**Model-View-Controller (MVC)  
  
Model-View-Controller (MVC)** is a design pattern that separates an application into three interconnected components. This separation helps manage complex applications by organizing code and improving modularity, maintainability, and scalability.

**1. Model**

* Represents the **data** and business logic of the application.
* It communicates with the database or external APIs to fetch, update, and manipulate data.
* It notifies the **View** when data changes.

**Responsibilities:**

* Fetching and storing data.
* Defining business rules and logic.
* Managing state.

**Example:**  
In a user management system, the User model handles data related to users, such as saving a new user or retrieving user details.

**2. View**

* Responsible for the **user interface** (UI) and displaying data to the user.
* It listens for updates from the **Model** and updates the UI accordingly.
* The **View** is typically passive, meaning it doesn’t contain logic beyond UI updates.

**Responsibilities:**

* Displaying data to the user.
* Receiving user input and passing it to the **Controller**.

**Example:**  
An HTML page or a React component that displays user information and provides forms for user input.

**3. Controller**

* Acts as a **mediator** between the **Model** and the **View**.
* It handles user input, interacts with the **Model**, and determines which **View** to display.
* The **Controller** processes data before passing it to the **View**.

**Responsibilities:**

* Handling user requests.
* Invoking methods on the **Model** to retrieve or update data.
* Selecting the appropriate **View** to render.

**Example:**  
A controller in a web application handles requests to create, read, update, or delete users and then updates the view accordingly.

**Flow of MVC:**

1. The **user interacts** with the **View** (e.g., clicking a button).
2. The **View** sends the input to the **Controller**.
3. The **Controller** processes the input and interacts with the **Model**.
4. The **Model** updates its data and notifies the **View** of any changes.
5. The **View** refreshes to reflect updated data.

**Model-View-Controller (MVC)** is a design pattern that separates an application into three interconnected components. This separation helps manage complex applications by organizing code and improving modularity, maintainability, and scalability.

**1. Model**

* Represents the **data** and business logic of the application.
* It communicates with the database or external APIs to fetch, update, and manipulate data.
* It notifies the **View** when data changes.

**Responsibilities:**

* Fetching and storing data.
* Defining business rules and logic.
* Managing state.

**Example:**  
In a user management system, the User model handles data related to users, such as saving a new user or retrieving user details.

**2. View**

* Responsible for the **user interface** (UI) and displaying data to the user.
* It listens for updates from the **Model** and updates the UI accordingly.
* The **View** is typically passive, meaning it doesn’t contain logic beyond UI updates.

**Responsibilities:**

* Displaying data to the user.
* Receiving user input and passing it to the **Controller**.

**Example:**  
An HTML page or a React component that displays user information and provides forms for user input.

**3. Controller**

* Acts as a **mediator** between the **Model** and the **View**.
* It handles user input, interacts with the **Model**, and determines which **View** to display.
* The **Controller** processes data before passing it to the **View**.

**Responsibilities:**

* Handling user requests.
* Invoking methods on the **Model** to retrieve or update data.
* Selecting the appropriate **View** to render.

**Example:**  
A controller in a web application handles requests to create, read, update, or delete users and then updates the view accordingly.

**Flow of MVC:**

1. The **user interacts** with the **View** (e.g., clicking a button).
2. The **View** sends the input to the **Controller**.
3. The **Controller** processes the input and interacts with the **Model**.
4. The **Model** updates its data and notifies the **View** of any changes.
5. The **View** refreshes to reflect updated data.

**Advantages of MVC:**

* **Separation of concerns:** Each component has a specific role, making code more organized and easier to maintain.
* **Scalability:** Modular design allows easier scaling and adding new features.
* **Reusability:** Components can be reused in different parts of the application.

Controller -> Service -> Repository  
  
  
The **Controller → Service → Repository** pattern builds on the MVC concept by introducing a more layered architecture, which is especially useful in backend development. Each layer has a distinct role, improving separation of concerns and maintainability.

**1. Controller Layer:**

* The **Controller** handles HTTP requests and responses.
* It validates and processes incoming requests, then delegates the business logic to the **Service** layer.
* The controller typically does not contain business logic.

**Responsibilities:**

* Handle routing and HTTP requests (GET, POST, PUT, DELETE).
* Return appropriate HTTP responses (e.g., JSON or HTML).

