# Model-View-Controller (MVC)

## Simple Introduction

- What is MVC?
- Why do we need it?
- How it helps users

# Section 1: Why MVC?

## What we will see

- Problem with complex systems
- Need for better design
- Idea of MVC

# Problem in Software

- Systems are complex
- Hard for users to understand
- Hard to change

## Analogy

A messy kitchen → hard to cook

# Think About

- Why do apps feel confusing sometimes?
- Is it design or user problem?

# Mental Models

- Users need to *understand* systems
- Systems should match user thinking

## Analogy

Map vs real city

# Reflect

- When did you feel lost using software?
- What helped you understand it?

# Summary: Why MVC

- Systems are complex
- Users need clarity
- Design should match thinking

# Section 2: Big Idea of MVC

## What we will see

- Model
- View
- Controller

# MVC Overview

- Model → Data + Business Logic
- View → Display
- Controller → Receives input, orchestrates Model and View

## Analogy

Restaurant:

- Kitchen = Model
- Menu/plate = View
- Waiter = Controller

Any questions so far?

# Model

- Stores data
- Contains business rules and validation
- Manages state

### Example

Bank: stores balance, enforces "no overdraft" rule

# View

- Shows data to user

**Example**

ATM screen

# Controller

- Receives user input
- Calls the right Model operation
- Selects which View to show next

### Example

ATM buttons → Controller checks PIN (Model) → shows balance screen (View)

# Think Deeper

- Can Model exist without View?
- Can View exist without Model?

# Answer

- **Yes** — Model can exist without View (e.g., a background job processing data)
- **No** — View needs a Model to have something to display
- This is why Model is the most independent component

# Worked Example: To-Do App

Three parts working together:

```
Model      → stores list of tasks ["Buy milk", "Call Amma"]
View       → shows tasks as a list on screen
Controller → handles "Add task" button click
```

Flow:

1. User clicks **Add** → Controller receives the click
2. Controller calls Model → Model adds task to list
3. Model notifies View → View re-renders the updated list

# Section 3: Real System Problems

## What we will see

- Large systems
- Multiple domains
- Ownership issues

# Domain Services

- Systems have many parts
- Each part = domain

**Example**

Bank:

- Accounts
- Loans
- Payments

# Analogy

Company departments:

- HR
- Finance
- Operations

Each has its own system

# Think About

- Should all systems be one big system?

# Monolith vs Modules

- **Monolith** — one big system, simpler to start, hard to scale
- **Modular** — split into domains, each team owns one part
- Real-world systems usually start as monoliths and split later

# Ownership Problem

- Many teams involved
- Hard to manage everything

# Solution

- Split into smaller parts
- Each team owns one part

## Analogy

Group project split into tasks

# Summary: System Design

- Divide system into domains
- Assign ownership
- Integrate carefully

# Section 4: User Perspective

## What we will see

- Personal tools
- Tasks vs systems

# Personal Information Systems

- Users do tasks
- Tasks need many systems

# Example

Designer needs:

- Design tool
- Time tracking

# Analogy

Phone apps:

- Maps
- Messages
- Notes

All separate, but used together

# Summary: User Needs

- Users think in tasks
- Not systems
- Tools should match tasks

# Section 5: Deep MVC Patterns

## What we will see

- Model vs Editor
- View vs Controller
- Tools

# Model vs Editor

- Model → data
- Editor → interaction

# Why Separate?

- Data is complex
- UI should be simple

# Input / Output Separation

- View → display
- Controller → input

# Analogy

TV:

- Screen = View
- Remote = Controller

# Tools for Tasks

- UI = combination of parts

# Composite Tool

- Many views together
- One tool manages them

# Example

IDE (code editor):

- File explorer
- Code window
- Terminal

# Summary: Advanced MVC

- Separate concerns
- Combine for tasks
- Keep system flexible

# Section 6: Synchronization

## What we will see

- Keeping UI consistent

# Problem

- Multiple views
- Same data

# Synchronize Selection

- Select in one place
- Update everywhere

# Synchronize Model & View

- When data changes
- UI updates

# Analogy

Google Docs:

- One edit → all users see it

# Reflect

- Have you seen delayed updates in apps?

# Summary: Sync

- Keep data + UI consistent
- Model notifies View via events/callbacks (Observer pattern)
- This is how original Smalltalk MVC worked — modern web frameworks handle it differently (e.g., React re-renders on state change)

# Section 7: Future

## What we will see

- Where MVC is going

# Future Ideas

MVC evolved as UIs became more complex:

- **MVVM** (Model-View-ViewModel) — used in Android, Angular; View binds directly to ViewModel
- **Component-based** — React, Vue break UI into self-contained components; each manages its own state
- **REST APIs** — Controller becomes an API endpoint; View is a separate frontend app

## Key trend

Separation of concerns remains the core idea — only the boundaries shift

# Key Idea

Users should:

- Understand systems
- Control systems

# When NOT to Use MVC

MVC adds structure — but structure has a cost

- A simple script that reads a CSV and prints output? → No MVC needed
- A static webpage with no dynamic data? → No MVC needed
- A small tool used by one person? → Probably overkill

## Rule of thumb

Use MVC when **data, display, and user interaction** are all present and likely to change independently

# Think About This 🤔

- Can users build their own tools?

# Final Summary

- MVC separates concerns
- Helps users understand systems
- Makes software flexible

Questions?

# Thank you!

Any questions?

Thejesh GN