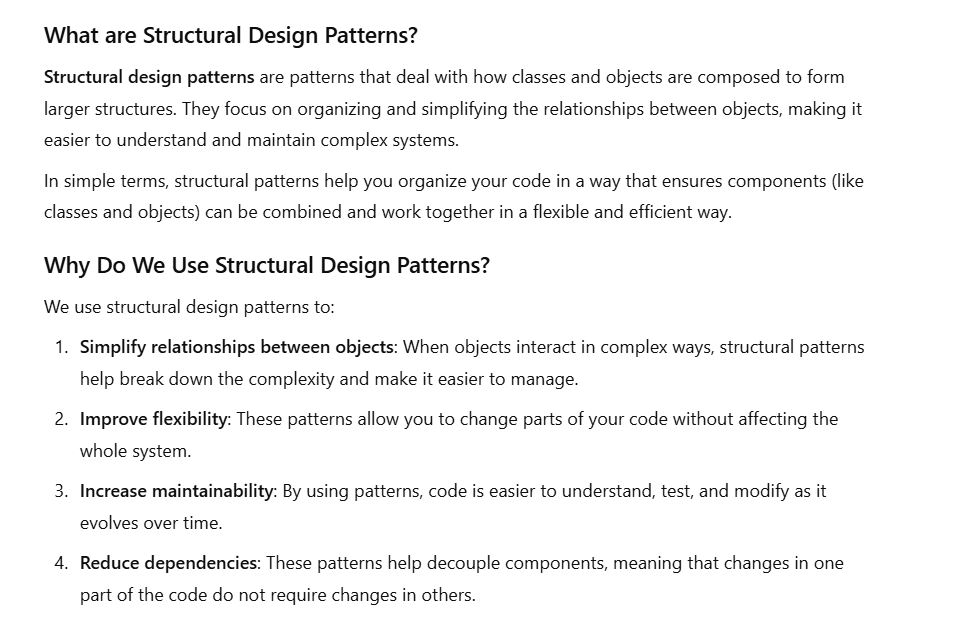
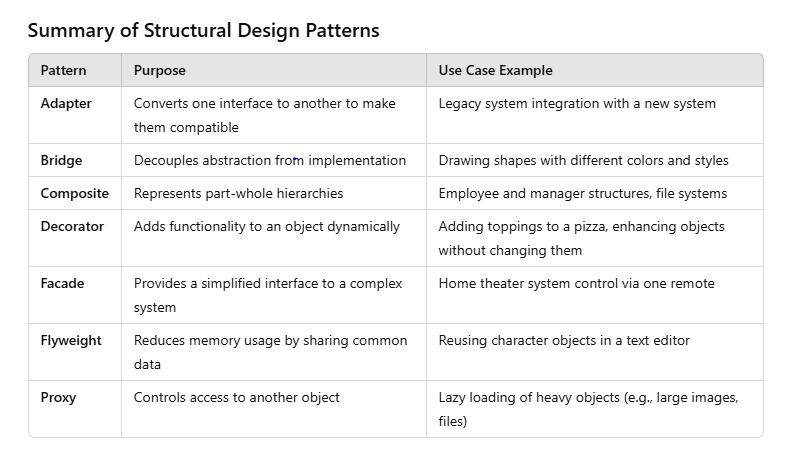
Structural design pattern

