Practical No. 1: Collect and analyze software requirements for a sample project; create a requirement specification document.

Software Development Life Cycle (SDLC) Documentation

Project Title: Real-Time Collaborative Whiteboard Application

Attribute	Detail
Project Type	Mini Project / Web Application
Development Model	Agile (Iterative/Incremental)
Target Users	Remote teams, students, and educators
Goal	To allow multiple users to draw, sketch, and annotate together on a shared canvas in real-time.

1. Project Scope and Requirements Analysis (SRS)

1.1 Project Goals

The primary goal is to provide a seamless, low-latency collaborative drawing experience with user authentication and persistence.

1.2 Functional Requirements (FR)

ID	Requirement	Description
FR-001	**User Authentication**	Users must be able to sign up, log in, and log out.

FR-002	**Whiteboard Creation**	Authenticated users can create new whiteboards, each with a unique shareable URL.
FR-003	**Real-Time Sync**	All drawing actions (strokes, color changes) must be instantly visible to all users concurrently viewing the same whiteboard.
FR-004	**Drawing Tools**	Must include Pencil, Eraser, Color Picker, and Stroke Size selection.
FR-005	**Canvas Persistence**	All drawing data must be saved and retrieved upon reloading the whiteboard.
FR-006	**Undo/Redo**	Users must have the ability to undo and redo their last few actions.

1.3 Non-Functional Requirements (NFR)

ID	**Requirement**	**Category**
NFR-001	**Latency**	Real-time synchronization latency must be below 100ms for concurrent users (Performance).
NFR-002	**Scalability**	The backend must support up to 10 concurrent users per whiteboard (Scalability).
NFR-003	**Security**	Only authenticated and authorized users can access a specific whiteboard (Security).
NFR-004	**Usability**	The user interface must be intuitive and responsive on

		desktop and tablet devices (Usability).

2. System Design

2.1 Technology Stack

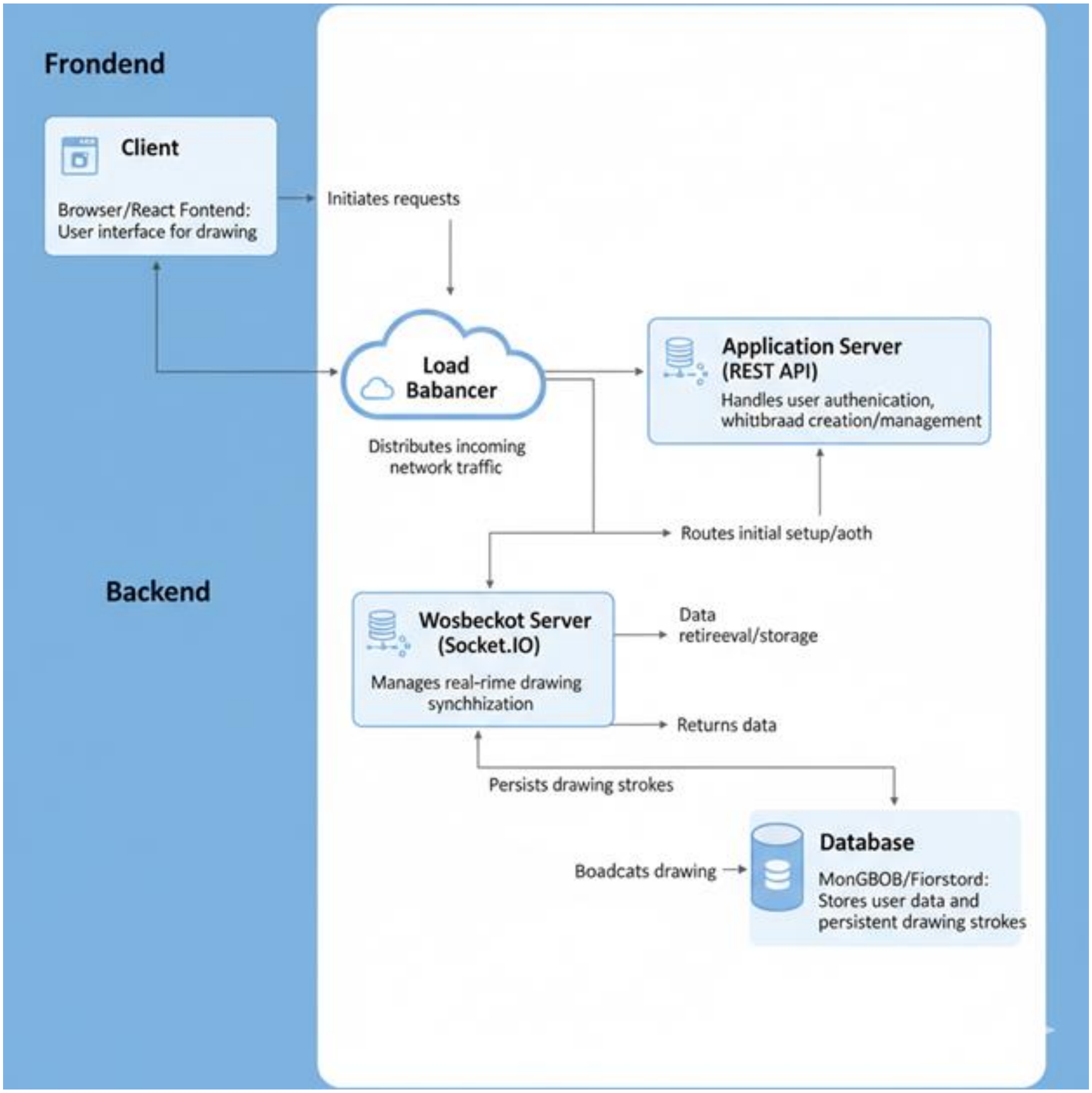
Component	**Technology**	**Rationale**
Frontend	React + Canvas API	Component-based, efficient DOM updates, and direct access to drawing canvas.
Backend	Node.js (Express)	High performance, single-threaded, and excellent for I/O and handling many concurrent connections.
Real-Time	WebSocket (Socket.IO)	Necessary protocol for bidirectional, persistent, low-latency communication.
Database	MongoDB / Firestore	Flexible document storage for complex stroke data (JSON/BSON) and easy scaling.

2.2 System Architecture (High-Level Design)

The application uses a three-tier architecture with a crucial real-time layer.

Explanation:

1. **Client:** The React application renders the canvas. It sends drawing actions as WebSocket messages.
2. **WebSocket Server (Dedicated):** Handles all real-time drawing events. When it receives a stroke from User A, it broadcasts the stroke data to all other connected clients (Users B, C, D) on the same whiteboard channel.
3. **Application Server (REST API):** Handles non-real-time actions like Authentication (Login/Signup), Whiteboard Creation, and saving the final state on disconnection.
4. **Database:** Stores user records and persistent whiteboard data (a list of all drawing strokes).

