JavaBean-based overall solution.
This class offers the following functionality:

1. Manages a `WebApplicationContext` instance per servlet. The servlet's configuration is determined by beans in the servlet's namespace.
2. Publishes events on request processing, whether or not a request is successfully handled.

Subclasses must implement `doService(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` to handle requests. Because this extends `HttpServletBean` rather than HttpServlet directly, bean properties are automatically mapped onto it. Subclasses can override `initFrameworkServlet()` for custom initialization.
Detects a "contextClass" parameter at the servlet init-param level, falling back to the default context class, `XmlWebApplicationContext`, if not found. Note that, with the default `FrameworkServlet`, a custom context class needs to implement the `ConfigurableWebApplicationContext` SPI.

Accepts an optional "contextInitializerClasses" servlet init-param that specifies one or more `ApplicationContextInitializer` classes. The managed web application context will be delegated to these initializers, allowing for additional programmatic configuration, e.g. adding property sources or activating profiles against the context's environment. See also `ContextLoader` which supports a "contextInitializerClasses" context-param with identical semantics for the "root" web application context.
Passes a "contextConfigLocation" servlet init-param to the context instance, parsing it into potentially multiple file paths which can be separated by any number of commas and spaces, like "test-servlet.xml, myServlet.xml". If not explicitly specified, the context implementation is supposed to build a default location from the namespace of the servlet.
Note: In case of multiple config locations, later bean definitions will override ones defined in earlier loaded files, at least when using Spring's default ApplicationContext implementation. This can be leveraged to deliberately override certain bean definitions via an extra XML file.
The default namespace is "'servlet-name'-servlet", e.g. "test-servlet" for a servlet-name "test" (leading to a "/WEB-INF/test-servlet.xml" default location with XmlWebApplicationContext). The namespace can also be set explicitly via the "namespace" servlet init-param.
As of Spring 3.1, `FrameworkServlet` may now be injected with a web application context, rather than creating its own internally. This is useful in Servlet 3.0+ environments, which support programmatic registration of servlet instances. See `FrameworkServlet(WebApplicationContext)` Javadoc for details.

`public class DispatcherServlet`
`extends FrameworkServlet`
Central dispatcher for HTTP request handlers/controllers, e.g. for web UI controllers or HTTP-based remote service exporters. Dispatches to registered handlers for processing a web request, providing convenient mapping and exception handling facilities.

All Implemented Interfaces:
Serializable, Servlet, ServletConfig, Aware, ApplicationContextAware, EnvironmentAware, EnvironmentCapable

This servlet is very flexible: It can be used with just about any workflow, with the installation of the appropriate adapter classes. It offers the following functionality that distinguishes it from other request-driven web MVC frameworks:

- It is based around a JavaBeans configuration mechanism.
- It can use any `HandlerMapping` implementation - pre-built or provided as part of an application - to control the routing of requests to handler objects. Default is `BeanNameUrlHandlerMapping` and `DefaultAnnotationHandlerMapping`. HandlerMapping objects can be defined as beans in the servlet's application context, implementing the HandlerMapping interface, overriding the default HandlerMapping if present. HandlerMappings can be given any bean name (they are tested by type).
- It can use any `HandlerAdapter`; this allows for using any handler interface. Default adapters are `HttpRequestHandlerAdapter`, `SimpleControllerHandlerAdapter`, for Spring's `HttpRequestHandler` and `Controller` interfaces, respectively. A default `AnnotationMethodHandlerAdapter` will be registered as well. HandlerAdapter objects can

be added as beans in the application context, overriding the default HandlerAdapters. Like HandlerMappings, HandlerAdapters can be given any bean name (they are tested by type).

- The dispatcher's exception resolution strategy can be specified via a `HandlerExceptionResolver`, for example mapping certain exceptions to error pages. Default are `AnnotationMethodHandlerExceptionResolver`, `ResponseStatusExceptionResolver`, and `DefaultHandlerExceptionResolver`. These HandlerExceptionResolvers can be overridden through the application context. HandlerExceptionResolver can be given any bean name (they are tested by type).
- Its view resolution strategy can be specified via a `ViewResolver` implementation, resolving symbolic view names into View objects. Default is `InternalResourceViewResolver`. ViewResolver objects can be added as beans in the application context, overriding the default ViewResolver. ViewResolvers can be given any bean name (they are tested by type).
- If a `View` or view name is not supplied by the user, then the configured `RequestToViewNameTranslator` will translate the current request into a view name. The corresponding bean name is "viewNameTranslator"; the default is `DefaultRequestToViewNameTranslator`.
- The dispatcher's strategy for resolving multipart requests is determined by a `MultipartResolver` implementation. Implementations for Apache Commons FileUpload and Servlet 3 are included; the typical choice is `CommonsMultipartResolver`. The MultipartResolver bean name is "multipartResolver"; default is none.
- Its locale resolution strategy is determined by a `LocaleResolver`. Out-of-the-box implementations work via HTTP accept header, cookie, or session. The LocaleResolver bean name is "localeResolver"; default is `AcceptHeaderLocaleResolver`.
- Its theme resolution strategy is determined by a `ThemeResolver`. Implementations for a fixed theme and for cookie and session storage are included. The ThemeResolver bean name is "themeResolver"; default is `FixedThemeResolver`.

NOTE: The `@RequestMapping` annotation will only be processed if a corresponding `HandlerMapping` (for type-level annotations) and/or `HandlerAdapter` (for method-level annotations) is present in the dispatcher. This is the case by default. However, if you are defining custom `HandlerMappings` or `HandlerAdapters`, then you need to make sure that a corresponding custom `DefaultAnnotationHandlerMapping` and/or `AnnotationMethodHandlerAdapter` is defined as well - provided that you intend to use `@RequestMapping`.

AnnotationList

@ResponseBody

`@Target(value={TYPE,METHOD}) @Retention(value=RUNTIME) @Documented`
Indicates that the method response should be bound to the web response body. Supported for annotation handler methods in servlet environments.
As of 4.0 it can be added to the type, in which case it does not need to be added to the method and can be inherited.
When building a JSON endpoint it is an amazing way to convert objects to JSON for easier consumption.

@RESTController

@Target(value=TYPE) @Retention(value=RUNTIME) @Documented @Controller @ResponseBody
public @interface RestController
Convenience annotation that is itself annotated with @ResponseBody and @Controller. Types annotated with @RESTController are treated as controllers whose @RequestMapping annotation assume @ResponseBody semantics.
Note: @RESTController is processed if an appropriate HandlerMapping HandlerAdapter pair is found such as the RequestMappingHandlerMapping - RequestMappingHandlerAdapter which are the default in MVC Java config and MVC namespace. @RESTController is not supported with DefaultAnnotationHandlerMapping and AnnotationMethodHandlerAdapter.