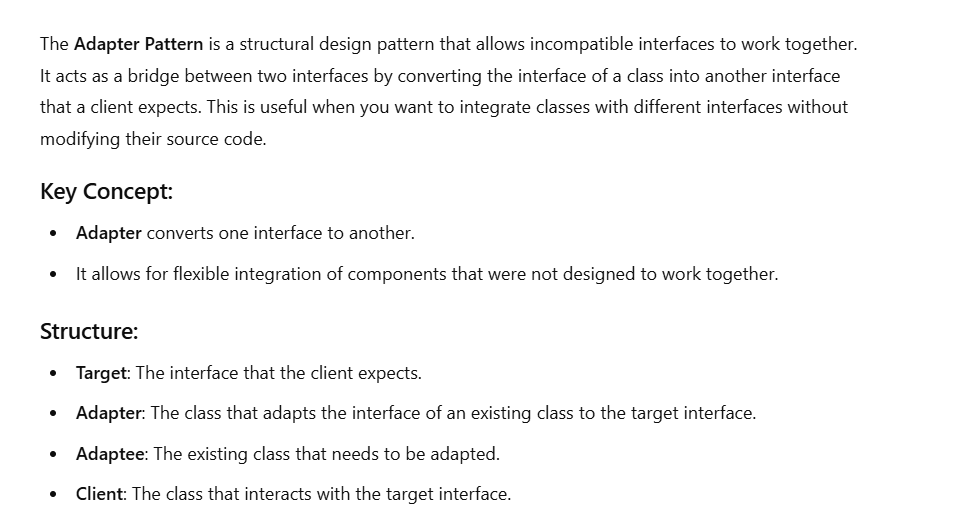


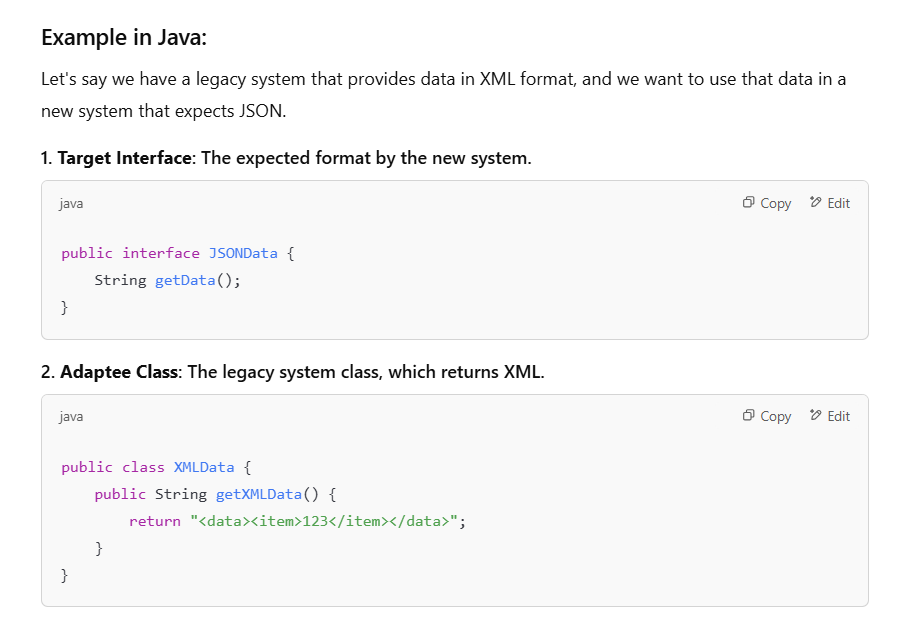


1 :- Adapter :-

Ex:-- calling a webservice need to convert object into json . this is nothing but adapter pattern

| **Pattern** | **Purpose** | **Use Case Example** |
| --- | --- | --- |
| **Adapter** | Converts one interface to another to make them compatible | Legacy system integration with a new system |







In this example, the Client is expecting data in JSON format, but the XMLData class provides data in XML. The XMLToJSONAdapter class acts as a bridge, adapting the XML data to the JSON format without modifying the existing XMLData class.

where we use adapter design pattern in java ?

The Adapter Design Pattern is typically used in Java when you want to allow two incompatible interfaces to work together. This pattern acts as a bridge between the two interfaces, making them compatible without changing their existing code.

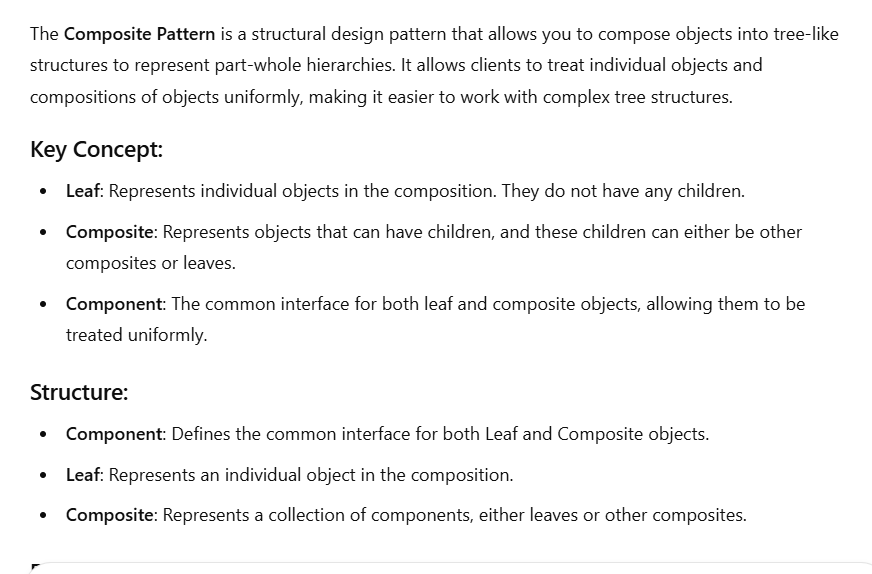
**Here are common use cases for the Adapter pattern:**