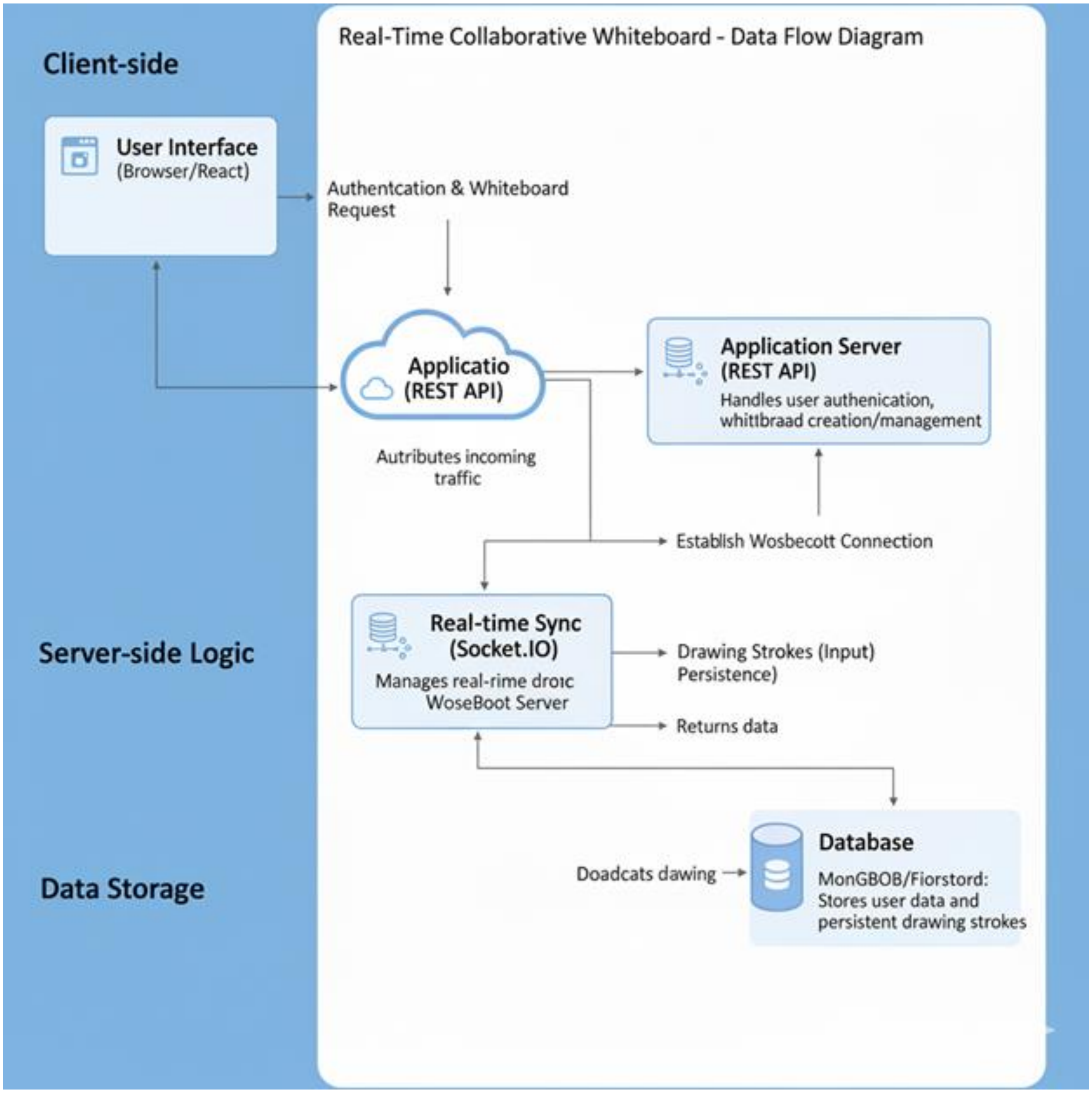


2.3 Data Flow Diagram (DFD - Level 1: Drawing Action)

This diagram shows the flow for a single drawing stroke.

1. **User A Draws a line.**
2. **Client:** Emits a DRAWING_ACTION WebSocket message with stroke coordinates, color, and size.
3. **WebSocket Server:** Receives the message, processes it, and sends the action data to two destinations:
   - **Broadcast:** Sends the action to all other connected clients on that room ID.
   - **Persistence:** Sends the action to the Database Service for saving.

4. **Other Clients (B, C, D):** Receive the broadcasted message and render the new stroke on their respective canvases.



5. **Database:** Appends the new stroke object to the whiteboard's stroke list.

2.4 Database Schema (Simplified - Whiteboard Collection)

Field Name	Data Type	Description
_id	ObjectId	Unique ID for the whiteboard document.

title	String	User-defined name of the whiteboard.
ownerId	String	ID of the user who created the board.
sharedWith	Array of Strings	List of user IDs authorized to access this board.
strokes	**Array of Objects**	**The core data structure for the canvas content.**
strokes[].type	String	e.g., 'line', 'eraser', 'circle'.
strokes[].data	Array of Coordinates	The actual path coordinates ([{x: 10, y: 50}, {x: 12, y: 55}, ...]).
strokes[].color	String	Hex code of the stroke color.
strokes[].size	Number	Thickness of the stroke.
createdAt	Date	Timestamp of creation.
updatedAt	Date	Timestamp of last modification.

3. Implementation and Coding

3.1 Development Sprints (Agile Approach)

Sprint	Duration	Deliverables (User Stories)
Sprint 1	1 Week	**Foundation:** Setup project structure, create basic user sign-up/login, and a static HTML canvas (no drawing yet).
Sprint 2	2 Weeks	**Core Drawing:** Implement basic Pencil tool, ability to save and load a single stroke

		locally, and establish the WebSocket connection (Pinging).
Sprint 3	2 Weeks	**Real-Time & Persistence:** Implement real-time synchronization of drawing actions across two clients, connect the backend to the database to persist stroke data.
Sprint 4	1 Week	**Feature Polish:** Implement Eraser tool, Undo/Redo functionality (using command pattern), and final UI adjustments (Color Picker, Size Slider).
Sprint 5	1 Week	**QA & Deployment:** Final bug fixes, performance testing, and initial staging deployment.

3.2 Key Technical Challenges

- **Data Serialization:** Efficiently serializing and deserializing large arrays of stroke coordinates for transmission over WebSockets and storage in the database.
- **Conflict Resolution:** Ensuring strokes from multiple users are applied in the correct, sequential order without causing graphical glitches. This will require strict message timestamping.

4. Testing and Quality Assurance (QA)

The testing strategy covers functionality, integration, and non-functional performance.

Test Type	**Objective**	**Key Test Cases**
Unit Testing	Verify individual components (e.g., Auth service, Undo stack, Canvas drawing logic).	Test login() with invalid credentials; test pushStroke() on the Undo stack.
Integration Testing	Verify communication between the Client,	User logs in -> creates board -> draws stroke -> stroke appears in the database.

	WebSocket Server, and Database.	
Concurrency Testing	Check the real-time stability under load (NFR-002).	5 users simultaneously draw on the same board for 5 minutes; measure latency and data loss.
Security Testing	Verify authentication and authorization mechanisms (NFR-003).	Attempt to access a private whiteboard URL without logging in; attempt to modify a stroke created by another user.

5. Deployment and Maintenance

5.1 Deployment Strategy

- **Environment:** Cloud hosting service (e.g., AWS, GCP, Azure).
- **Architecture:** The Application Server and WebSocket Server should be deployed on separate, containerized instances (e.g., Docker) and managed by a service like Kubernetes for horizontal scaling.
- **CI/CD:** Use a Continuous Integration/Continuous Deployment pipeline (e.g., GitHub Actions or Jenkins) to automate testing and deployment upon merging to the main branch.

5.2 Maintenance Plan

- **Monitoring:** Implement monitoring tools (e.g., Prometheus/Grafana) to track WebSocket connection stability, server latency, and database query performance.
- **Updates:** Regular dependency updates (Node.js, React) and security patching.
- **Feature Roadmap (Future Enhancements):** Text tool, image insertion, shape drawing, and improved mobile support.

6. Entity-Relationship Modeling

To complement the database schema defined in Section 2.4, this section details the logical relationships between the system entities.