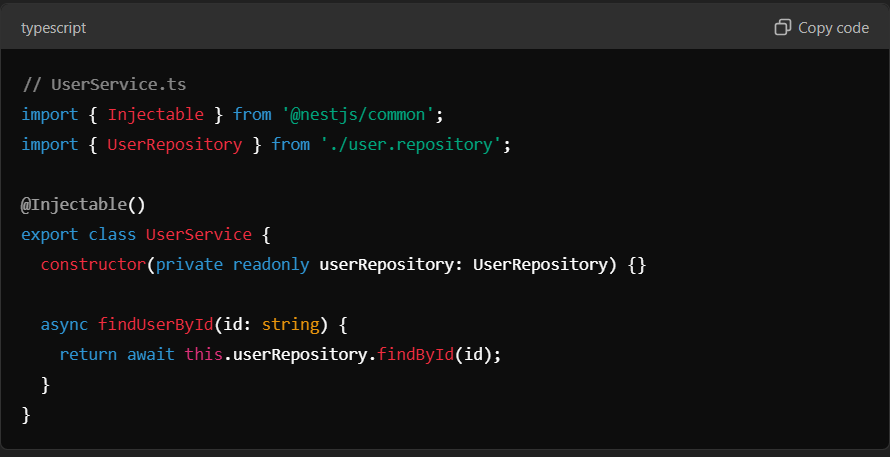
**Example:**

  
  
  
  
**2. Service Layer:**

* The **Service** layer contains the business logic of the application.
* It performs complex operations, applies rules, and interacts with multiple repositories if needed.
* This layer is reusable and modular.

**Responsibilities:**

* Implement business rules and logic.
* Coordinate data between controllers and repositories.

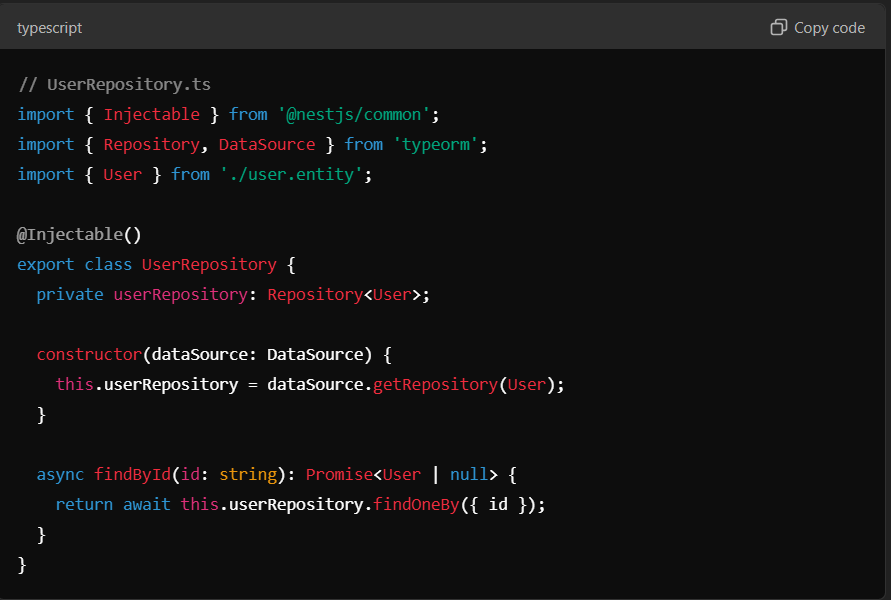


**3. Repository Layer:**

* The **Repository** layer handles data access and communication with the database.
* It abstracts database operations, making it easier to switch data sources if needed.
* Repositories typically use Object-Relational Mapping (ORM) tools like TypeORM, Sequelize, etc.

**Responsibilities:**

* Perform CRUD (Create, Read, Update, Delete) operations.
* Manage database queries and transactions.



**Flow of Execution:**

1. **Client** sends a request to the **Controller** (e.g., GET /users/:id).
2. The **Controller** delegates the logic to the **Service** layer.
3. The **Service** calls the **Repository** to fetch data from the database.
4. The **Repository** interacts with the database and returns the result to the **Service**.
5. The **Service** processes the result and returns it to the **Controller**.
6. The **Controller** sends the final response to the client.

**Advantages of Controller → Service → Repository Pattern:**

* **Separation of concerns:** Each layer has a specific role, making the codebase modular and easier to maintain.
* **Testability:** Business logic in the service layer can be unit tested independently.
* **Reusability:** The service and repository layers can be reused across different controllers.
* **Flexibility:** Switching databases or data sources is easier by modifying the repository.