MVC Pattern

Resources

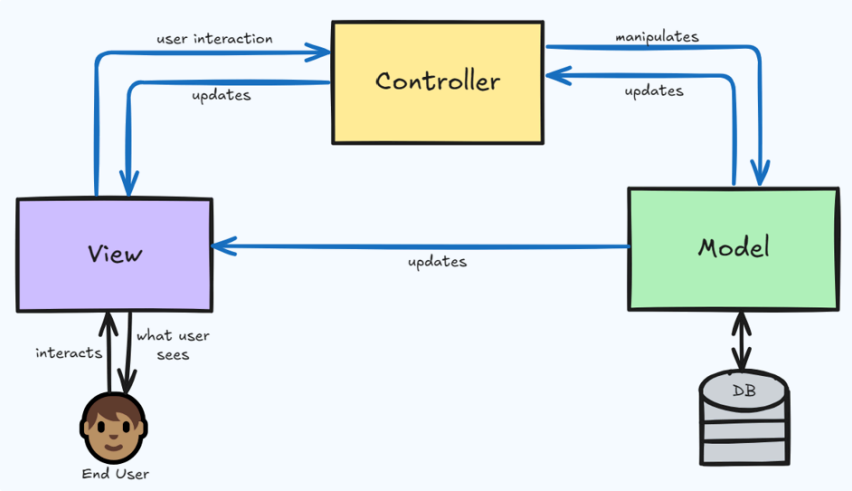
- 37. MVC Design Pattern | MVC Architecture Overview | Low Level System Design

What is an MVC Pattern?

MVC stands for Model-View-Controller. It is one of the popular software architectural patterns. It is used by the development team to organise and structure their code into three separate, independent and interconnected components/ distinct layers of the application, each having its own set of responsibilities. This separation makes your code cleaner, easier to maintain, and scalable.

MVC Architecture

Structure of MVC Pattern



Role of the 3 Components

The MVC framework consists of three main components:

1. **Model:** This is the **“data”** part. The Model represents the data stored in your application. This can be a simple in-memory data structure, an internal or external data storage system. This component is responsible for storing and managing application-specific data-related operations like database interactions, data validation, business rules and logic.

2. **View:** This is the **“UI”** part. It is what the user sees and interacts with. It accepts the data provided by the model and displays it on screen in a way that users can interact with. The View component does not contain any logic. It is responsible for displaying data and handling user input.
3. **Controller:** The **“brain”** part. A component that connects both the Model and the View. It contains application business logic and is responsible for handling application flow and navigation by coordinating with the other two components. It receives user input, processes it, figures out what to do with it, calls the appropriate function on the Model and decides what view to render upon success/failure. The data received from the Model is passed to the View for rendering.

How It Works

1. The **user** interacts with the **View** (clicks a button, submits a form).
2. The **Controller** receives the input from the View, processes it and updates the **Model** based on the input.
3. The **Model** notifies the **View** about changes in data.
4. The **View** updates itself to reflect the new data.

Real-life Examples

1. Spring Boot → A popular Java framework that uses MVC
2. Django → A high-level web development framework in Python(MVT variant)
3. Ruby on Rails

Implementation

Let’s understand the MVC Pattern using a simple Blog Application example:

```java
package com.conceptcoding.additionalpatterns.model;

import java.util.Date;

// Model class: Blog - Holds data about a single blog post.
public class Blog {
    private String title;
    private String content;
    private String author;
    private Date createdAt;

    // Constructor
    public Blog(String title, String content, String author, Date createdAt) {
        this.title = title;
        this.content = content;
        this.author = author;
        this.createdAt = createdAt;
    }

    // Getters and Setters
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
```

```
        this.content = content;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }
}
```

```
package com.conceptcoding.additionalpatterns.view;

import com.conceptcoding.additionalpatterns.model.Blog;

import java.text.SimpleDateFormat;
import java.util.List;

// View class: BlogView
// Responsible for displaying blog post details in the console.
// No business logic — just formatting and printing.
public class BlogView {

    // Display a single blog post
    public void displayBlogDetails(Blog blog) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy
HH:mm:ss");

        System.out.println("===== Blog Post =====");
        System.out.println("Title   : " + blog.getTitle());
        System.out.println("Author  : " + blog.getAuthor());
        System.out.println("Date    : " +
sdf.format(blog.getCreatedAt()));
        System.out.println("Content : " + blog.getContent());
    }

    // Display a list of all blog posts
    public void displayAllBlogs(List<Blog> blogs) {
        System.out.println("===== All Blog Posts =====");
        for (Blog blog : blogs) {
            System.out.println("- " + blog.getTitle() + " by " +
blog.getAuthor());
        }
    }
}
```