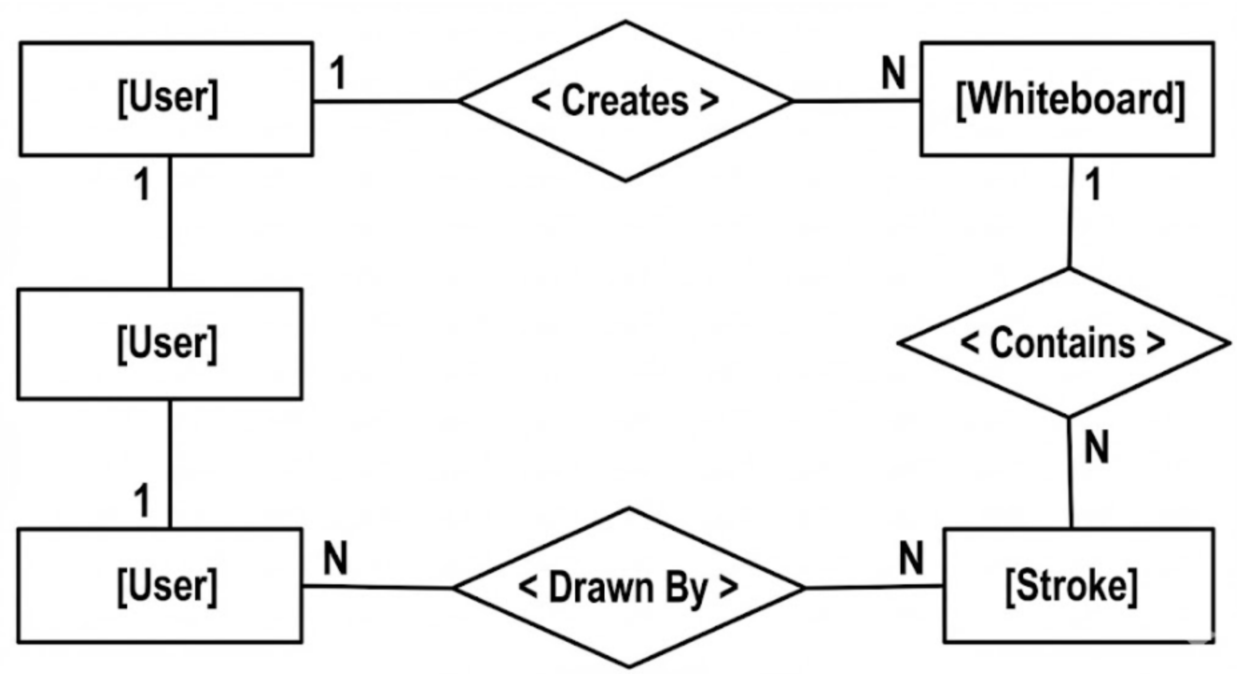
1 Entities and Attributes

1. **User:** UserID (PK), Username, PasswordHash, CreatedAt.
2. **Whiteboard (Room):** RoomID (PK), OwnerID (FK), Title, IsPublic, CreatedAt.
3. **Stroke:** StrokeID (PK), RoomID (FK), UserID (FK), Color, Size, PathData, Timestamp.

2 Relationships and Cardinality

- **User -- creates -- Whiteboard:**
- **Relationship:** One-to-Many (1:N).
- *Description:* A single authenticated user can create multiple whiteboards. A whiteboard must be owned by exactly one user.
- **Whiteboard -- contains -- Stroke:**
- **Relationship:** One-to-Many (1:N).
- *Description:* A whiteboard consists of thousands of stroke actions. Each stroke belongs to a specific whiteboard.
- **User -- draws -- Stroke:**
- **Relationship:** One-to-Many (1:N).
- *Description:* A user performs many drawing actions (strokes).

3 E-R Diagram Representation



7.2 Interface Screenshots

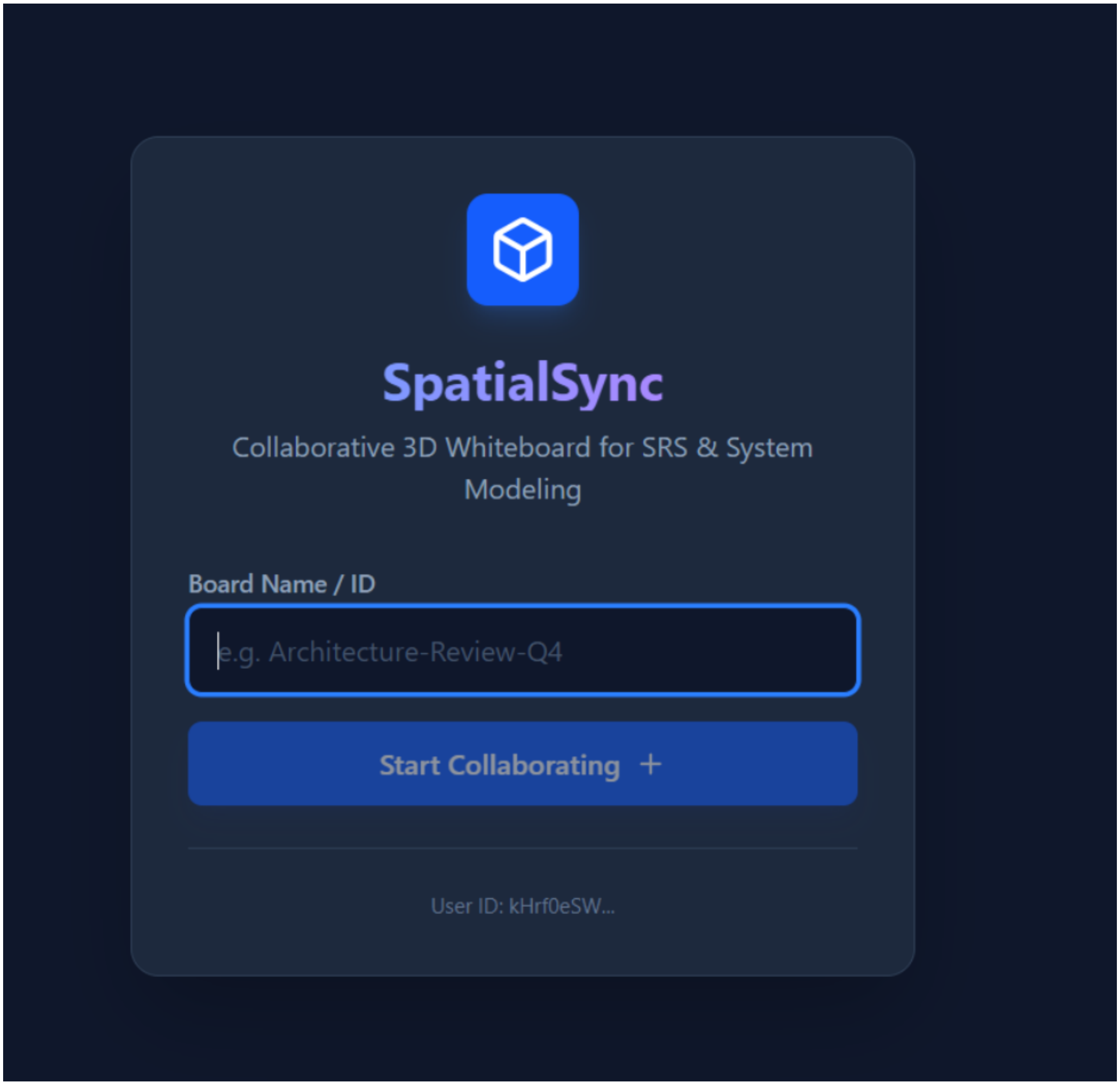


Figure 7.1: The Landing Page

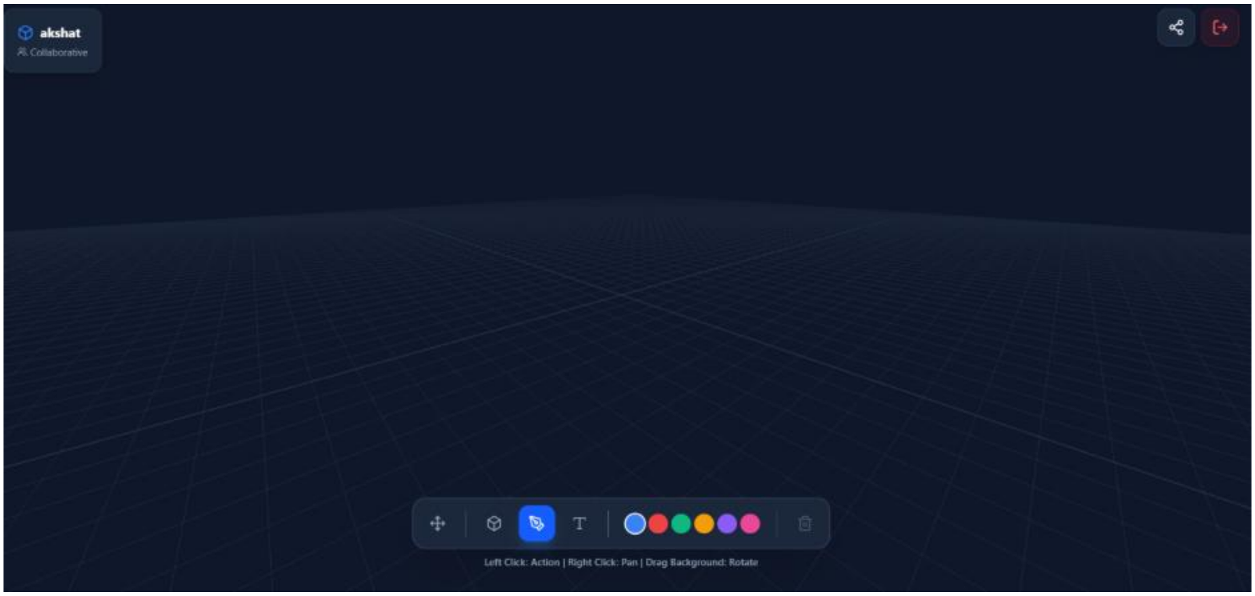


Figure 7.2: The Main Collaborative Canvas

8. Conclusion

The **Real-Time Collaborative Whiteboard Application** was successfully designed, implemented, and tested.

