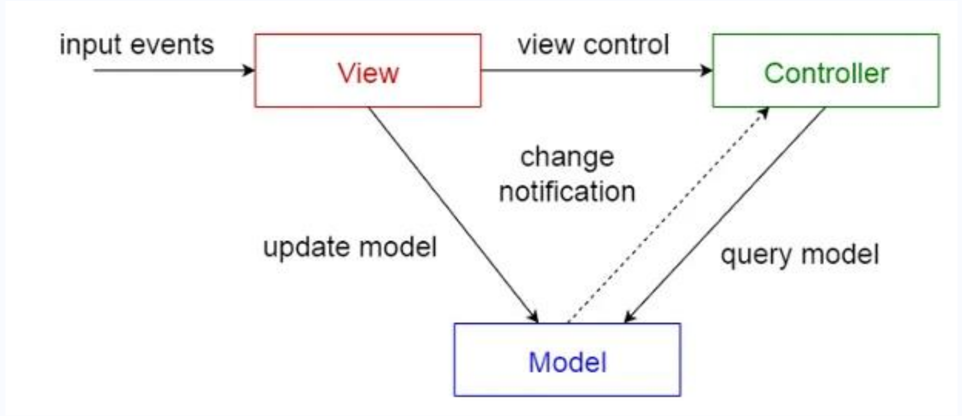
Model-view-controller pattern

- The model-view-controller (MVC) pattern divides an application into three components: A model, a view, and a controller.
  - The model, which is the central component of the pattern, contains the application data and core functionality for processing the data.
  - The view displays application data and interacts with the user. It can access data in the model.
  - The controller handles the input from the user and mediates between the model and the view. It listens to external inputs from the view or from a user and creates appropriate outputs.
  - The controller interacts with the model by calling a method on it to generate appropriate responses.

MVC – Communication between the Components

- **User Interaction with View:** The user interacts with the View, such as clicking a button or entering text into a form.
- **View Receives User Input:** The View receives the user input and forwards it to the Controller.
- **Controller Processes User Input:** The Controller receives the user input from the View. It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.
- **Controller Updates Model:** The Controller updates the Model based on the user input or application logic.
- **Model Notifies View of Changes:** If the Model changes, it notifies the View.
- **View Requests Data from Model:** The View requests data from the Model to update its display.
- **Controller Updates View:** The Controller updates the View based on the changes in the Model or in response to user input.
- **View Renders Updated UI:** The View renders the updated UI based on the changes made by the Controller.

MVC – Communication between the Components



- **User Interaction with View:** The user interacts with the View, such as clicking a button or entering text into a form.
- **View Receives User Input:** The View receives the user input and forwards it to the Controller.
- **Controller Processes User Input:** The Controller receives the user input from the View. It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.
- **Controller Updates Model:** The Controller updates the Model based on the user input or application logic.
- **Model Notifies View of Changes:** If the Model changes, it notifies the View.
- **View Requests Data from Model:** The View requests data from the Model to update its display.
- **Controller Updates View:** The Controller updates the View based on the changes in the Model or in response to user input.
- **View Renders Updated UI:** The View renders the updated UI based on the changes made by the Controller.

When to Use the MVC Design Pattern

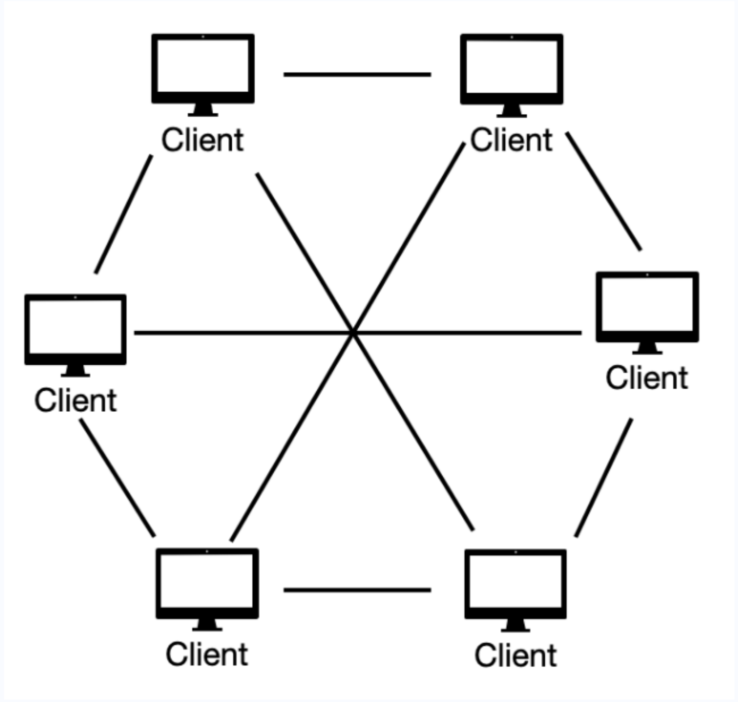
- Complex Applications: Use MVC for apps with many features and user interactions, like e-commerce sites. It helps organize code and manage complexity.
- Frequent UI Changes: If the UI needs regular updates, MVC allows changes to the View without affecting the underlying logic.
- Reusability of Components: If you want to reuse parts of your app in other projects, MVC's modular structure makes this easier.
- Testing Requirements: MVC supports thorough testing, allowing you to test each component separately for better quality control.

When Not to Use the MVC Design Pattern

- Simple Applications: For small apps with limited functionality, MVC can add unnecessary complexity. A simpler approach may be better.
- Real-Time Applications: MVC may not work well for apps that require immediate updates, like online games or chat apps.
- Tightly Coupled UI and Logic: If the UI and business logic are closely linked, MVC might complicate things further.
- Limited Resources: For small teams or those unfamiliar with MVC, simpler designs can lead to faster development and fewer issues.

Peer-to-Peer Architecture

- Peer-to-peer (P2P) architecture, in the context of networking, refers to a decentralised model where each node, or peer, in the network acts both as a client and server.
- This design allows the direct sharing of resources, be it bandwidth, storage, or processing power, creating cooperative networking and efficient resource discovery without relying on a central server.
- P2P technology has gained prominence over the years because of its benefits like flexibility, resilience, and scalability.



Advantages of P2P Architecture

- Scalability: one of the most significant benefits of P2P networks is their scalability.
- Efficiency: P2P networks have an edge over traditional client-server systems as well in terms of efficiency.
- Resilience: the decentralised nature of P2P networks allows them to remain functional even when one or more nodes fail.
- Flexibility: They allow devices to join and leave the network at will without disrupting overall operation.

Disadvantages of P2P Architecture

- Security: one of the primary concerns is security.
  - The lack of a central authority for managing security policies can make these networks vulnerable to various threats, including malware, illegal content sharing, and data breaches.
- Weak Content regulation: in P2P networks, there's no centralised control to prevent the sharing of copyrighted or illegal material.
  - This lack of regulation can lead to legal and ethical issues.
- Quality of Service (QoS) can also be a significant challenge for P2P systems.
  - Since each node operates autonomously, maintaining a constant quality or standard of service can be tricky.
  - For instance, the transfer speed can vary based on the peers' connection quality involved in a file-sharing process.