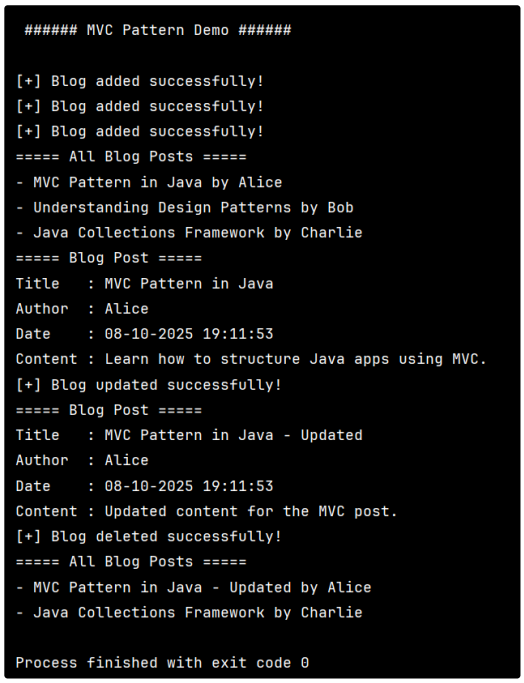
```
package com.conceptcoding.additionalpatterns.controller;

import com.conceptcoding.additionalpatterns.model.Blog;
import com.conceptcoding.additionalpatterns.view.BlogView;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

// Controller class: BlogController
// Acts as a bridge between Model (Blog) and View (BlogView).
// Handles creation, updating, and displaying of blog posts.
public class BlogController {

    private final List<Blog> blogs; // acts as an in-memory database
    private final BlogView view;
```

```
17
18      // Constructor connects controller with the view
19      public BlogController(BlogView view) {
20          this.view = view;
21          this.blogs = new ArrayList<>();
22      }
23
24      // Add a new blog post
25      public void addBlog(String title, String content, String author) {
26          Blog blog = new Blog(title, content, author, new Date());
27          blogs.add(blog);
28          System.out.println("[+] Blog added successfully!");
29      }
30
31      // Update an existing blog by index
32      public void updateBlog(int index, String newTitle, String
    newContent) {
33          if (index >= 0 && index < blogs.size()) {
34              Blog blog = blogs.get(index);
35              blog.setTitle(newTitle);
36              blog.setContent(newContent);
37              System.out.println("[+] Blog updated successfully!");
38          } else {
39              System.out.println("[+] Invalid blog index!");
40          }
41      }
42
43      // Delete a blog post
44      public void deleteBlog(int index) {
45          if (index >= 0 && index < blogs.size()) {
46              Blog removed = blogs.remove(index);
47              System.out.println("[+] Blog deleted successfully!");
48          } else {
49              System.out.println("[+] Invalid blog index!");
50          }
51      }
52
53      // Display a single blog post
54      public void showBlog(int index) {
55          if (index >= 0 && index < blogs.size()) {
56              view.displayBlogDetails(blogs.get(index));
57          } else {
58              System.out.println("[+] Invalid blog index!");
59          }
60      }
61
62      // Display all blogs
63      public void showAllBlogs() {
64          view.displayAllBlogs(blogs);
65      }
66  }
```

```
1   package com.conceptcoding.additionalpatterns;
2
3   import com.conceptcoding.additionalpatterns.controller.BlogController;
4   import com.conceptcoding.additionalpatterns.view.BlogView;
5
6   // Demonstrates MVC pattern in action for Blog application.
7   public class MVCPatternDemo {
8
9       public static void main(String[] args) {
10          System.out.println("\n ###### MVC Pattern Demo ###### \n");
11
12          // Create the view
13          BlogView view = new BlogView();
14
15          // Create the controller (connected with the view)
16          BlogController controller = new BlogController(view);
17
18          // Add some blog posts
```

```
        controller.addBlog("MVC Pattern in Java", "Learn how to structure Java apps using MVC.", "Alice");
        controller.addBlog("Understanding Design Patterns", "Design patterns make your code reusable and clean.", "Bob");
        controller.addBlog("Java Collections Framework", "Learn about different collections and their use cases.", "Charlie");

        // Display all blogs
        controller.showAllBlogs();

        // Show details of a specific blog
        controller.showBlog(0);

        // Update first blog
        controller.updateBlog(0, "MVC Pattern in Java - Updated", "Updated content for the MVC post.");

        // Show updated blog
        controller.showBlog(0);

        // Delete second blog
        controller.deleteBlog(1);

        // Show remaining blogs
        controller.showAllBlogs();
    }
}
```

Output



Advantages of the MVC Pattern

- **Separation of Concerns:** Each component has a distinct responsibility.
- **Speeds up the development process** as multiple developers can work simultaneously on Model, View, and Controller.
- **Easier Maintenance:** Changes in one component don't affect others.
- **Increased Reusability:** Models can be reused across different views. This allows for Multiple Views Support, which means that one Model can have multiple Views, for example, the same data shown as a table, chart, or graph.

- **Easier Testability:** Each component can be tested independently. Mock objects can be used for isolated testing with in-memory data mimicking an external data source.
- **Easier to scale** the application as new features can be added without disrupting existing code.
- **Increased Flexibility:** Easy to modify or replace one component without affecting others. For example, databases can be switched without changing the presentation and altering UI without touching business logic.

Disadvantages of the MVC Pattern

- **Increased Complexity:** This is an overkill for simple applications. More files and classes to manage.
- **Tight Coupling in Some Cases:** Controller often depends on both Model and View and hence changes in Model might require Controller updates.
- **Performance Overhead:** Multiple layers can add latency and leads to more objects in memory.
- **Risk of Over-Engineering:** MVC is not adviced to be used for small applications.
- **Maintainance cost** could be increase for maintaining multiple components.