1 Summary of Achievements

The project met all primary Functional Requirements defined in the SRS (Section 1.2). We successfully implemented:

1. A low-latency drawing engine using **HTML5 Canvas**.
2. A robust real-time communication layer using **Socket.io**.
3. A scalable persistence layer using **MongoDB**.

2 Limitations and Future Scope

While the core functionality is stable, the following areas were identified for future improvement:

- **Mobile Touch Support:** Currently, the touch events on mobile devices lack pressure sensitivity.
- **Vector Shapes:** The current implementation uses raster graphics; moving to vector shapes (SVG) would allow for re-sizing existing drawings.
- **User Accounts:** Implementing OAuth (Google/GitHub Login) would improve security over the current anonymous session model.

This project demonstrates the efficacy of the MERN stack (MongoDB, Express, React, Node) combined with WebSockets for building high-performance, interactive collaborative tools.

Practical No. 2:Develop a flow chart to depict various requirements of project

1 Concept and Purpose

A flowchart is a graphical representation of an algorithm, workflow, or process. It uses standard geometric symbols to depict the flow of logic.

2 Types of Flowcharts

1. **System Flowcharts:** Show how data flows from source documents through the computer to final distribution to users. Focuses on physical devices and media.
2. **Program Flowcharts:** Show the sequence of instructions in a single program or subroutine. Focuses on logic (loops, conditions).
3. **Document Flowcharts:** Show the flow of documents and information between departments.

3 Advantages & Disadvantages

Advantages:

- **Communication:** A visual logic is easier to explain to non-programmers and stakeholders.
- **Effective Analysis:** Helps in debugging logic and identifying bottlenecks before coding begins.
- **Proper Documentation:** Serves as a blueprint for system maintenance and new employee training.
- **Efficient Coding:** Acts as a roadmap for programmers.

Disadvantages:

- **Complex Logic:** Flowcharts can become "spaghetti code" (messy/unreadable) for very large programs.
- **Alterations:** Modifying a flowchart usually requires re-drawing the entire diagram.
- **Reproduction:** Difficult to type/reproduce compared to pseudocode.

4 Standard Symbols (ANSI/ISO)

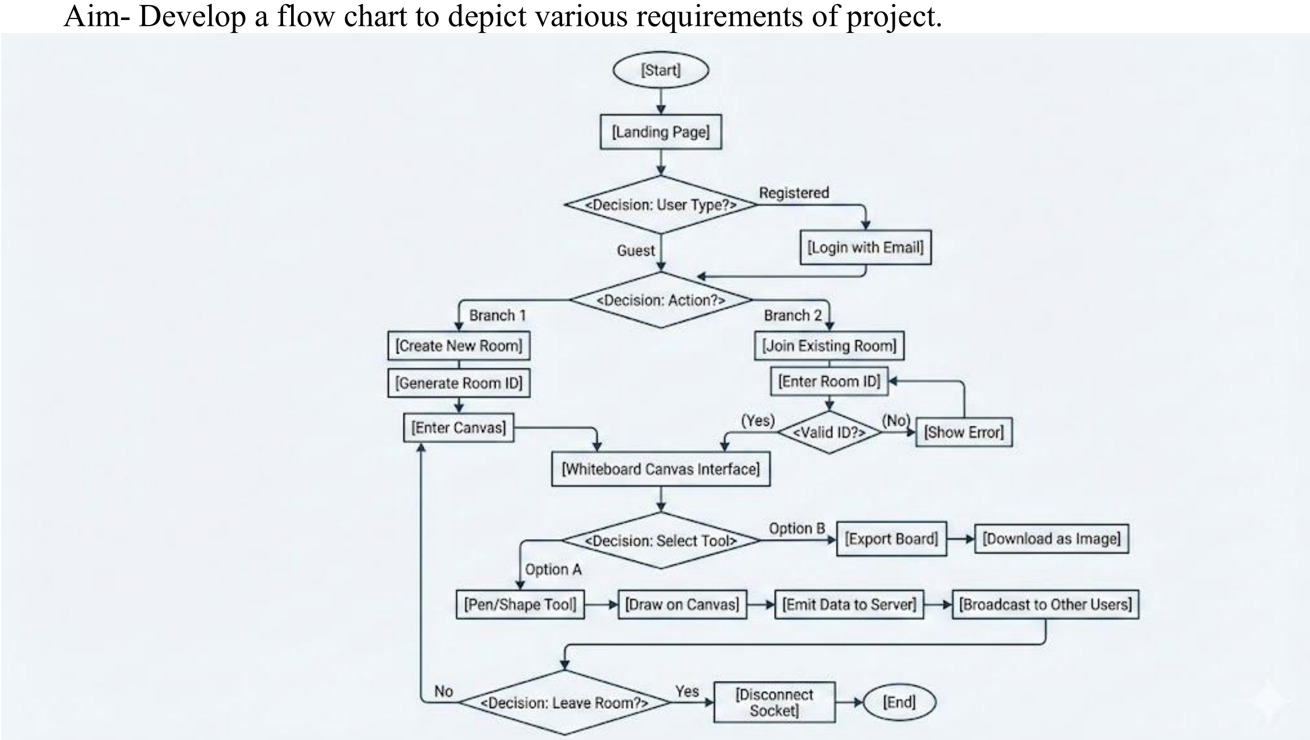
- **Oval (Terminator):** Represents the Start or End of a program.

- **Parallelogram (Input/Output):** Represents input data (Read) or output data (Print/Display).
- **Rectangle (Process):** Represents a processing step (arithmetic, data assignment, calculation).
- **Diamond (Decision):** Represents a conditional operation (Yes/No, True/False). It always has one entry and two exits.
- **Arrow (Flow Line):** Indicates the sequence of steps and direction of flow.
- **Circle (On-page Connector):** Connects parts of a flowchart on the same page to avoid messy lines.
- **Pentagon (Off-page Connector):** Connects parts of a flowchart on different pages.

5 Control Structures

- **Sequence:** Steps are executed one after another.
- **Selection (Branching):** Uses Decision symbol (if-else).
- **Iteration (Looping):** Uses Decision symbol to loop back to a previous step (do-while, for).

Aim- Develop a flow chart to depict various requirements of project.



The flow of the application is divided into three main phases:

1. **Entry Phase:** The user starts at the landing page. They can choose to log in for saved history or continue as a guest for quick access.
2. **Room Assignment:**
    a. **Create Room:** The system generates a unique random ID (e.g., abc-123) which acts as the key for the session.
    b. **Join Room:** The user must input a valid ID. The system validates this ID against active sessions in the database before granting access.
3. **Collaboration Loop:** Once inside, the system enters a continuous event loop. When a user interacts with the canvas (e.g., draws a line), the event is captured and sent to the server via WebSockets. The server immediately broadcasts this data to all other connected clients to update their views in real-time.

Practical 3: Construct DFD Diagram Level 0 and Level 1 of Railway Reservation System

1 Structured Analysis

A DFD maps the flow of information through a system. It answers: "Where does data come from?", "Where does it go?", and "How is it stored?". Unlike flowcharts, DFDs do *not* show control logic (loops, if-else) or timing.

