Search articles...

Proxy Design Pattern

Topic Tags:

System Design LLD

Problem Statement: Controlling Access 🔒

Imagine you're building a video streaming application. Your goal is to allow users to watch videos on demand. However, not all users have access to every video, and you need to restrict content based on user permissions (e.g., free vs premium users).

Each video requires significant bandwidth, and streaming them to unauthorized users would be costly. Additionally, some users might abuse the service by automating multiple video requests simultaneously.

The Problem: Directly accessing the video stream without restrictions can lead to:
• Unauthorized access to premium videos.
• Overloading the system with excessive requests.
• Increased bandwidth costs.
The Challenge: How can you create a solution that ensures only authorized users can access the video streams, while also optimizing bandwidth usage?

Solving It the Traditional Way: A Messy Solution 🛠️

Here's a naive implementation where the VideoService class handles all operations directly:

VideoService.java :

Java

```java
// VideoService.java
public class VideoService {
    public void playVideo(String userType, String videoName) {
        if (userType.equals("premium")) {
            System.out.println("Streaming premium video: " + videoName);
        } else if (userType.equals("free")) {
            System.out.println("Streaming free video: " + videoName);
        } else {
            System.out.println("Access denied: Invalid user type.");
        }
    }
}
```

Main.java

Java

```java
// Main.java
public class Main {
    public static void main(String[] args) {
        VideoService videoService = new VideoService();
        // Free user trying to watch a video
        videoService.playVideo("free", "Free Video 1");
        // Premium user trying to watch a video
        videoService.playVideo("premium", "Premium Video 1");
        // Unauthorized user
        videoService.playVideo("guest", "Video 1");
    }
}
```

The Problems with This Approach:

1. No Centralized Access Control: The VideoService class is responsible for both providing video content and checking permissions.
2. Code Duplication: Each request requires repetitive permission checks within the VideoService class.
3. Scalability Issues: Adding more user types or complex access rules requires significant changes to the VideoService class, making it hard to maintain.

Interviewer's Follow-up Questions: Can We Improve the Code? 🤔

An interviewer might ask:
1. How can we separate the responsibility of access control from the video streaming logic?
2. What if we want to cache frequently requested videos to reduce bandwidth usage?
3. Can we limit the number of requests a user can make to prevent abuse?
These questions highlight the need for a better design that centralizes access control and optimizes system performance.

Ugly Code: When We Realize the Code Needs Restructuring 🤦

Let's say we need to add more features like caching or limiting user requests:

VideoService.java :

Java

```java
public class VideoService {
    private Map<String, String> cachedVideos = new HashMap<>();
    private Map<String, Integer> requestCounts = new HashMap<>();
    public void playVideo(String userType, String videoName) {
        // Limit requests
        requestCounts.put(userType, requestCounts.getOrDefault(userType, 0
        if (requestCounts.get(userType) > 5) {
            System.out.println("Access denied: Too many requests.");
            return;
        }

        // Caching logic
        if (cachedVideos.containsKey(videoName)) {
            System.out.println("Streaming cached video: " + videoName);
        } else {
            System.out.println("Streaming new video: " + videoName);
```

```
            cachedVideos.put(videoName, videoName);
        }
    }
}
```

Main.java

Java

```java
// Main.java
public class Main {
    public static void main(String[] args) {
        VideoService videoService = new VideoService();
        // Free user trying to watch a video
        videoService.playVideo("free", "Free Video 1");
        // Premium user trying to watch a video
        videoService.playVideo("premium", "Premium Video 1");
        // Unauthorized user
        videoService.playVideo("guest", "Video 1");
    }
}
```

Why is this Code Problematic ?

1. Complexity: Adding more features makes the class harder to read and maintain.
2. Poor Scalability: Adding new access rules or caching mechanisms requires changes in multiple places.

The Savior: Proxy Design Pattern 🙏

The Proxy Design Pattern is the ideal solution for this problem. It provides a surrogate or placeholder for another object, controlling access to it. In our example, the proxy acts as a gatekeeper between users and the VideoService class.

How the Proxy Design Pattern Works 🔧

1. Subject: Define a common interface for the real object and the proxy.
2. Real Subject: The actual object that performs the core operations (e.g., VideoService).
3. Proxy: Controls access to the real object, adding additional functionality like caching, access control, or request limiting.

