Spring Framework is a popular Java framework for building enterprise-level applications. It provides a wide range of features and functionalities that enable to develop high-performance, scalable, and maintainable applications. One of the key strengths of Spring is its support for *design patterns*. In this article, we will explore some of the most commonly used design patterns in Spring Framework and see how they are used in practice.

**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 through the use of 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.*

To define a singleton bean in Spring, you can use the *@Component* annotation or one of its stereotype annotations (*@Service*, *@Repository*, etc.) on a class. For example, here is how you can define a singleton bean using the *@Component* annotation:

@Component  
public class MySingletonBean {  
 // implementation here  
}

When the Spring container is initialized, it will create an instance of MySingletonBean and cache it. Any subsequent requests for MySingletonBean 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**

The Factory Pattern is another commonly used design pattern in Spring. It provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. *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 BeanFactory uses a number of different strategies to create and manage the beans, including the Singleton and Prototype design patterns.

Let’s consider an example where we have a UserService interface and a UserServiceImpl class that implements this interface. We want to create a UserService object and use it in our application. We can define the UserServiceImpl as a bean in our Spring configuration file as follows:

<bean id="userService" class="com.example.UserServiceImpl"/>  
  
OR  
  
@Configuration  
public class AppConfig {  
 @Bean  
 public UserService userService() {  
 return new UserServiceImpl();  
 }  
}

Now, when we want to use the UserService object in our application, we can obtain it from the Spring container using the BeanFactory. Here is an example of how we can do that:

public class UserController {  
 private BeanFactory beanFactory;  
  
 public void setBeanFactory(BeanFactory beanFactory) {  
 this.beanFactory = beanFactory;  
 }  
  
 public void doSomething() {  
 UserService userService = (UserService) beanFactory.getBean("userService");  
 // use userService object here  
 }  
}

In the above example, we have a UserController class that has a BeanFactory instance injected via its setter method. The doSomething() method of the UserController class uses the BeanFactory to obtain a UserService object from the Spring container and then uses it in the application.

Note that when we call the getBean() method of the BeanFactory to obtain the UserService object, Spring uses the Factory pattern to create the UserService object. Spring creates a new instance of the UserServiceImpl class and initializes it with any dependencies it may have.

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.

Here is an example of how we can use the *ApplicationContext* to create and manage bean objects in our Spring application:

public class UserServlet extends HttpServlet {  
 private ApplicationContext applicationContext;  
  
 public void init() throws ServletException {  
 applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");  
 }  
  
 public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 UserService userService = (UserService) applicationContext.getBean("userService");  
 // use userService object here  
 }  
}

In the above example, we have a UserServlet class that initializes the ApplicationContext in its init() method. The doGet() method of the UserServlet class uses the ApplicationContext to obtain a UserService object from the Spring container and then uses it in the application.

**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.

One of the most well-known 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. Here is an example of how we can use the JdbcTemplate class to execute a simple SQL query:

public class UserDaoImpl implements UserDao {  
 private JdbcTemplate jdbcTemplate;  
  
 public void setDataSource(DataSource dataSource) {  
 jdbcTemplate = new JdbcTemplate(dataSource);  
 }  
  
 public User getUserById(int userId) {  
 return jdbcTemplate.queryForObject("SELECT \* FROM users WHERE id = ?", new Object[]{userId}, new UserRowMapper());  
 }  
}

In the above example, we have a UserDaoImpl class that uses the JdbcTemplate to execute a SQL query to retrieve a user by their ID. The JdbcTemplate’s execute() method is called internally by the queryForObject() method, which passes a callback object (an instance of the UserRowMapper class) to the execute() method. The UserRowMapper class is responsible for mapping the results of the SQL query to a User object.

Note that the execute() method of the JdbcTemplate class uses the Template Method pattern to provide a skeletal implementation of the database access process, while allowing the user to customize the behavior of the method by providing a callback object.

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 that contains the view to render and the model data to use when rendering the view. Here is an example of how we can use the AbstractController class to handle an HTTP request:

public class UserController extends AbstractController {  
 private UserService userService;  
  
 public void setUserService(UserService userService) {  
 this.userService = userService;  
 }  
  
 protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) throws Exception {  
 int userId = Integer.parseInt(request.getParameter("userId"));  
 User user = userService.getUserById(userId);  
  
 ModelAndView modelAndView = new ModelAndView("userView");  
 modelAndView.addObject("user", user);  
  
 return modelAndView;  
 }  
}

In the above example, we have a UserController class that extends the AbstractController class to handle HTTP requests. The handleRequestInternal() method of the AbstractController class is overridden to provide a customized implementation that retrieves a user by their ID using a UserService object, and then returns a ModelAndView object that contains the user data to render in the view.