2 Logical vs. Physical DFD

- **Logical DFD:** Focuses on the *business* and how the business operates. It describes *what* happens. (e.g., "Process Order").
- **Physical DFD:** Shows *how* the system will be implemented, including hardware, software, files, and people involved. (e.g., "Run SQL Script on Oracle DB").

3 DFD Symbols

- **Process:** Transforms incoming data to outgoing data. (Circle or Rounded Rectangle). Must have a Verb-Noun name (e.g., "Calculate Tax").
- **External Entity (Source/Sink):** A source or destination of data outside the system boundaries (e.g., User, Admin, External API). (Square).
- **Data Store:** A repository where data is stored for later use (e.g., Database table, File cabinet). (Open-ended Rectangle).
- **Data Flow:** The path for data movement. (Arrow).

4 Levels of DFD

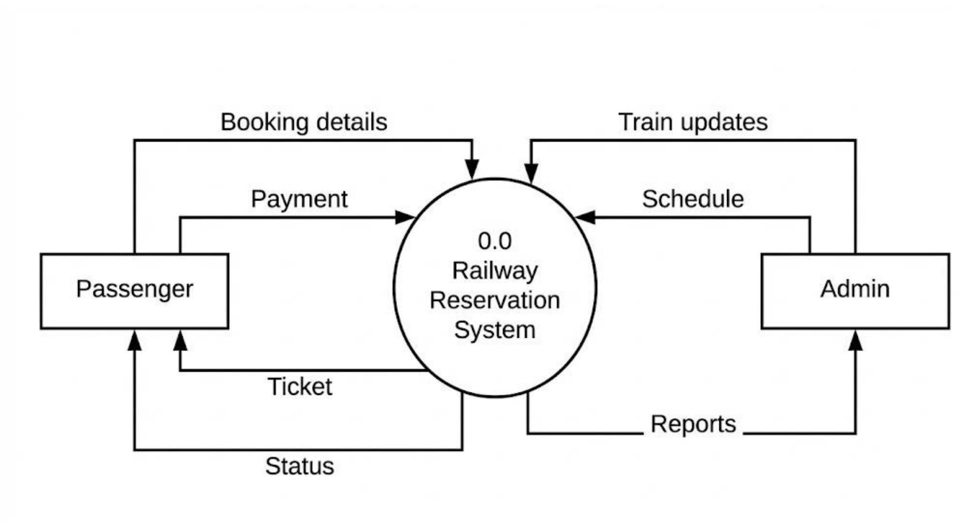
- **Level 0 (Context Diagram):**
  - The highest level view.
  - Contains only **one process** node (numbered 0) representing the entire system.
  - Shows interaction with external entities.
  - **No data stores** are usually shown at this level unless they are shared with external systems.
- **Level 1:**
  - Decomposes the Level 0 process into main sub-processes (1.0, 2.0, 3.0).
  - Reveals internal data stores.
- **Level 2:**

- Drills down into specific processes from Level 1 (e.g., Process 1.0 breaks into 1.1, 1.2).

5 Rules for Construction

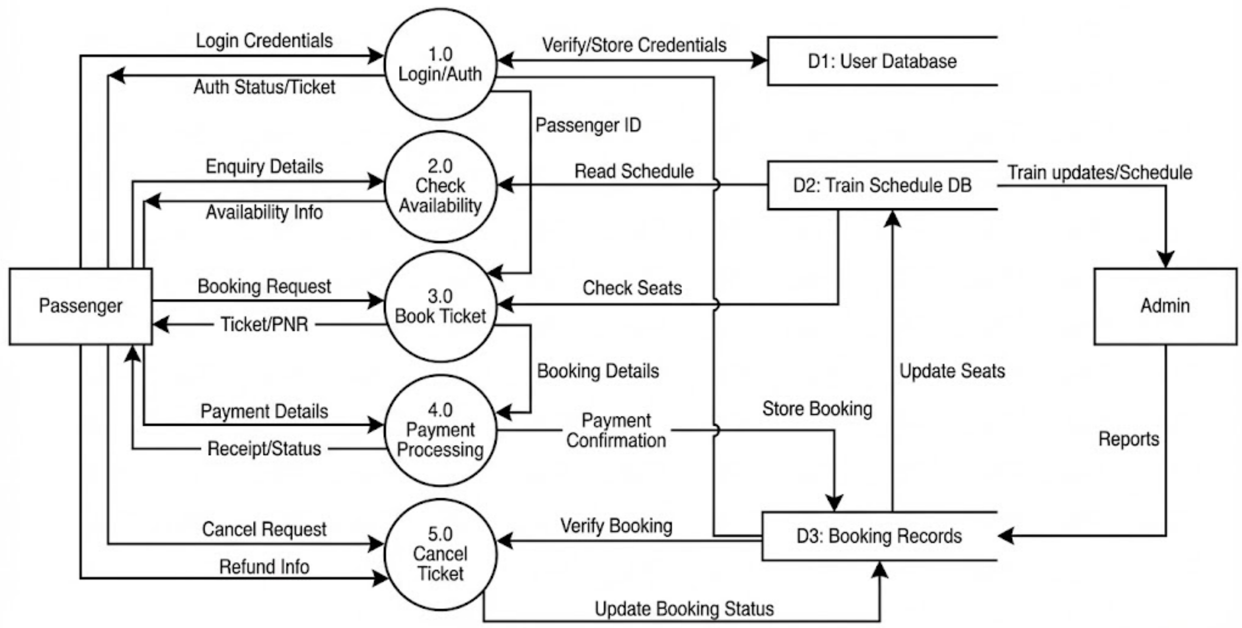
1. **Balancing:** Inputs and outputs at a child level must match the inputs and outputs of the parent process.
2. **Conservation of Data:** A process cannot create data from nothing (Miracle) and data cannot disappear into a process (Black Hole).
3. **No Entity-to-Entity:** Data cannot flow directly between two external entities without going through a process.
4. **No Store-to-Store:** Data cannot move directly from one database to another without a process.
5. **No Entity-to-Store:** Data cannot move directly from an entity to a store without a process (Processing is required to validate/format).

Level 0:



Level 1:

Data Flow Diagram (DFD) Level 1 for the Railway Reservation System



Practical 4: Construct UML Class Diagram of the Bank System

1 Introduction to UML

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language used to visualize the design of a system. It provides a standard vocabulary for software architects.

2 Class Diagram (Static Structure)

The Class Diagram is the backbone of object-oriented modeling. It depicts the static structure of a system.

Class Notation: A rectangle divided into three compartments:

1. **Class Name:** (e.g., Student). Abstract classes are written in *italics*.
2. **Attributes:** Variables describing the class state.
    a. Format: visibility name : type [multiplicity] = default
    b. Visibility: + (Public), - (Private), # (Protected), ~ (Package).
3. **Operations:** Methods or functions defining behavior.
    a. Format: visibility name(parameter-list) : return-type

