Architectural Pattern

Layered Architecture Pattern :-

- n-tiered patterns
- organized in horizontal layer
- self-dependent
- all components are interconnected
- Responsibilities

Layer

1. Presentation layer: User interface (display / enter data)
2. Business Layer: Responsible for business logic as per rqst
3. Persistent / Application layer: medium of communication between 'presentation layer' and 'data layer'. Handles function like - obj relational mapping.
4. Data layer: Has a data storage system / database for managing data

✓ Monolithic Architecture

Pros: • Scalability → scaled independtly
• flexibility → Diff technology
• Maintainability → changes in one layer does not affect other.

Cons: • Complexity → more layer, diff to mng
• Performance overhead ← introduce latency
• Strict Layer Separation (lead to inefficiencies & increase development effort)

✓ UI never talks directly to Database

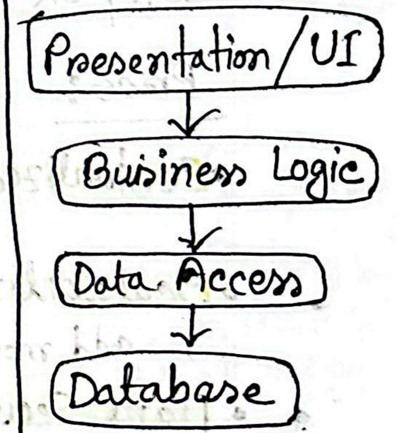
✓ Loose coupling, clean flow

✓ High maintainability

✓ Top to Bottom dependency

✗ Not ideal for highly scalable system

✗ Multiple layers lead Performance overhead



Client-Server Architecture:

√network model

√client-server → communicate to specific task/share data

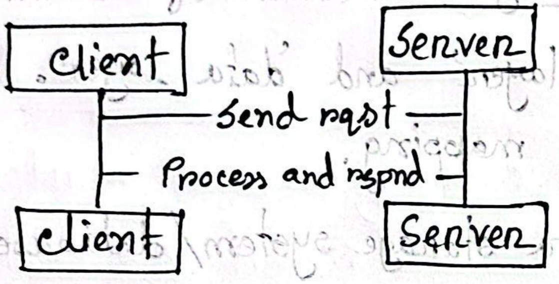
Client initiate req (computer)

server proccesses these req (another computer/server)

retrieves the relevant info

displays it to user

Types: **Two-tier Architecture:** simplest form



Three tier Architecture: Middleware layer, often called the application layer, ~~such as~~ between client and server

n-tier Architecture: Involves multiple intermediary layers, such as security and business logic, to manage complex data processing and ensure security.

Pros:

- **Centralized Control:** servers handle multiple clients allowing centralized management.
- **Scalability:** Allow developers to add more clients/servers
- **Data Security:** servers can be secured with layers, protecting user data

Cons:

- **Single point of Failure** → if server goe down, client lose access
- **Network dependency** → poor connectivity lead performance issue
- **Resource Intensive** → increase cost, bez require resources to manage clients

MVC Architecture

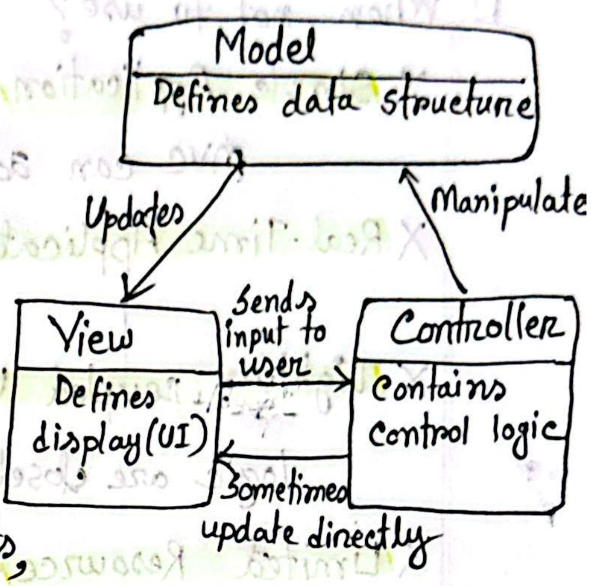
Separates application into 3 main components –

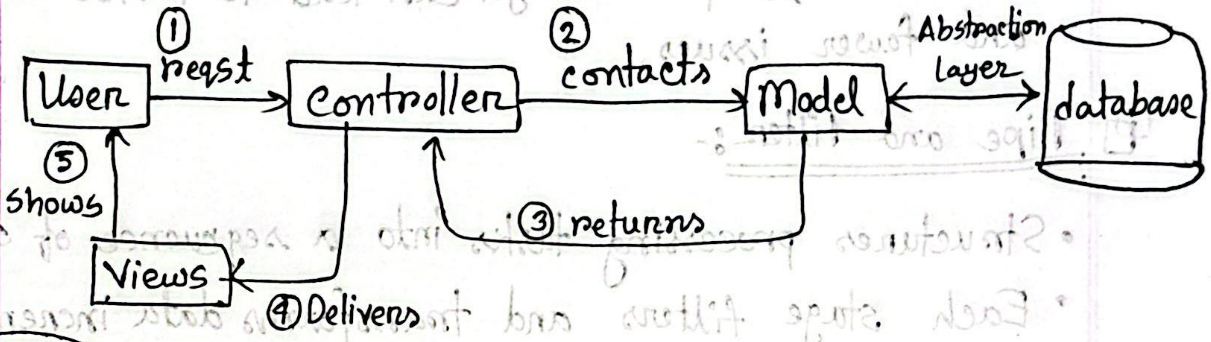
① Model

② View

③ Controller

make easy to maintain code, allows reusability of components, promotes modular approach.





Manages data and business rules ← The Model: defines what data the app should contain. If state of data changes, model will notify the view or sometimes the controller.

UI ← The View: Defines how the app's data should be displayed

Handles user input and coordinate model, View ← The controller: contains logic that updates the model and / or view response to input from the user of the app.

□ When to use?

✓ Complex Application → many features and UI interaction (e-commerce). Help to manage complexity

✓ Frequent UI changes → allows change to View without affecting logi.

✓ Reusability of components → for reuse parts of app, use MVC's modular structure to make it easy.

✓ Testing Requirements → supports through testing, allow to test each component separatedly.

□ When not to use?

X Simple Application → For small apps with limited functiona
MVC can add unneccessary complexity.

X Real-Time Application → for immediate updates, MVC may not work well. (online games)

X Tightly coupled UI and logic → If UI and business logic are closely linked, MVC complicate things further.

X Limited Resource → For small teams or unfamiliar with MVC, simpler design can lead to faster development and fewer issues.

□ <u>Pipe and filter:-</u>

• Structures processing tasks into a sequence of stages/pipe
• Each stage filters and transforms data incrementally
• Enable filters to operate independently, improving scalability and reusability.
• Each filter is tasked with ~~separate~~ specific operations, (such as validating/formatting data) and passes its output to the next stage via interconnected pipes.
• Pipe and filter architecture ensures flexibility and ease of maintainance by isolating concerns regarding data analysis, allowing components to be reused accross various systems.

• Pipes → Channels for data flow
• Filters → Independent processing components / Data processor

Event-Driven Architecture (EDA):