**Proxy Pattern**

In Spring framework, the Proxy pattern is used extensively in the implementation of various AOP (Aspect Oriented Programming) features. AOP is a programming paradigm that allows the separation of cross-cutting concerns from the core business logic of an application. In other words, it allows developers to encapsulate certain aspects of application behavior and apply them to multiple classes or modules in a modular and reusable way.

One of the most common examples of the Proxy pattern in Spring is the use of dynamic proxies to provide declarative transaction management for methods in a service layer. Here is an example of how we can use Spring’s transaction management using dynamic proxies:

@Transactional  
public void updateEmployee(Employee employee) {  
 employeeDao.update(employee);  
}

In the above example, the *@Transactional* annotation tells Spring to apply transaction management to the updateEmployee() 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. Here is an example:

public interface EmployeeService {  
 void updateEmployee(Employee employee);  
}  
  
@Service  
public class EmployeeServiceImpl implements EmployeeService {  
 @Override  
 public void updateEmployee(Employee employee) {  
 employeeDao.update(employee);  
 }  
}  
  
@Controller  
public class EmployeeController {  
 @Autowired  
 private EmployeeService employeeService;  
  
 @PostMapping("/employee")  
 public void updateEmployee(@RequestBody Employee employee) {  
 employeeService.updateEmployee(employee);  
 }  
}

In the above example, the EmployeeController class has a dependency on the EmployeeService interface, which is implemented by the EmployeeServiceImpl class. When the controller is created, Spring creates a dynamic proxy object that implements the EmployeeService interface and delegates method calls to the actual EmployeeServiceImpl object. This allows Spring to inject the actual implementation of the service at runtime without the need for the controller to know about the implementation.

When a bean is configured for DI using a class, Spring creates a CGLIB proxy object that extends the class and overrides the necessary methods to perform the DI operations. Here is an example:

@Service  
public class EmployeeService {  
 @Autowired  
 private EmployeeDao employeeDao;  
  
 public void updateEmployee(Employee employee) {  
 employeeDao.update(employee);  
 }  
}

In the above example, the EmployeeService class has a dependency on the EmployeeDao class, which is injected using the @Autowired annotation. When the service is created, Spring creates a CGLIB proxy object that extends the EmployeeService class and overrides the updateEmployee() method to perform the DI operations.

**Decorator Pattern**

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

In Spring, the Decorator pattern is used in several places to provide additional functionality without changing the original behavior of the objects. One of the most common examples of the Decorator pattern in Spring is the use of the *HandlerInterceptor*interface to provide additional behavior 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. Here is an example:

public class LoggingFilter implements Filter {  
 @Override  
 public void init(FilterConfig filterConfig) throws ServletException {  
 // Initialization logic here  
 }  
   
 @Override  
 public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  
 // Logging logic here  
 chain.doFilter(request, response);  
 }  
   
 @Override  
 public void destroy() {  
 // Cleanup logic here  
 }  
}

In the above example, the LoggingFilter class implements the Filter interface, which is used to intercept HTTP requests and responses in a web application. The filter adds additional behavior to the request processing pipeline by logging the request and response data.

To use the filter in a Spring application, we can configure the DelegatingFilterProxy class to delegate requests to the LoggingFilter class, as shown below:

<filter>  
 <filter-name>loggingFilter</filter-name>  
 <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>loggingFilter</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

In this configuration, the DelegatingFilterProxy class acts as a Decorator for the LoggingFilter class, intercepting all requests and delegating them to the filter for processing.