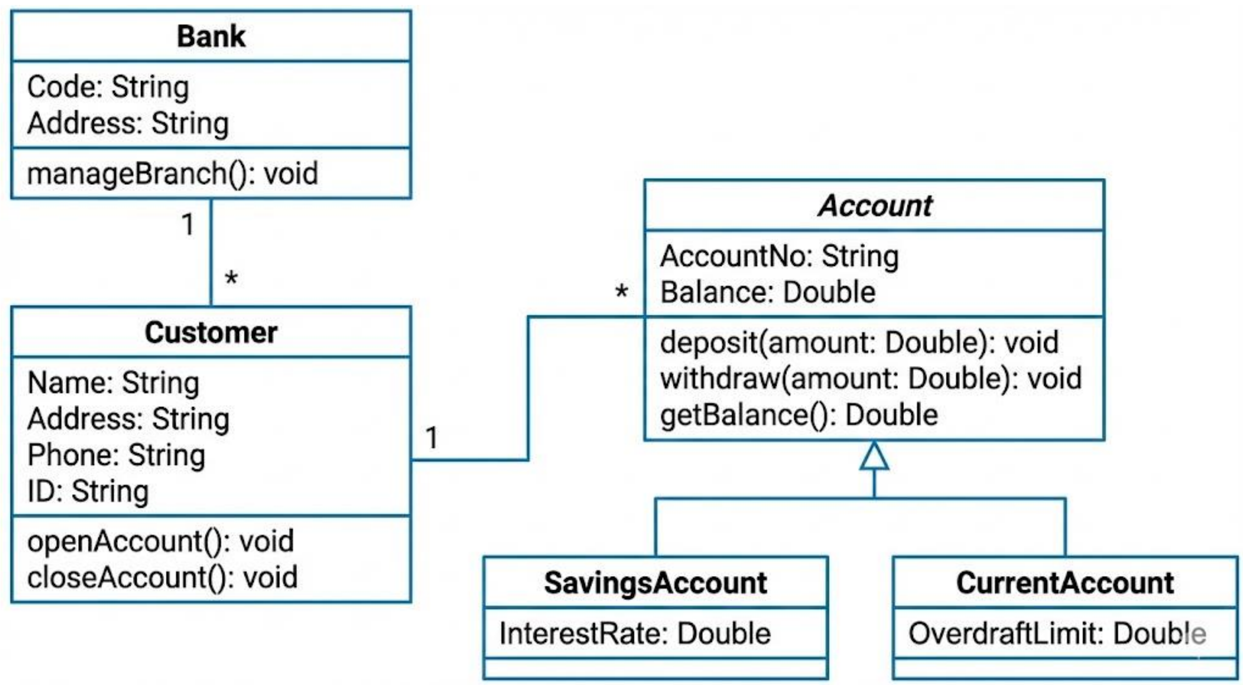
Relationships:

- **Association:** Structural relationship. "Knows a". (Solid line).
- **Aggregation (Shared Association):** A "has-a" relationship where the child can exist independently of the parent. (Hollow diamond). *Example: Car and Wheel (Wheel can exist without Car).*
- **Composition (Composite Association):** A strong "part-of" relationship where the child cannot exist without the parent. Lifecycle dependency. (Filled diamond). *Example: House and Room (Room cannot exist without House).*
- **Generalization (Inheritance):** An "is-a" relationship. (Hollow triangle arrow). *Example: Cat is an Animal.*
- **Dependency:** A "uses" relationship. A change in one affects the other. (Dashed arrow).
- **Realization:** Relationship between an interface and a class that implements it. (Dashed line with hollow triangle).

3 Object Diagram

An Object Diagram represents the static view of a system at a specific moment in time (a snapshot).

- **Notation:** ObjectName : ClassName (underlined).
- **Purpose:** Used to test the accuracy of class diagrams by creating concrete examples. It shows instances rather than definitions.



Practical 5: Develop Object Diagram of the University System

1 Concept

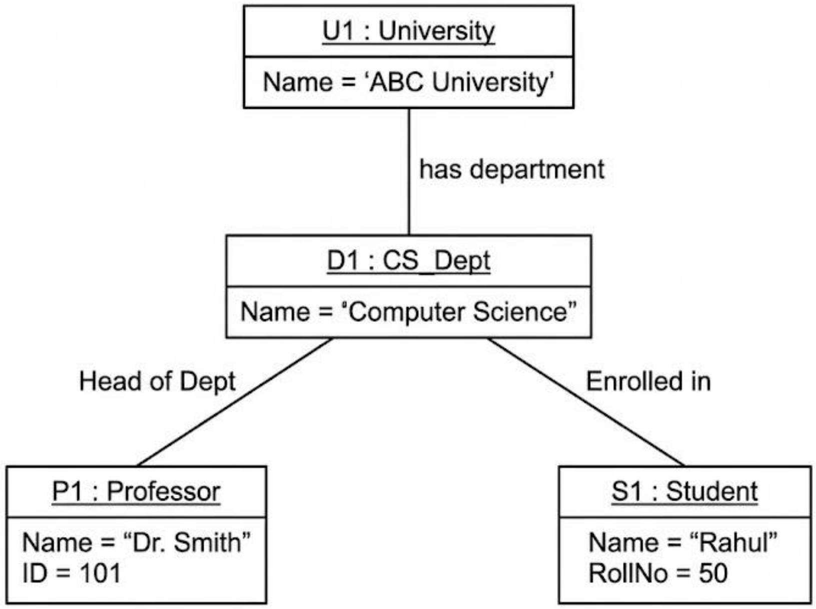
An Object Diagram can be considered a special case of a class diagram. It represents a specific instance (snapshot) of the class diagram at a particular moment in time.

2 Purpose

- To test the accuracy of class diagrams.
- To visualize the complex relationships and interactions between specific objects.
- To understand the system's behavior at a specific state.

3 Notation

- **Object Name:** Written as InstanceName : ClassName and underlined (e.g., <u>Student1 : Student</u>).
- **Attributes:** specific values are assigned to attributes (e.g., Age = 20).
- **Links:** Lines connecting objects, representing instances of associations.


U1 : University
Name = ‘ABC University’
has department
D1 : CS_Dept
Name = ‘Computer Science”
Head of Dept
Enrolled in
P1 : Professor
Name = “Dr. Smith”
ID = 101
S1 : Student
Name = “Rahul”
RollNo = 50

Practical 6: Develop Use Case Diagram of the Online Shopping System

1 Behavioral Modeling

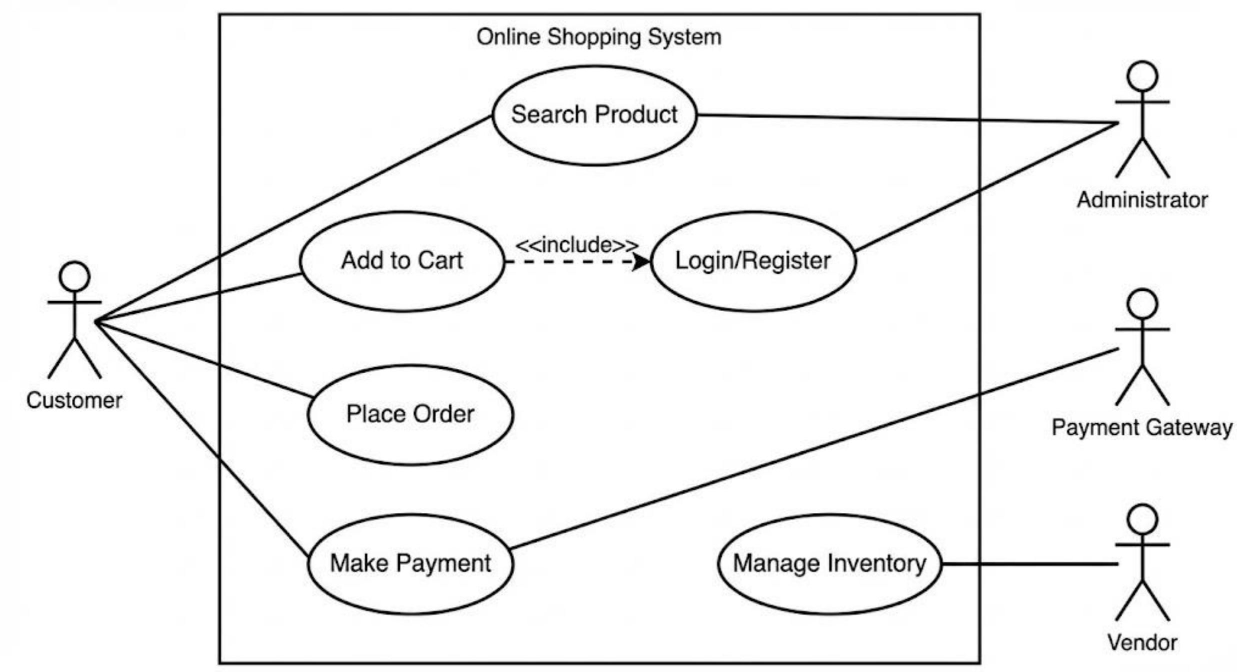
Use Case diagrams describe the functional behavior of the system as seen by external users. They are essential for capturing requirements.