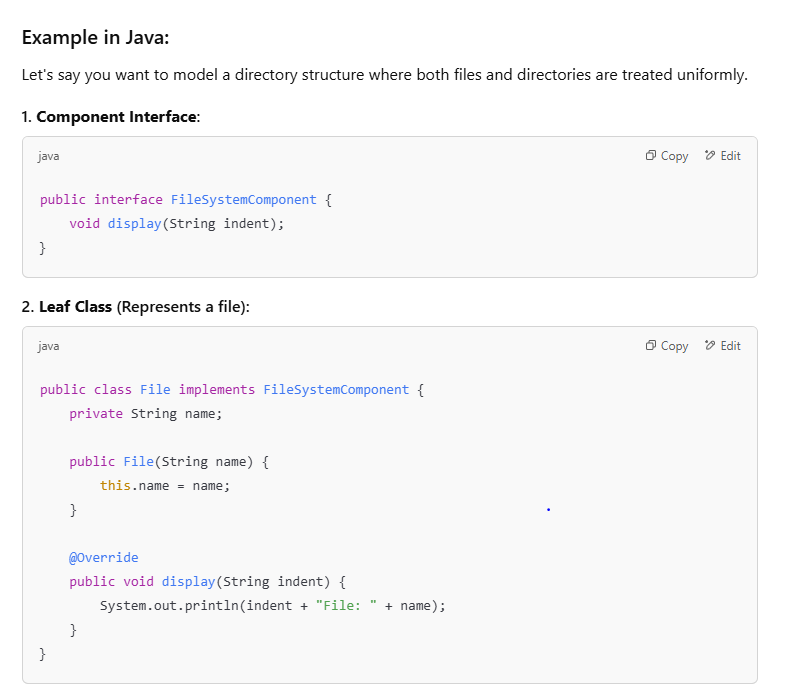
1. **Legacy Systems Integration**: When you need to integrate an old system or library with a new system, but their interfaces don't match. The adapter can convert the legacy interface to the new system’s interface.
2. **Third-Party Libraries**: When using third-party libraries that have different interface structures than your own code, you can create an adapter to make their API compatible with your code.
3. **Different Data Formats**: When you need to work with data that is in different formats (e.g., XML to JSON, or different network protocols), an adapter can be used to translate between these formats.
4. **UI Components**: Sometimes, different UI components or frameworks require slightly different interface structures. The Adapter pattern can help bridge these differences so that components can be used interchangeably.

**Example of an Adapter in Java**

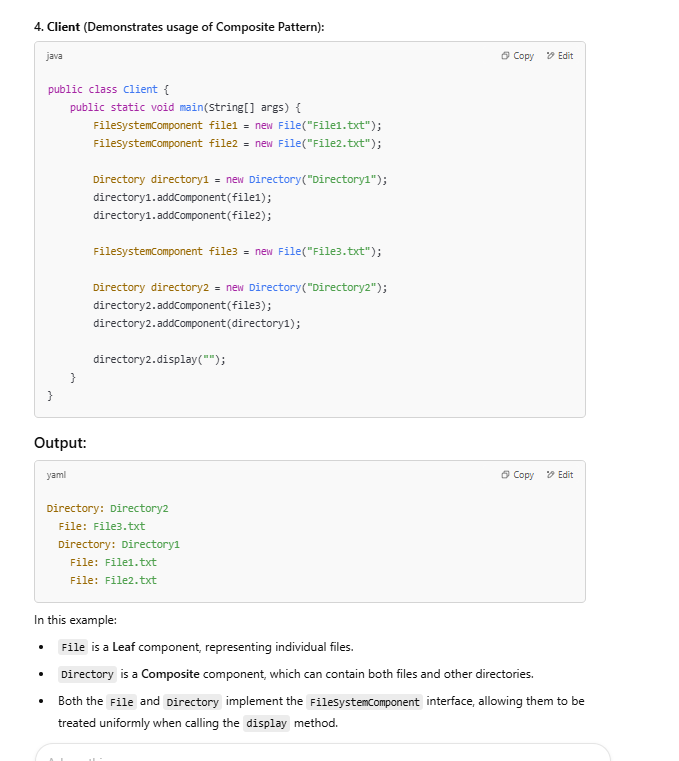
Imagine you're working with a legacy class OldSystem that has a method oldMethod(), but you need to use it in your modern system that requires a method newMethod().