**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.

In Spring, the Observer pattern is used in several places to allow objects to observe changes in other objects and react accordingly. 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. Here is an example:

public class Order {  
 private List<OrderItem> items;  
 private double totalPrice;  
   
 // getters and setters  
   
 public void addItem(OrderItem item) {  
 items.add(item);  
 totalPrice += item.getPrice();  
 ApplicationEventPublisher publisher = // get ApplicationEventPublisher instance  
 publisher.publishEvent(new OrderItemAddedEvent(this, item));  
 }  
}  
  
public class OrderItemAddedEvent extends ApplicationEvent {  
 private Order order;  
 private OrderItem item;  
   
 public OrderItemAddedEvent(Order order, OrderItem item) {  
 super(order);  
 this.order = order;  
 this.item = item;  
 }  
   
 // getters and setters  
}  
  
public class OrderProcessor implements ApplicationListener<OrderItemAddedEvent> {  
 @Override  
 public void onApplicationEvent(OrderItemAddedEvent event) {  
 // Process the order item added event here  
 }  
}

In the above example, the Order class contains a list of OrderItem objects and a totalPrice field. When a new OrderItem is added to the order, the addItem method is called and the total price is updated. The method then publishes an OrderItemAddedEvent to the ApplicationEventPublisher instance, which notifies all registered listeners of the event.

The OrderProcessor class implements the ApplicationListener interface and is registered as a listener for OrderItemAddedEvent events. When an event is published, the onApplicationEvent method is called, which processes the event as required.

Another example of the Observer pattern in Spring is the use of the *@EventListener* annotation to handle events. Here is an example:

@Component  
public class OrderProcessor {  
 @EventListener  
 public void handleOrderItemAddedEvent(OrderItemAddedEvent event) {  
 // Process the order item added event here  
 }  
}

In this example, the OrderProcessor class is annotated with the @Component annotation, which makes it a Spring bean. The class contains a method annotated with the @EventListener annotation, which specifies that the method should handle OrderItemAddedEvent events.

When an event is published, Spring automatically detects the annotated method and calls it to handle the event.

**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.

In Spring, the Command pattern is used in several places to allow objects to execute commands and undo or redo them if necessary. One of the most common examples of the Command pattern in Spring is the use of the *JdbcTemplate*class to execute database queries. Here is an example:

@Autowired  
private JdbcTemplate jdbcTemplate;  
  
public void updateOrder(Order order) {  
 jdbcTemplate.update("UPDATE orders SET status = ? WHERE id = ?", order.getStatus(), order.getId());  
}

In this example, the JdbcTemplate class is used to execute a database query to update an Order object in the database. The update method of the JdbcTemplate class takes a SQL query string and an array of parameters to be used in the query.

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 in Spring controllers. Here is an example:

@Controller  
@RequestMapping("/orders")  
public class OrderController {  
 @Autowired  
 private OrderService orderService;  
  
 @RequestMapping(value = "/{id}", method = RequestMethod.GET)  
 public String getOrder(@PathVariable("id") Long id, Model model) {  
 Order order = orderService.getOrderById(id);  
 model.addAttribute("order", order);  
 return "orderDetails";  
 }  
}

In this example, the OrderController class is annotated with the @Controller and @RequestMapping annotations, which specify that the class is a Spring controller and that all HTTP requests to “/orders” should be handled by methods in this controller.

The getOrder method is annotated with the @RequestMapping annotation, which specifies that HTTP GET requests to “/orders/{id}” should be handled by this method. The method takes a Long parameter, which represents the ID of the order to be retrieved from the database.

The getOrder method is an example of a Command object that encapsulates the logic for retrieving an order from the database. The method is called by the Spring MVC framework when a GET request is received for “/orders/{id}”.

**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 has the opportunity to handle the request or pass it on to the next object in the chain.

In Spring, the Chain of Responsibility pattern is used in several places to allow multiple objects to handle requests in a flexible and extensible way. One of the most common examples of the Chain of Responsibility pattern in Spring is the use of *Interceptors*to handle HTTP requests. Here is an example:

@Configuration  
public class AppConfig extends WebMvcConfigurerAdapter {  
  
 @Override  
 public void addInterceptors(InterceptorRegistry registry) {  
 registry.addInterceptor(new LoggingInterceptor())  
 .addPathPatterns("/api/\*\*")  
 .excludePathPatterns("/api/public/\*\*");  
 registry.addInterceptor(new AuthorizationInterceptor())  
 .addPathPatterns("/api/\*\*")  
 .excludePathPatterns("/api/public/\*\*");  
 }  
  
 // Other configuration methods...  
}

In this example, the AppConfig class extends the *WebMvcConfigurerAdapter*class and overrides the addInterceptors method to register two interceptors: a *LoggingInterceptor*and an *AuthorizationInterceptor*.

The addInterceptor method is an example of a Chain object that adds an Interceptor object to the chain. The Interceptor object is an example of a Handler object that handles the HTTP request and optionally passes it on to the next object in the chain.