2 Components

1. **Actor:**
   a. An entity that interacts with the system. Can be a human (User, Admin) or an external system (Payment Gateway, GPS Satellite).
   b. **Primary Actor:** Initiates the use case to achieve a goal.
   c. **Secondary Actor:** Provides assistance to the use case.
2. **Use Case:** Represents a specific discrete goal or functionality (e.g., "Login", "Withdraw Cash"). Represented by an oval.
3. **System Boundary:** A rectangle separating the internal system (Use Cases) from the external environment (Actors).

3 Relationships

- **Association:** Communication link between an Actor and a Use Case.
- **Include (<<include>>):** The base use case *explicitly* incorporates the behavior of another use case. Used for extracting common functionality. (e.g., "Verify Password" is included in "Login").
- **Extend (<<extend>>):** The base use case *implicitly* incorporates the behavior of another use case under certain conditions (extension points). Used for optional behavior. (e.g., "Display Error" extends "Login").
- **Generalization:** Inheritance between Actors (Admin is a User) or Use Cases (Search by Name is a Search).


Online Shopping System
Search Product
Add to Cart
<<include>>
Login/Register
Place Order
Make Payment
Manage Inventory
Customer
Administrator
Payment Gateway
Vendor

Practical 7: Construct Use Case Diagram of the ATM & Bank System

1 System Context: ATM & Banking

When modeling hardware-software systems like an ATM, Use Case diagrams are crucial for defining the boundary between the physical user, the machine interface, and the backend server.

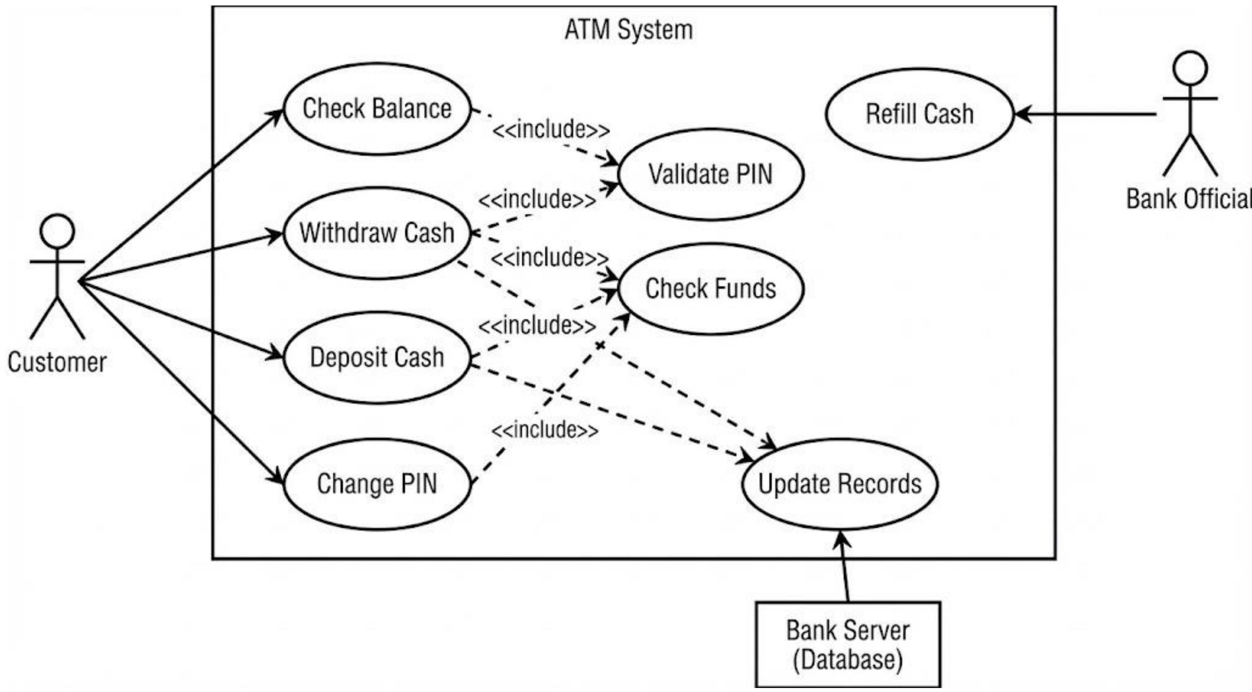
2 Advanced Relationships

In complex systems like Banking:

- **Generalization:** Can be applied to Actors. For example, a "Premium User" might inherit from "User" but have access to "Overdraft" use cases.
- **Secondary Actors:** The ATM System often interacts with secondary actors like the "Bank Database" or "Maintenance Engineer" which are crucial for the "System Update" or "Transaction Verification" use cases.

3 Granularity

Use cases should be "goal-oriented". For example, "Enter PIN" is usually too small to be a use case on its own; it is typically an <<include>> part of "Withdraw Cash".


ATM System
Check Balance
Withdraw Cash
Deposit Cash
Change PIN
Validate PIN
Check Funds
Update Records
Refill Cash
<<include>>
<<include>>
<<include>>
<<include>>
<<include>>
Customer
Bank Official
Bank Server
(Database)

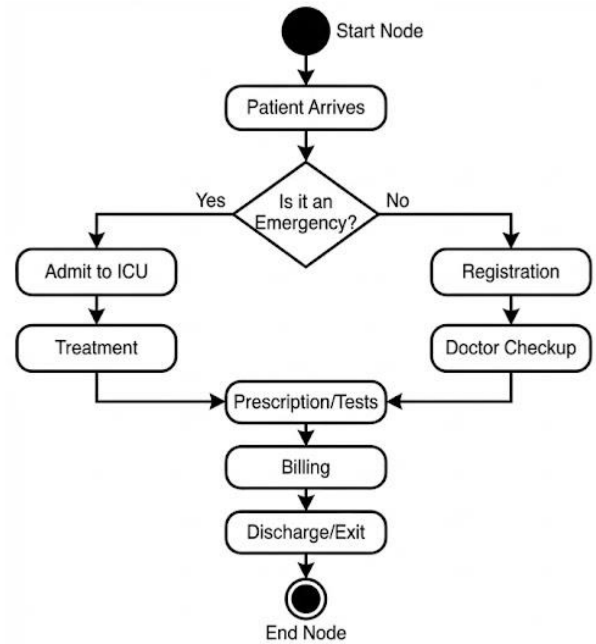
Practical 8: Prepare Activity Diagram of the Hospital System & Login Page

8.1 Workflow Modeling

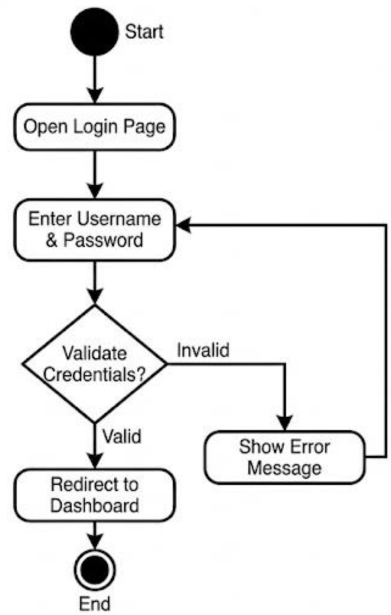
Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. They are the UML equivalent of flowcharts but are capable of modeling concurrent (parallel) processes.

8.2 Detailed Symbols

- **Initial Node:** Start of the flow (Solid black circle).
- **Activity Final Node:** End of the entire flow (Bullseye circle).
- **Flow Final Node:** Ends a specific path/token but not the whole activity (Circle with X).
- **Action/Activity State:** A step in the activity (Rounded rectangle).
- **Decision Node:** Branching point based on a condition (Diamond). Has one input and multiple outputs.
- **Merge Node:** Merging multiple alternate flows back into one (Diamond).
- **Fork Node:** Splitting a single flow into multiple concurrent (parallel) flows (Thick black bar).
- **Join Node:** Synchronizing multiple concurrent flows back into one; waits for all incoming flows to arrive (Thick black bar).
- **Swimlanes:** Partitions that group activities based on *who* (which class/actor/department) performs them.