2:-- Composite : example :- file system











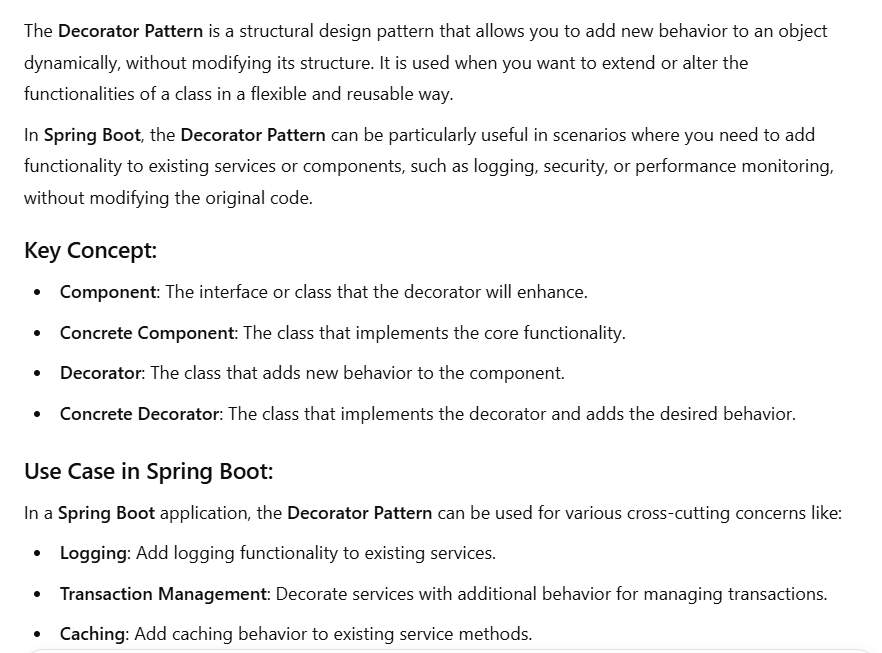
In this example, the OldSystemAdapter makes OldSystem work with the ModernSystem interface without modifying OldSystem. This is a classic use case of the Adapter Design Pattern in Java.

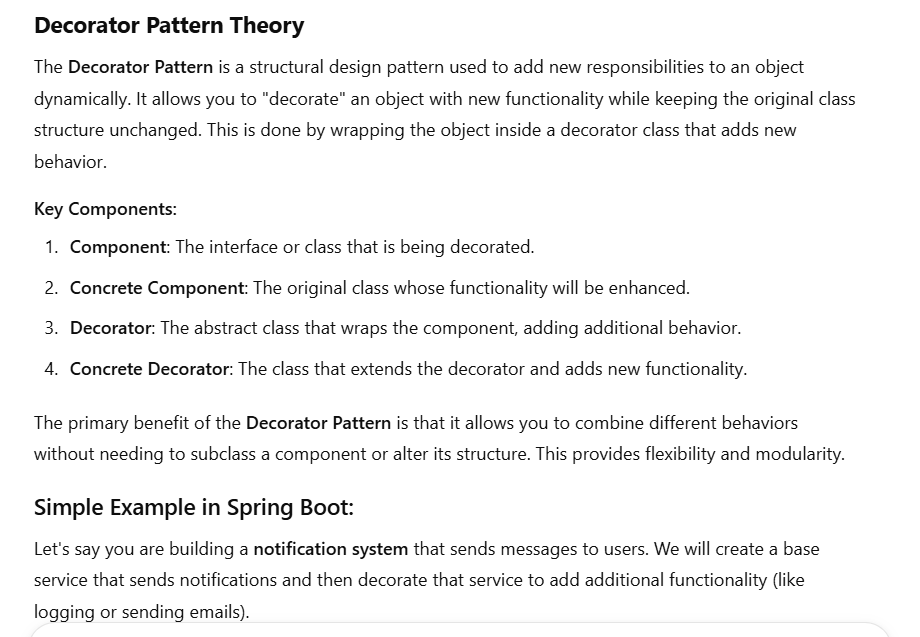
### Use Case in Java & Spring Boot:

In **Spring Boot**, the Composite Pattern can be applied in scenarios where you need to manage complex hierarchies or structures. Here are some use cases:

1. **Menu Structure**: If your application has a dynamic menu structure where menus can contain both individual menu items and other submenus, you can use the Composite Pattern to treat both the menu and the menu items uniformly.
2. **User Permissions**: If you're managing user permissions and roles where users can be individual users or groups of users, each group can contain individual users or other groups, you can use the Composite Pattern to represent these hierarchies.
3. **Content Management**: In applications where content can be composed of nested elements (like a blog or a website with pages containing sections, which in turn contain paragraphs), you can use the Composite Pattern to structure the content.