The Interceptor objects can be configured with a set of path patterns that specify which requests they should handle. In this example, both interceptors are configured to handle all requests to “/api/” except those that match “/api/public/”.

Another example of the Chain of Responsibility pattern in Spring is the use of *AOP (Aspect-Oriented Programming)* to add behavior to Spring beans. Here is an example:

@Aspect  
@Component  
public class LoggingAspect {  
  
 @Before("execution(\* com.example.service.\*.\*(..))")  
 public void logBefore(JoinPoint joinPoint) {  
 System.out.println("Before calling method: " + joinPoint.getSignature().getName());  
 }  
  
 // Other advice methods...  
}

In this example, the LoggingAspect class is annotated with the @Aspect and @Component annotations to indicate that it is an aspect component that provides cross-cutting behavior to Spring beans.

The aspect provides advice methods that are executed before, after, or around methods in Spring beans. The advice methods are an example of Handler objects that handle the request to execute a method in a Spring bean.

The execution pointcut expression in the @Before annotation specifies that the advice method should be executed before any method in the com.example.service package is called.

**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. Here is an example:

@Service  
@Scope("singleton")  
public class MyService {  
 // ...  
}

In this example, the MyService class is annotated with the @Service annotation, which tells Spring that this class is a Spring bean. Additionally, the @Scope annotation is used to specify that this bean should be created as a Singleton, meaning that only one instance of the bean will be created and shared across all parts of the application that need it.

By using the Singleton scope, Spring is able to avoid creating multiple instances of the MyService class, which can save a significant amount of memory in large applications.

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. Here is an example:

@Component  
public class MyObjectPool {  
   
 private List<MyObject> objects;  
   
 public MyObjectPool(int size) {  
 objects = new ArrayList<MyObject>(size);  
 for (int i = 0; i < size; i++) {  
 objects.add(new MyObject());  
 }  
 }  
   
 public MyObject borrowObject() {  
 if (objects.isEmpty()) {  
 throw new RuntimeException("No more objects in pool");  
 }  
 return objects.remove(0);  
 }  
   
 public void returnObject(MyObject object) {  
 objects.add(object);  
 }  
}

In this example, the MyObjectPool class represents an object pool for instances of the MyObject class. The pool is created with a fixed number of instances, and each instance is expensive to create. By using an object pool, Spring can avoid the overhead of creating and destroying MyObject instances each time they are needed.

The borrowObject and returnObject methods are examples of the Flyweight pattern in action. When an object is borrowed from the pool, it is returned to the caller with any necessary state changes. When the object is no longer needed, it is returned to the pool so that it can be reused by another caller.

**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. SpEL expressions are evaluated at runtime and can be used to inject values into bean properties, invoke methods, and perform operations on data. Here’s an example of how SpEL can be used to inject values into bean properties:

@Component  
public class MyService {  
  
 @Value("#{ systemProperties['myProperty'] }")  
 private String myProperty;  
  
 // ...  
  
}

In this example, the @Value annotation is used to inject the value of the myProperty system property into the myProperty field of the MyService class. The SpEL expression #{ systemProperties['myProperty'] } is evaluated at runtime and retrieves the value of the myProperty system property.

SpEL can also be used to perform more complex operations, such as invoking methods and performing operations on data. Here’s an example:

@Component  
public class MyService {  
  
 @Value("#{ T(java.lang.Math).random() \* 100.0 }")  
 private double randomNumber;  
  
 // ...  
  
}

In this example, the @Value annotation is used to inject a random number into the randomNumber field of the MyService class. The SpEL expression #{ T(java.lang.Math).random() \* 100.0 } invokes the java.lang.Math.random() method to generate a random number between 0 and 1, and then multiplies it by 100 to get a random number between 0 and 100.

SpEL can also be used to access and manipulate complex data structures, such as maps and lists. Here’s an example:

@Component  
public class MyService {  
  
 @Value("#{ myMap['myKey'] }")  
 private String myValue;  
  
 // ...  
  
}

In this example, the @Value annotation is used to inject the value of the myKey key in the myMap map into the myValue field of the MyService class. The SpEL expression #{ myMap[‘myKey’] } retrieves the value of the myKey key in the myMap map.

In summary, Spring Framework provides powerful support for a wide range of design patterns, including Singleton, Factory, Template Method, Proxy, Decorator, and Observer, etc. By understanding and applying these patterns effectively, developers can build high-quality, scalable, and maintainable applications.