Solving the Problem with Proxy Design Pattern 🌐

Step 1: Define the VideoService Interface :

This interface provides the blueprint for the real service and the proxy:
VideoServiceInterface.java :

Java

```java
// VideoServiceInterface.java
public interface VideoServiceInterface {
    void playVideo(String userType, String videoName);
}
```

Step 2: Implement the Real VideoService Class :

In this step, we will create the RealVideoService class that implements the VideoService interface. This class will provide the actual implementation for the methods defined in the interface, such as loading and playing videos.
RealVideoService.java :

Java

```java
// RealVideoService.java
public class RealVideoService implements VideoServiceInterface {
    @Override
    public void playVideo(String userType, String videoName) {
        System.out.println("Streaming video: " + videoName);
    }
}
```

Step 3: Implement the Proxy Class :

In this step, we will create the ProxyVideoService class that implements the VideoService interface. The proxy class will control access to the RealVideoService and add additional functionality, such as caching or access control, without modifying the actual implementation of the RealVideoService

Java

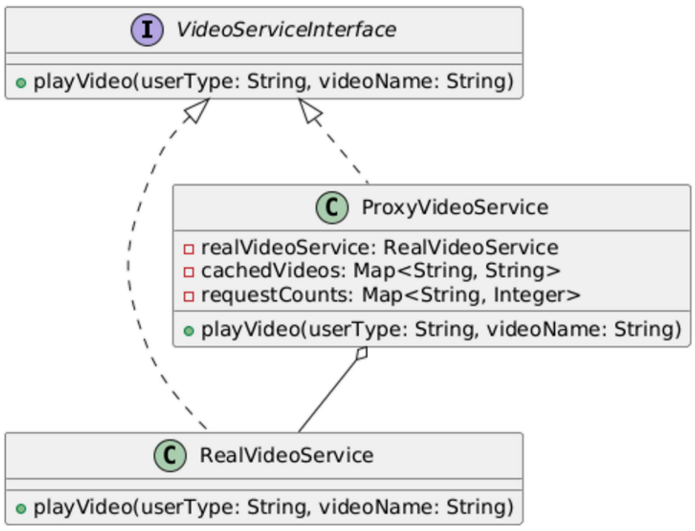
```java
// ProxyVideoService.java
import java.util.HashMap;
import java.util.Map;
public class ProxyVideoService implements VideoServiceInterface {
  private RealVideoService realVideoService;
  private Map<String, String> cachedVideos = new HashMap<>();
  private Map<String, Integer> requestCounts = new HashMap<>();
  public ProxyVideoService(RealVideoService realVideoService) {
    this.realVideoService = realVideoService;
  }

  @Override
  public void playVideo(String userType, String videoName) {
    // Check user permissions
    if (!userType.equals("premium") && videoName.startsWith("Premium")) {
      System.out.println(
          "Access denied: Premium video requires a premium account.");
      return;
    }

    // Limit requests
    requestCounts.put(userType, requestCounts.getOrDefault(userType, 0) +
    if (requestCounts.get(userType) > 5) {
      System.out.println("Access denied: Too many requests.");
      return;
    }

    // Caching logic
    if (cachedVideos.containsKey(videoName)) {
      System.out.println("Streaming cached video: " + videoName);
    } else {
      realVideoService.playVideo(userType, videoName);
      cachedVideos.put(videoName, videoName);
    }
  }
}
```

Step 4: Use the Proxy in the Application :

In this step, we will integrate the ProxyVideoService into our application to control access to the RealVideoService. By using the proxy, we can add additional functionality, such as caching or access control, without modifying the actual implementation of the RealVideoService. This approach allows us to manage the complexity and enhance the functionality of the video service in a flexible and maintainable way.

Java

```java
// Main.java
public class Main {
    public static void main(String[] args) {
        RealVideoService realService = new RealVideoService();
        ProxyVideoService proxyService = new ProxyVideoService(realService
        // Free user trying to watch a video
        proxyService.playVideo("free", "Free Video 1");
        // Premium user trying to watch a video
        proxyService.playVideo("premium", "Premium Video 1");
        // Unauthorized user
        proxyService.playVideo("guest", "Video 1");
        // Too many requests
        for (int i = 0; i < 6; i++) {
            proxyService.playVideo("free", "Free Video 2");
        }
    }
}
```



Advantages of Using the Proxy Design Pattern 🏆

1. Centralized Access Control:

The proxy manages access rules, ensuring consistent and secure access to the real service.

2. Caching:
Frequently requested videos can be cached in the proxy, reducing bandwidth usage and improving performance.

3. Request Limiting:
The proxy can enforce limits on user requests, preventing abuse of the service.

4. Scalability:
Adding new access rules or optimizations requires changes only in the proxy, leaving the real service untouched.

Real-life Use Cases of the Proxy Pattern 🌎

1. Security Proxies:

Control access to sensitive resources, ensuring only authorized users can access them.

2. Caching Proxies:
Cache frequently requested data, improving performance and reducing load on the server.

3. Virtual Proxies:
Delay the creation of expensive objects until they are actually needed.

4. Firewall Proxies:

Filter and control network traffic based on predefined rules.

Conclusion 🎯

The Proxy Design Pattern is a powerful tool for controlling access to objects while adding additional functionality like caching and request limiting. In our video streaming application example, the proxy acts as a gatekeeper, ensuring secure, efficient, and scalable access to video content.
By decoupling access control from the core service logic, the Proxy Pattern simplifies maintenance, enhances security, and improves performance. Whether you're building a video streaming service, a security system, or a caching mechanism, the Proxy Pattern is an essential design pattern for clean and robust architecture.