***Singleton Pattern -***

*In Spring, singleton objects are created by default. This means that only one instance of a particular bean is created per Spring context. This is achieved using a singleton bean scope.*

***When you define a bean in Spring with a singleton scope, Spring will create only one instance of the bean and cache it. Any subsequent requests for the bean will return the cached instance.***

*It’s worth noting that while singleton beans are created only once per Spring context,* ***they may still have state that can be modified by multiple threads if the bean is not designed to be thread-safe****. Therefore, it’s important to ensure that singleton beans are thread-safe, either through proper synchronization or by avoiding mutable state altogether.*

***Factory Pattern*** *–*

*In Spring, the Factory Pattern is used to create instances of beans dynamically, based on the current state of the application. Spring provides two main implementations of the Factory pattern: BeanFactory and ApplicationContext.*

*The BeanFactory is the core interface for accessing the Spring container, and it is responsible for creating and managing the bean objects.*

*The ApplicationContext interface extends the functionality of the BeanFactory by providing additional features, such as support for* ***internationalization, resource loading, and event propagation.*** *The ApplicationContext interface is typically used in web applications, where it provides additional features, such as support for web scopes and the ability to obtain objects from the ServletContext.*

***Template Method Pattern***

*In Spring framework, the Template Method pattern is used extensively in the implementation of various template classes. These template classes provide a skeletal implementation of an algorithm or process and allow the user to override specific parts of the algorithm or process as needed.*

*Examples of the Template Method pattern in Spring is the JdbcTemplate class, which provides a set of convenience methods for working with relational databases. The JdbcTemplate class provides a template method named execute(), which accepts a callback object that defines the SQL statement to execute, and the parameters to pass to the statement. The JdbcTemplate class takes care of all the low-level details of opening and closing database connections, executing the statement, and handling any exceptions that may occur.*

*Another example of the Template Method pattern in Spring is the AbstractController class, which provides a skeletal implementation of a controller class for handling HTTP requests.*

*The AbstractController class defines a template method named handleRequestInternal(), which is called by the Spring framework to handle HTTP requests. The handleRequestInternal() method is responsible for processing the request, and returning a ModelAndView object.*

***Proxy Pattern***

*In Spring framework, the Proxy pattern is used extensively in the implementation of various AOP (Aspect Oriented Programming) features. it allows developers to encapsulate certain aspects of application behavior and apply them to multiple classes or modules in a modular and reusable way.*

*Examples of the Proxy pattern in Spring is the use of dynamic proxies to provide declarative transaction management. The @Transactional annotation tells Spring to apply transaction management to the method. When this method is invoked, Spring creates a dynamic proxy object that intercepts the method call and performs the necessary transaction management operations, such as beginning a transaction, committing the transaction, or rolling back the transaction if an exception occurs.*

*Another example of the Proxy pattern in Spring is the use of JDK dynamic proxies and CGLIB proxies to implement the dependency injection (DI) mechanism. When a bean is configured for DI using an interface, Spring creates a JDK dynamic proxy object that implements the interface and delegates method calls to the actual bean object.*

***Decorator Pattern***

*The Decorator pattern is a design pattern that allows behaviour to be added to an individual object, either statically or dynamically, without affecting the behaviour of other objects from the same class. It is used to extend or modify the behaviour of objects at runtime without affecting their original structure.*

*One of the most common examples of the Decorator pattern in Spring is the use of the HandlerInterceptor interface to provide additional behaviour to HTTP request processing in a Spring MVC application. Here is an example:*

public class LoggingInterceptor implements HandlerInterceptor {  
 @Override  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
 // Logging logic here  
 return true;  
 }  
   
 @Override  
 public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  
 // Logging logic here  
 }  
   
 @Override  
 public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  
 // Logging logic here  
 }  
}

*In the above example, the LoggingInterceptor class implements the HandlerInterceptor interface, which provides three methods that can be used to intercept HTTP requests before and after they are processed by a controller. This allows developers to add additional behavior to the request processing pipeline without changing the original behavior of the controller.*

*Another example of the Decorator pattern in Spring is the use of the DelegatingFilterProxy class to add additional behavior to web application filters.*

***Observer Pattern***

*The Observer pattern is a design pattern that allows one object to observe changes in another object and react to those changes. It is used to maintain consistency between related objects and to reduce coupling between them.*

*One of the most common examples of the Observer pattern in Spring is the use of the ApplicationEventPublisher interface to publish events to registered listeners. Another example of the Observer pattern in Spring is the use of the @EventListener annotation to handle events.*

***Command Pattern***

*The Command pattern is a behavioral design pattern that separates the request for a particular action from the object that performs the action. The pattern allows requests to be stored as objects, which can be passed as parameters to other objects or stored for later use.*

*One of the most common examples of the Command pattern in Spring is the use of the JdbcTemplate class to execute database queries. The update method is an example of a Command object that encapsulates a database query and its parameters as an object. The JdbcTemplate class is an example of an Invoker object that executes the command object.*

*Another example of the Command pattern in Spring is the use of the @RequestMapping annotation to map HTTP requests to methods.*

***Chain of Responsibility Pattern***

*The Chain of Responsibility is a behavioral design pattern that allows a set of objects to handle requests or events in a sequential manner. In this pattern, each object in the chain can handle the request or pass it on to the next object in the chain.*

*One of the most common examples of the Chain of Responsibility pattern in Spring is the use of Interceptors to handle HTTP requests. Another example of the Chain of Responsibility pattern in Spring is the use of AOP (Aspect-Oriented Programming) to add behavior to Spring beans.*

***Flyweight Pattern***

*The Flyweight is a structural design pattern that allows sharing objects that have common state across multiple contexts, thus reducing the overall memory footprint of an application. In Spring, the Flyweight pattern is used in several places to optimize memory usage and improve performance.*

*One of the most common examples of the Flyweight pattern in Spring is the use of the Singleton scope for beans. Another example of the Flyweight pattern in Spring is the use of the Object Pool design pattern for managing objects that are expensive to create.*

***Interpreter Pattern*** *–*

*The Interpreter Pattern is a design pattern that defines a grammar for a language and provides an interpreter to execute the grammar. In the Interpreter Pattern, the grammar is defined using a set of rules or expressions, and the interpreter evaluates the grammar by interpreting these expressions.*

*In Spring Framework, the Interpreter Pattern is used to implement the Spring Expression Language (SpEL). SpEL is a powerful expression language that can be used to configure and manipulate beans in Spring applications.*

***Builder Pattern***

*Spring's BeanFactory and ApplicationContext use the builder pattern to create complex objects with different configurations.*

***Service Locator Pattern***

*ServiceLocatorFactoryBean stores the information of all the beans in the context. When the client code asks for a service (bean) using a name, it simply locates that bean in the context and returns it. Client code does not need to write spring-related code to locate a bean.*