System Architecture Overview

This project follows a **Modular MVC Architecture** enhanced with a **Use-Case (Service) Layer**, inspired by the ECB (Entity–Control–Boundary) pattern. This structure provides clean separation of concerns, testability, and future scalability.

1. Architectural Pattern Used

Modular MVC with Use-Case/Service Layer (MVC + ECB Hybrid)

The system is built on top of the classic **Model–View–Controller (MVC)** pattern, with an additional **Use-case/Service layer** to separate business rules from controllers.

This hybrid approach allows:

- Clean routing
- Thin, readable controllers
- Centralized business logic
- Reusable service functions
- Database logic isolated inside models
- Easy scaling into microservices or CQRS in future versions

2. Why This Architecture Is Used

I. **Clear Separation of Responsibilities**

Each layer has one focused job:

- **Model** → Data and database operations
- **View** → Frontend UI (HTML/CSS)
- **Controller** → Handles HTTP requests and responses
- **Service (Use-Case)** → Core application rules
- **Route** → Maps URLs to controller actions

This keeps code clean, readable, and maintainable.

II. Scalable for Future Features

The project may expand to include:

- Tutor ranking algorithm
- Real-time chat
- Notifications
- Admin panel
- Payment integration

The MVC + Use-case separation ensures the system remains stable as these features are added.

III. Easy Unit Testing

The **service layer** allows isolated testing (Jest) without needing controllers or HTTP calls.

IV. Avoids "Fat Controllers"

All logic goes into services instead of controllers, keeping controllers thin and simple.

3. Architecture Components

Below is a breakdown of each part of the architecture.

3.1 Models (M)

Models represent **data entities** and handle **database interactions**.

In this project, main models include:

- User
- Tuition
- Application

Responsibilities:

- Define data schema
- Validate database fields
- Perform CRUD operations
- No business logic

Example Actions:

- User.create()
- Tuition.findByLocation()

3.2 Views (V)

Views represent the **frontend UI** served to users:

- HTML pages
- CSS for styling
- Client-side JS (if any)

Examples:

- login.html
- tuition-list.html
- post-tuition.html

Views in this project are served as **static assets** via Express.

3.3 Controllers (C)

Controllers handle **incoming HTTP requests** and **send responses**.

They do not contain business logic.

Responsibilities:

- Receive route request
- Validate input format

- Call service (use-case) function
- Send JSON/HTML response

Example:

```
async function createTuition(req, res) {
  const data = req.body;
  const result = await tuitionService.postTuition(data);
  res.json(result);
}
```

3.4 Use-Case / Service Layer (ECB "Control")

This is the core intelligence of the system.

Each feature has its own **service function**, which performs all rules and conditions.

Examples:

- postTuition()
- applyForTuition()
- shortlistTutor()
- hireTutor()
- filterTuitions()

Responsibilities:

- Enforce business rules
- Coordinate models
- Handle validation
- Throw errors for invalid actions
- Ensure consistency

Benefits:

- Makes controllers thin
- Reusable logic
- Easy to test with Jest

3.5 Routes

Routes connect **URLs** to **controller functions**.

Example:

POST /tuitions → createTuition()

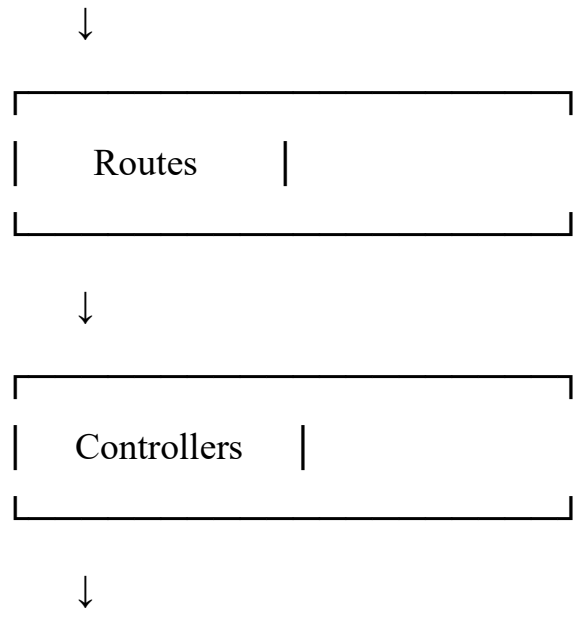
GET /tuitions → listTuitions()

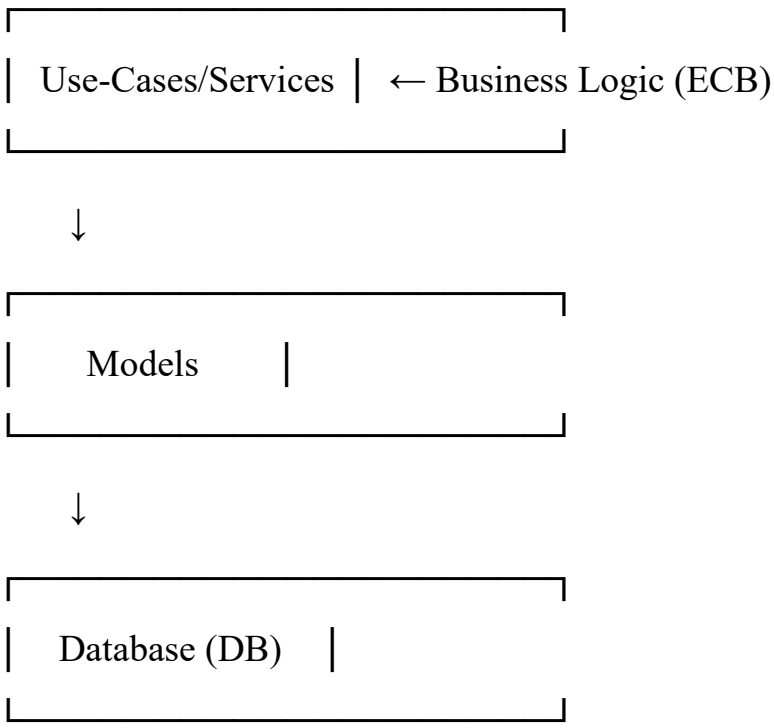
POST /apply → applyForTuition()

Responsibilities:

- Define endpoints
- Map endpoints to controller functions

4. High-Level Architecture Diagram





5. Testing Strategy

Unit Testing Tool: Jest

Why Jest?

- Simple and fast
- Built-in mocks
- No extra setup required
- Perfect for testing service layer functions
- Large community support

Example Test File:

- tests/tuitionService.test.js
- tests/authService.test.js

6. Folder Structure

src/

├── models/

├── views/ (public/)

├── controllers/

├── routes/

├── services/

├── middlewares/

├── validators/

└── utils/

tests/

public/

7. Why This Architecture Is Best for the Project

- Works perfectly with Node.js, Express, HTML, CSS
- Easy to onboard new developers
- Extensible for new features
- Supports clean unit testing
- Prevents code duplication
- Prevents controller bloat
- Maintains long-term stability