RESUMEN MVC

¿Qué es MVC?

El patrón de diseño Modelo-Vista-Controlador (MVC) especifica que una aplicación consta de un modelo de datos, información de presentación e información de control. El patrón requiere que cada uno de estos se separe en objetos diferentes.

El patrón MVC separa las preocupaciones de una aplicación en tres componentes distintos, cada uno responsable de un aspecto específico de la funcionalidad de la aplicación.

Esta separación de preocupaciones hace que la aplicación sea más fácil de mantener y ampliar, ya que los cambios en un componente no requieren cambios en los demás componentes.

El patrón de diseño MVC (Modelo-Vista-Controlador) divide una aplicación en tres partes: el Modelo (que gestiona los datos), la Vista (que es lo que ven los usuarios) y el Controlador (que conecta ambas). Esto facilita trabajar en cada parte por separado, permitiendo actualizar o corregir errores sin afectar la aplicación completa. Ayuda a los desarrolladores a añadir nuevas funciones sin problemas, simplifica las pruebas y permite mejores interfaces de usuario. En resumen, MVC ayuda a mantener todo organizado y mejora la calidad del software.

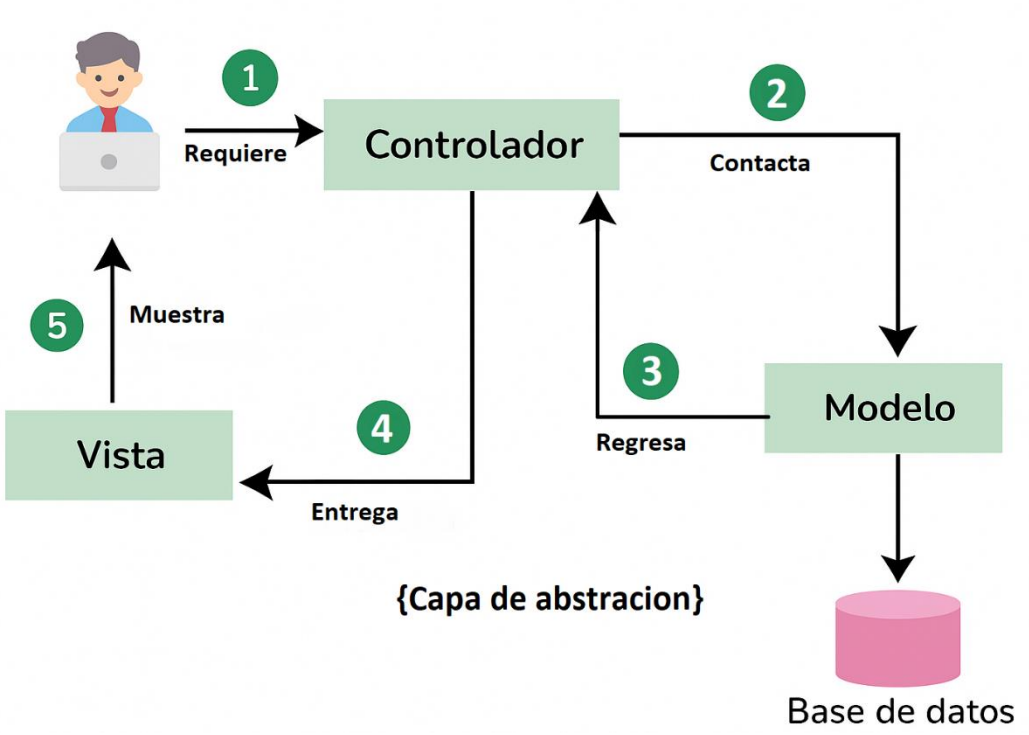
Arquitectura Modelo-Vista-Controlador

El patrón de arquitectura de software modelo-vista-controlador (MVC) se utiliza para crear aplicaciones web y de escritorio en una amplia variedad de lenguajes de programación. MVC divide el código de la aplicación en tres áreas principales de responsabilidad:

- Modelo. Los modelos organizan y gestionan los datos de la aplicación, almacenan el estado del sistema y se comunican con otras partes de la aplicación cuando se producen cambios relevantes en los datos.
- Vista. Las vistas proporcionan a los usuarios de la aplicación visualizaciones de los datos de la aplicación y del estado del sistema.
- Controlador. Los controladores proporcionan mecanismos para que los usuarios de la aplicación interactúen con los datos de la aplicación y el estado del sistema, y los modifiquen.