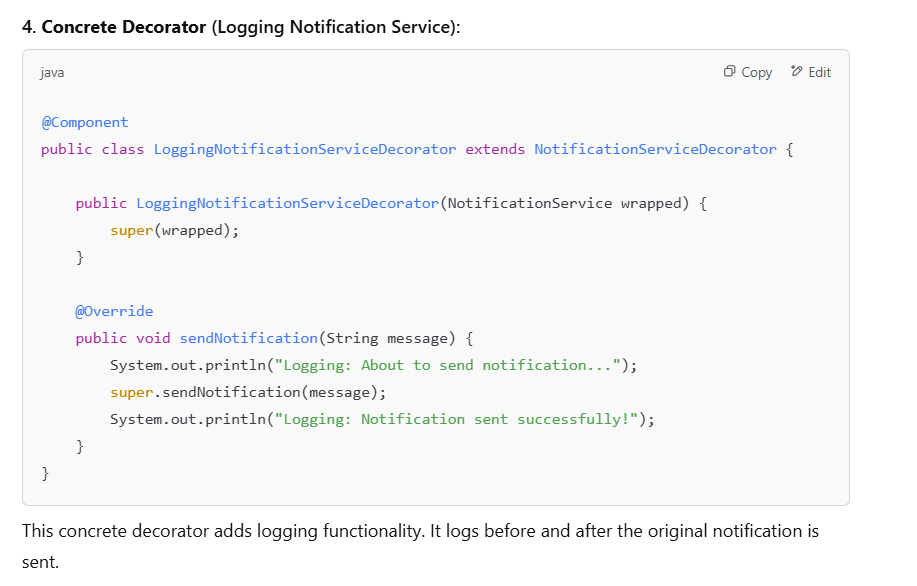
3: Decorator

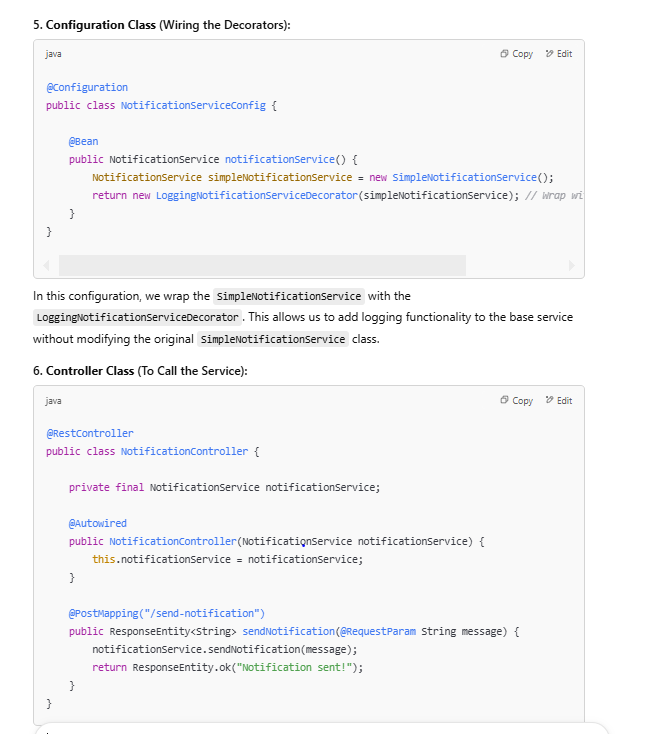


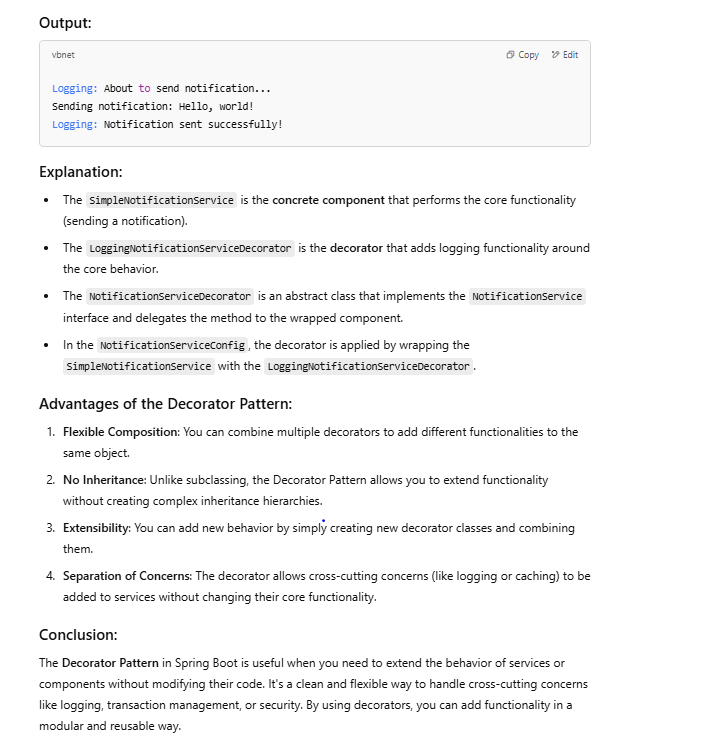












4: Façade :

Facade pattern makes a complex system easy to use for the client application

Here façade will have everything . Client just need to interact with façade and façade is responsible for everything else.Façade will be aware of other classes or method. Client will make single call to façade which will make all other call.Façade hides the complexity of the system. These can be multiple classes or these can be method within single class

**Single class represent entire subsystem**

**Provides a simplified interface to a complex subsystem.**

The **Facade Design Pattern** is a structural design pattern that provides a simplified interface to a complex subsystem or a group of classes. It helps to hide the complexity of the system by providing a higher-level interface that makes the subsystem easier to use.

**Key Concept:**

* **Facade**: Provides a simplified interface to a complex subsystem.
* **Subsystems**: The complex components that the facade simplifies.

The **Facade Pattern** is particularly useful in scenarios where you have a complex system with many interacting components, and you want to provide a simple interface to the clients of that system. It reduces the complexity and dependencies for the client by hiding the intricate details.