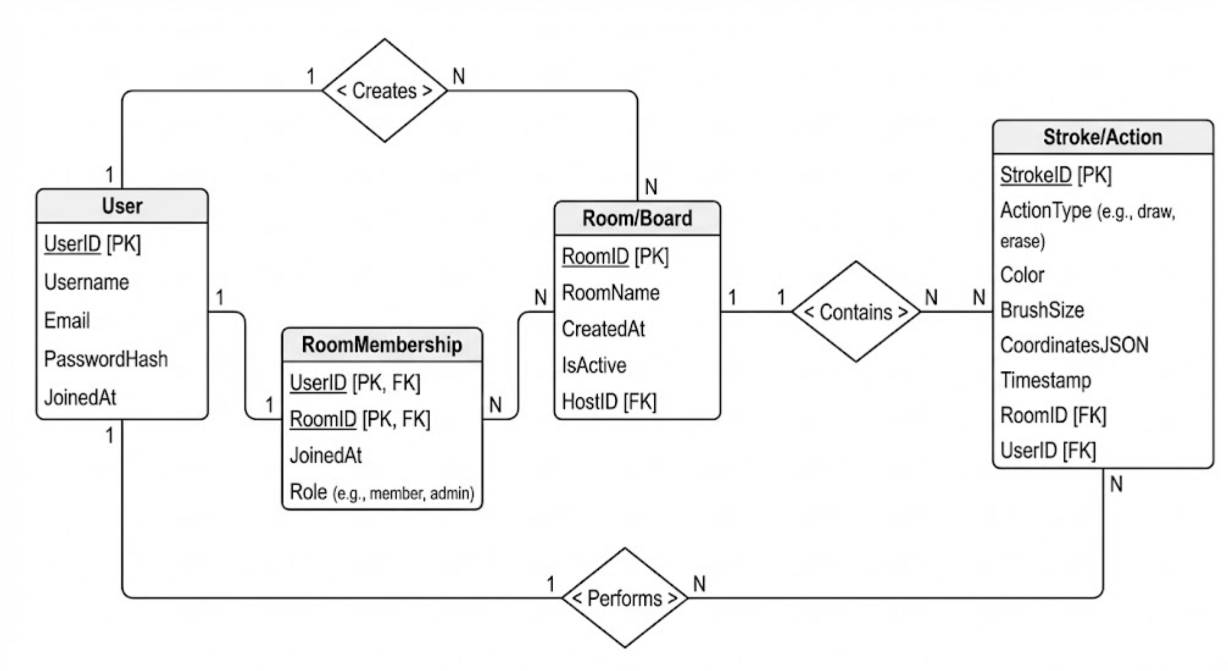


Part A: Hospital Management Flow



Part B: Login Page Flow

PRACTICAL 11: ER- Diagram



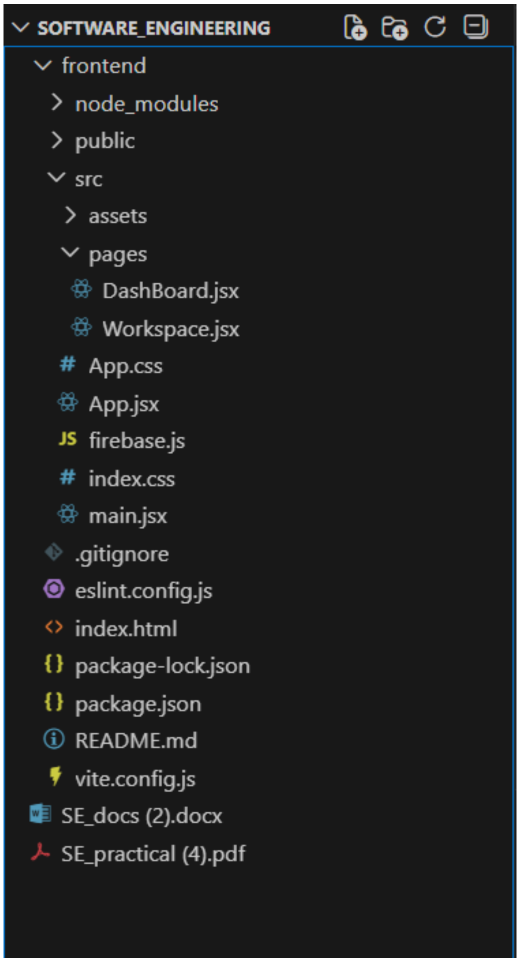
PRACTICAL 12: TEST CASES

While PRACTICAL 1 outlined the Testing Strategy, this section documents the specific execution of functional test cases.

TC ID	Test Scenario	Steps to Execute	Expected Result	Actual Result	Status
TC-01	**Create New Room**	1. Open App. 2. Click "Create Room".	System generates a unique Room ID (e.g., abc-123) and redirects to canvas.	Room ID generated, redirected.	**PASS**
TC-02	**Join Existing Room**	1. User A copies Room ID. 2. User B opens App. 3. User B pastes ID and clicks "Join".	User B enters the same session as User A.	User B joined successfully.	**PASS**
TC-03	**Real-Time Sync**	1. User A draws a circle. 2. User B observes their screen.	The circle appears on User B's screen within 100ms.	Circle appeared instantly.	**PASS**
TC-04	**Change Color**	1. Select "Red" from palette. 2. Draw a line.	Line renders in Red color.	Line is Red.	**PASS**

TC-05	**Eraser Tool**	1. Select "Eraser". 2. Drag over existing lines.	Lines under the cursor are removed/turn white.	Lines removed.	**PASS**
TC-06	**Session Persistence**	1. Draw on canvas. 2. Refresh the browser page.	Previous drawings should reload automatically.	Drawing reloaded from DB.	**PASS**

PRACTICAL 13: Coding screenshots




```
PS C:\Users\aksha\OneDrive\Desktop\Software_engineering\frontend> npm run dev

> frontend@0.0.0 dev
> vite


  VITE v7.2.4  ready in 699 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

```javascript
const firebaseConfig = {
  apiKey: "AIzaSyBNNWjs4aoGpEpHhHqtZ8uVCA86VPkxvYw",
  authDomain: "software-121340.firebaseapp.com",
  projectId: "software-121340",
  storageBucket: "software-121340.firebasestorage.app",
  messagingSenderId: "322241220296",
  appId: "1:322241220296:web:5194200d481c1ffa639ffd",
  measurementId: "G-VGWYDNDXW1"
};
const app = initializeApp(firebaseConfig);
const auth = getAuth(app);
const db = getFirestore(app);
const appId = typeof __app_id !== 'undefined' ? __app_id : 'default-app-id';

// --- Constants ---
const COLORS = [
  '#3B82F6', // Blue
  '#EF4444', // Red
  '#10B981', // Green
  '#F59E0B', // Yellow
  '#8B5CF6', // Purple
  '#EC4899'  // Pink
];

// --- Main App Component ---
export default function App() {
  const [user, setUser] = useState(null);
  const [boardId, setBoardId] = useState(null);

  useEffect(() => {
    const initAuth = async () => {
      if (typeof __initial_auth_token !== 'undefined' && __initial_auth_token) {
        await signInWithCustomToken(auth, __initial_auth_token);
      } else {
        await signInAnonymously(auth);
      }
    };
```

```javascript
function Workspace({ user, boardId, onLeave }) {
  const containerRef = useRef(null);
  const [activeTool, setActiveTool] = useState('move'); // move, cube, pen, text
  const [color, setColor] = useState(COLORS[0]);
  const [elements, setElements] = useState([]);
  const [selectedId, setSelectedId] = useState(null);
  const sceneRef = useRef(null);
  const cameraRef = useRef(null);
  const rendererRef = useRef(null);
  const labelRendererRef = useRef(null);
  const controlsRef = useRef(null);
  const raycaster = useMemo(() => new THREE.Raycaster(), []);
  const mouse = useRef(new THREE.Vector2());
  const plane = useMemo(() => new THREE.Plane(new THREE.Vector3(0, 1, 0), 0), []); // Ground plane

  // Refs for interaction
  const isDraggingRef = useRef(false);
  const dragStartRef = useRef(new THREE.Vector3());
  const dragObjectRef = useRef(null);
  const currentLineRef = useRef(null); // For drawing lines
```