Los modelos, las vistas y los controladores son clases independientes con responsabilidades específicas y relaciones definidas. El modelo no se comunica directamente con las vistas ni con los controladores, sino que transmite eventos que notifican cambios en sus datos o estado. Las vistas y los controladores tienen una referencia al modelo, pero no entre sí. Además, comparten una cantidad significativa de código. Este código compartido se puede trasladar a una superclase de componente

para reducir la duplicación de código y facilitar el desarrollo de vistas y controladores adicionales.



Comunicación entre los componentes

El siguiente flujo de comunicación garantiza que cada componente sea responsable de un aspecto específico de la funcionalidad de la aplicación, lo que conduce a una arquitectura más escalable y fácil de mantener.

- Interacción del usuario con la vista: el usuario interactúa con la vista, como hacer clic en un botón o ingresar texto en un formulario.
- La vista recibe la entrada del usuario: la vista recibe la entrada del usuario y la envía al controlador.
- El controlador procesa la entrada del usuario: El controlador recibe la entrada del usuario desde la vista. La interpreta, realiza las operaciones necesarias (como actualizar el modelo) y decide cómo responder.
- El controlador actualiza el modelo: el controlador actualiza el modelo en función de la entrada del usuario o la lógica de la aplicación.
- El modelo notifica a la vista sobre los cambios: si el modelo cambia, notifica a la vista.
- Vista solicita datos del modelo: la vista solicita datos del modelo para actualizar su visualización.
- El controlador actualiza la vista: el controlador actualiza la vista en función de los cambios en el modelo o en respuesta a la entrada del usuario.
- La vista representa la IU actualizada: la vista representa la IU actualizada en función de los cambios realizados por el controlador.

Cuando utilizar el patrón de diseño MVC

- Aplicaciones complejas: Use MVC para aplicaciones con muchas funciones e interacciones de usuario, como sitios de comercio electrónico. Ayuda a organizar el código y a gestionar la complejidad.
- Cambios frecuentes en la interfaz de usuario: si la interfaz de usuario necesita actualizaciones periódicas, MVC permite realizar cambios en la vista sin afectar la lógica subyacente.
- Reutilización de componentes: si desea reutilizar partes de su aplicación en otros proyectos, la estructura modular de MVC lo hace más fácil.
- Requisitos de prueba: MVC admite pruebas exhaustivas, lo que le permite probar cada componente por separado para un mejor control de calidad.

Cuando no utilizar el patrón de diseño MVC

- Aplicaciones sencillas: Para aplicaciones pequeñas con funcionalidad limitada, MVC puede añadir complejidad innecesaria. Un enfoque más simple podría ser mejor.
- Aplicaciones en tiempo real: MVC puede no funcionar bien para aplicaciones que requieren actualizaciones inmediatas, como juegos en línea o aplicaciones de chat.
- Interfaz de usuario y lógica estrechamente acopladas: si la interfaz de usuario y la lógica empresarial están estrechamente vinculadas, MVC podría complicar aún más las cosas.
- Recursos limitados: para equipos pequeños o aquellos que no están familiarizados con MVC, los diseños más simples pueden conducir a un desarrollo más rápido y a menos problemas.

Ejemplo

1. Modelo (Clase de estudiantes)

Representa los datos (nombre del estudiante y número de matrícula) y proporciona métodos para acceder y modificar estos datos.

```
2. class Student {
3.     private String rollNo;
4.     private String name;
5.
6.     public String getRollNo() {
7.         return rollNo;
8.     }
9.
10.    public void setRollNo(String rollNo) {
11.        this.rollNo = rollNo;
12.    }
13.
14.    public String getName() {
15.        return name;
16.    }
17.
18.    public void setName(String name) {
19.        this.name = name;
20.    }
21.}
```

2. Vista (clase StudentView)

Representa cómo se deben mostrar los datos (datos del estudiante) al usuario. Contiene un método ( ) para imprimir el nombre y el número de matrícula del estudiante.

```
class StudentView {
    public void printStudentDetails(String studentName, String
studentRollNo) {
        System.out.println("Student:");
        System.out.println("Name: " + studentName);
        System.out.println("Roll No: " + studentRollNo);
    }
}
```

3. Controlador (clase StudentController)

Actúa como intermediario entre el Modelo y la Vista. Contiene referencias a los objetos Modelo y Vista. Proporciona métodos para actualizar el Modelo y la Vista.

```
class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name) {
        model.setName(name);
    }

    public String getStudentName() {
        return model.getName();
    }

    public void setStudentRollNo(String rollNo) {
        model.setRollNo(rollNo);
    }

    public String getStudentRollNo() {
        return model.getRollNo();
    }

    public void updateView() {
        view.printStudentDetails(model.getName(), model.getRollNo());
    }
}
```