**Use Case of Facade in Spring Boot:**

In **Spring Boot**, the **Facade Pattern** can be useful in the following scenarios:

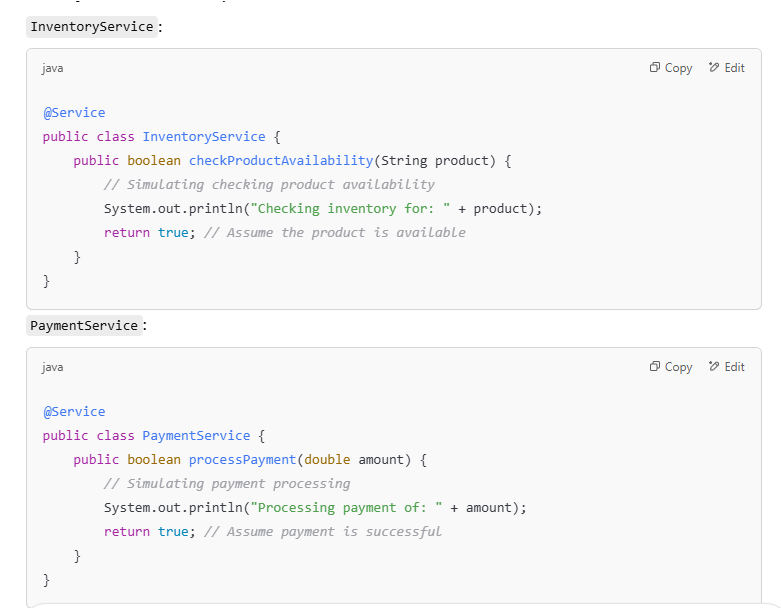
1. **Simplifying Complex Services**: If you have multiple services with complex logic (for example, a set of services responsible for handling different aspects of a payment processing system), the facade can provide a single entry point.
2. **Interacting with External Systems**: If you are interacting with several third-party APIs or complex subsystems (e.g., an email service, payment gateway, or external database), the facade can simplify the interaction for your controllers or clients.

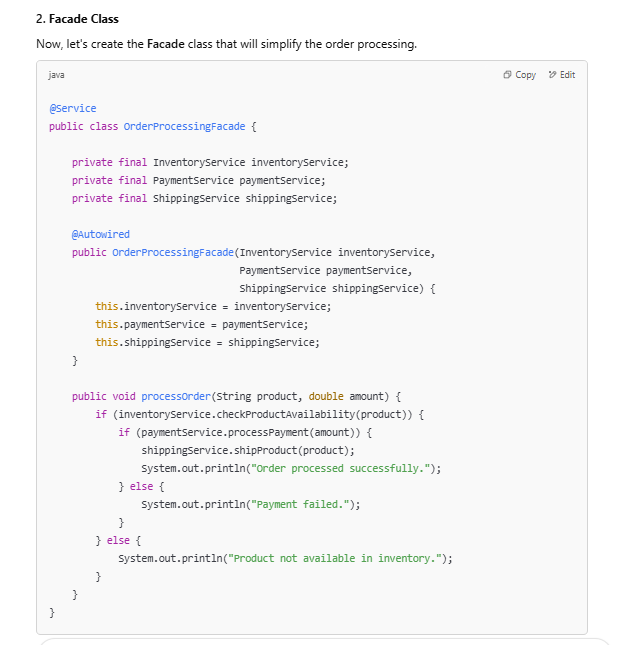
**Example: Facade Design Pattern in Spring Boot**

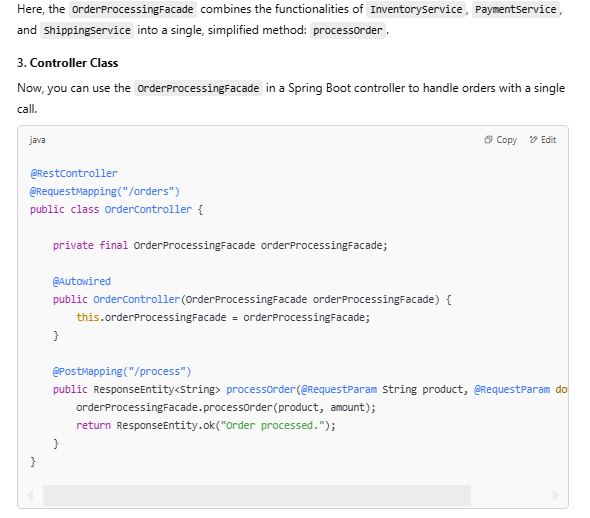
In this example, let's imagine that you are building an **Order Processing System**. You have several services that handle different parts of the order processing:

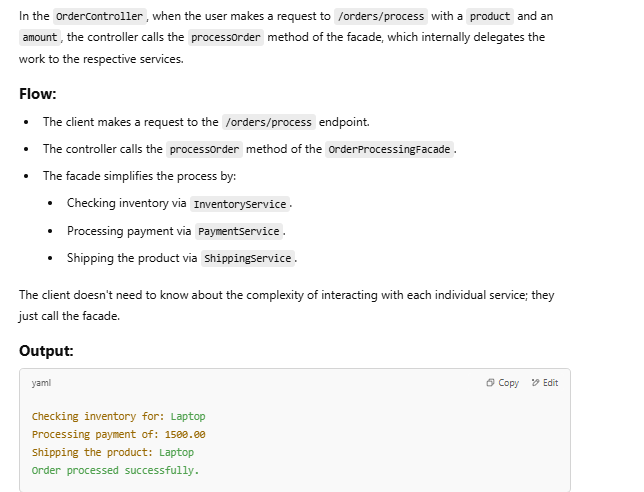
* **InventoryService** (check product availability),
* **PaymentService** (process payment),
* **ShippingService** (handle shipment).

You can create a **Facade** to simplify the order processing for the client.









**Advantages of Using Facade in Spring Boot:**

1. **Simplifies Complex Systems**: The facade hides the complexity of interacting with multiple services, making the system easier to use.
2. **Reduces Dependencies**: Clients do not need to interact with multiple services directly, reducing the dependencies and coupling between classes.
3. **Provides a Unified Interface**: The facade offers a single, unified interface for performing complex operations, such as order processing, without the client needing to know the internal details.
4. **Increases Flexibility**: You can change or replace individual subsystems (like changing the payment provider) without affecting the client code.

**When to Use Facade Pattern in Spring Boot:**

* **Complex Systems**: When you have a set of complex subsystems (e.g., user authentication, email services, payment processing, etc.) and want to provide a simpler interface to the client.
* **Integration with Third-Party Services**: When interacting with multiple third-party APIs and you want to simplify the interaction for your service layer.
* **Centralized Entry Point**: When you want to provide a centralized entry point to complex functionality that involves multiple services.

**Conclusion:**

The **Facade Pattern** is an excellent way to simplify interactions with a complex subsystem in Spring Boot. It reduces the complexity for the client by providing a simplified, unified interface while keeping the underlying system modular and flexible.

5:- Flyweight

The **Flyweight Design Pattern** is a structural design pattern that aims to reduce the number of objects created, to decrease memory usage, and to increase performance. It achieves this by sharing common data across multiple objects instead of having each object store its own copy of the data.

**Flyweight Pattern Components:**

1. **Flyweight**: This is the interface or abstract class that defines the shared and unshared properties.
2. **ConcreteFlyweight**: This is the concrete class implementing the flyweight interface. It stores the intrinsic (shared) state and is reusable.
3. **FlyweightFactory**: This is responsible for managing the creation and reuse of flyweight objects. It ensures that the flyweights are shared whenever possible.

**Use Case of Flyweight Pattern in Spring Boot:**

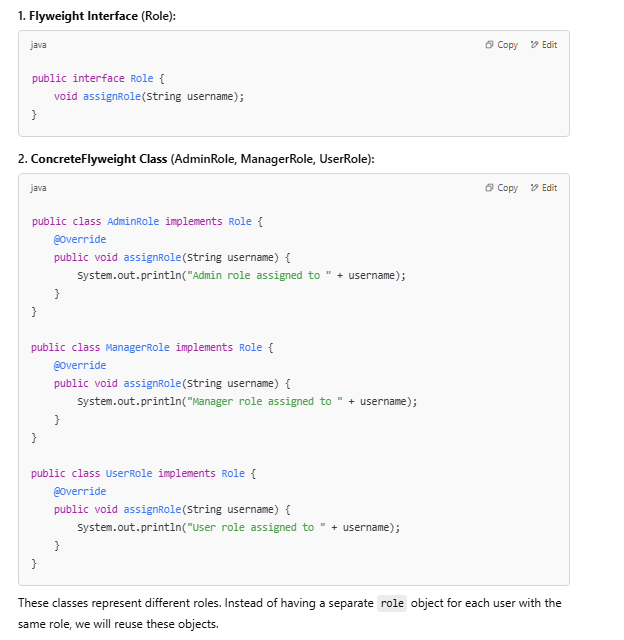
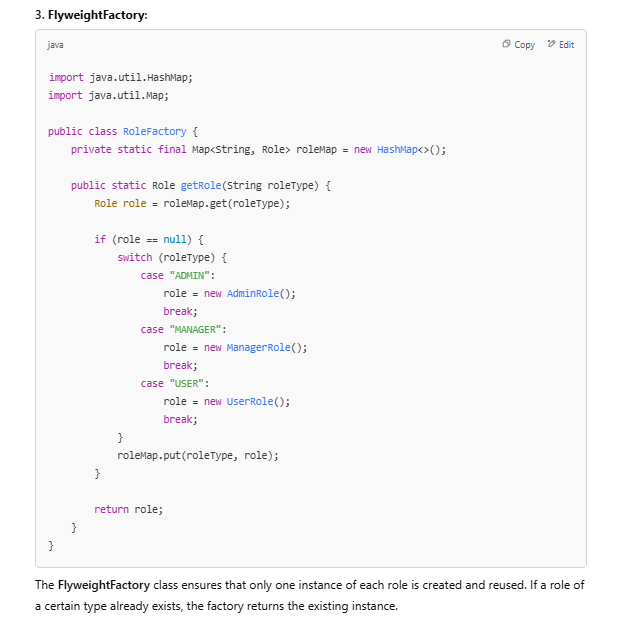
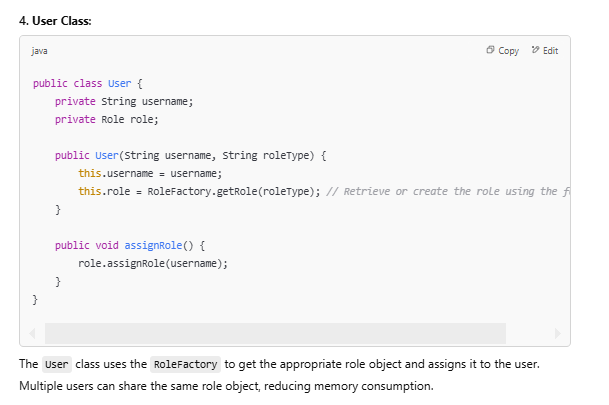
The Flyweight pattern is useful when:

1. **There are large numbers of objects** that share common state or properties.
2. **Memory optimization** is required, as many of these objects share common characteristics, so only the intrinsic state is stored.
3. **Reducing redundancy**: If you have similar data stored across objects, you can share it to save memory.

A **real-world example** of using the **Flyweight Design Pattern** in Spring Boot could be a **User Management System** where users have some shared information (e.g., role-based permissions, user types, or settings), and rather than creating many objects with the same information, you reuse objects that store this shared data.

**Example: Flyweight Design Pattern in a User Management System**

Let's consider a system that manages **users** with specific **roles** (like Admin, Manager, User). Each user has a role, but the role doesn't change per user. Instead of creating a new object for each user with the same role, we can use the **Flyweight Pattern** to share the role information



This simple Spring Boot controller allows you to create users with specific roles and assign them. The roles are reused across users, and only one instance of each role (Admin, Manager, User) is maintained.

**How the Flyweight Pattern Works:**

1. When a user is created, the role is fetched via the RoleFactory. If the role already exists (e.g., an "ADMIN" role), it is reused; otherwise, it is created and stored.
2. The User class doesn't store its own role data — it delegates the responsibility to the shared role object, thus minimizing memory usage.
3. The RoleFactory ensures that there is only one instance of each role type, effectively sharing the intrinsic state.

**Example Usage:**

* Creating a user with an Admin role:

pgsql

CopyEdit

POST /users/create?username=JohnDoe&role=ADMIN

This will output: User JohnDoe created with role ADMIN.

* Listing all users:

bash

CopyEdit

GET /users/list

This might return:

pgsql

CopyEdit

[JohnDoe assigned role AdminRole, JaneDoe assigned role ManagerRole]

**Flyweight Pattern Benefits:**

* **Memory Efficiency**: The same role object is shared by all users who have the same role, significantly reducing memory consumption when you have many users with the same role.
* **Performance**: The factory ensures that role objects are reused, which improves performance when creating many users with common roles.

**When to Use the Flyweight Pattern:**

* When you need to manage a large number of objects that share common data.
* When you have a lot of **intrinsic state** (shared information) and want to save memory by not duplicating this state.
* When creating **immutable objects** with shared data across multiple instances (e.g., roles, states, or settings that don't change).

**Conclusion:**

The **Flyweight Pattern** is useful when you need to manage objects that share common state to optimize memory usage. In the Spring Boot example, it helps by sharing role objects across multiple users, reducing memory consumption, and improving efficiency.

6: Proxy

.

The **Proxy Design Pattern** is used to provide an object representing another object, which acts as a surrogate or placeholder. The proxy controls access to the real object and can add additional functionality, such as lazy initialization, access control, logging, or monitoring. The proxy pattern is often used in scenarios like implementing remote method invocation, caching, or managing access to resources.

**Use Case of Proxy Pattern in Spring Boot:**

Let's say you're building an application where you want to control access to a sensitive service. For example, a **PaymentService** that processes payments might require special logging or security checks every time a payment is made. Using the **Proxy Pattern**, you can add this additional functionality without modifying the actual **PaymentService**.

**Example Scenario:**

1. **Real Object**: The PaymentService class that processes payments.
2. **Proxy Object**: A proxy class that adds logging and authorization before calling the actual payment processing.
3. **Use Case**: Add logging and authentication/authorization checks before the real payment is processed.

**Step-by-Step Implementation:**

