Software Design Specifications

for

MU-UniConnect

Prepared by: Infinity 8

Document Information

Title: MU-Unicoonect	
Project Manager:	**Document Version No:3**
	Document Version Date:9th April 2025
Prepared By: Infinity 8	**Preparation Date:9th May 2025**

Table of Contents

1 Introduction

This document is structured to guide the reader through the complete software design.
Section 2: Use Case View outlines functional requirements through user interactions and use cases.
Section 3: Design Overview presents the high-level architecture and component interactions.
Section 4: Logical View details the internal structure using class diagrams and object relationships.
Section 5: Data View covers the data architecture, including schemas and data flow.
Section 6: Exception Handling explains mechanisms for error detection, logging, and resolution.
Section 7: Configurable Parameters lists adjustable system settings that don't require code changes.
Section 8: Quality of Service defines non-functional attributes such as performance, scalability, and availability.

1.1 Purpose

This section defines the purpose of the Software Design Specification (SDS) within the overall project documentation and outlines the structure of the document. The SDS serves as a bridge between the Software Requirements Specification (SRS) and the implementation phase, providing a detailed design roadmap for the development team. It ensures that all stakeholders have a common understanding of the system architecture, components, and design rationale. The document is structured into sections covering use cases, architectural design, logical structure, data models, exception handling, configurable parameters, and quality attributes.

1.2 Scope

This Software Design Specification applies to the design and development of the **MU-UniConnect** system. It covers all aspects of the software architecture, including component design, data structures, interfaces, and interactions necessary to meet the functional and non-functional requirements outlined in the SRS. The document influences and guides software implementation, testing strategies, system integration, and future maintenance. The document influences and guides software implementation, testing strategies, system integration, and future maintenance. It ensures that all technical elements are aligned with project goals, promoting consistency, scalability, and maintainability across the development lifecycle.

1.3 Definitions, Acronyms, and Abbreviations

SDS	Software Design Specification
SRS	Software Requirements Specification
UI	UserInterface
API	Application Programming Interface
DBMS	Database Management System
QoS	Quality of Service

1.4 References

This subsection provides a list of all documents referenced in this Software Design Specification. Each document is identified with its title and, where applicable, a URL link to the source document library. If a direct link is not available, the source or location from which the document can be obtained is specified.

[1] **IEEE 1016-2009: IEEE Standard for Information Technology—Systems Design—Software Design Descriptions**

Available at: https://standards.ieee.org/standard/1016-2009.html

[2] **Project Architecture Overview Document**

Available at: https://kilthub.cmu.edu/ndownloader/files/12056780

[3] **Database Design Document**

Available from: https://www.tandfonline.com/doi/abs/10.1080/10118063.1999.9724542

2 Use Case View

This section identifies and describes the key use cases that drive the software design. These use cases are selected based on their importance to the system's core functionality, the extent of design elements they interact with, or the way they highlight complex or critical aspects of the design. The use cases are derived directly from the Software Requirements Specification (SRS) and are essential for shaping the architecture and logic of the final system.
The use cases are **Schedule Events, Club Management, Register for Event, Update College Information, Approve Registrations, View college Information.**

Diagram:

2.1 Use Case

2.1.1 Use Case: Schedule Events

Description:
Enables authorized users—Admins, Faculty, and Club Heads—to schedule college or club events while checking for real-time conflicts with existing scheduled events.

Actors:

- Admin
- Faculty
- Club Head

Preconditions:

- User is authenticated and has appropriate RBAC permissions.
- Event calendar module is accessible.

Postconditions:

- If no conflicts exist, the event is saved to the database.
- If conflicts are detected, the user is prompted to reschedule.

Usage Steps:

- **Basic Flow:**
  - User navigates to the "Schedule Events" section from the dashboard.
  - User inputs event details (e.g., name, date/time, location, capacity).
  - System checks for scheduling conflicts within **< 2 seconds** (SRS P2).
  - If no conflict is found, the system saves the event and notifies relevant users.
- **Alternative Flow (Conflict Detected):**
  - System displays overlapping/conflicting events.
  - User modifies event timing or location, or cancels scheduling.
- **Exception Handling:**
  - If the database is unavailable, the system displays:
    "Event scheduling temporarily disabled. Please try again later."

2.1.2 Use Case: Club Management

Description:
Allows Admins and Club Heads to manage club details, including creation, editing, deletion, member management, and recruitment postings.

Actors:

- Admin
- Club Head

Preconditions:

- User has valid "Admin" or "Club Head" RBAC role (SRS S3).
- Club database is operational.

Postconditions:

- Club data is updated in the MongoDB database.
- Notifications are sent to affected users (e.g., for new memberships).

Usage Steps:

- **Basic Flow:**
  - User accesses "Club Management" from the admin panel.
  - User creates or edits club details (name, description, member list, etc.).
  - System validates data and updates the database accordingly.
- **Alternative Flow (Invalid Data):**
  - If input data is invalid (e.g., duplicate club name), system displays inline error hints.
- **Exception Handling:**
  - If RBAC permission validation fails, the system logs a security violation and denies access.

2.1.3 Use Case: Register for Event

Description:
Allows students to register for events, subject to available capacity and predefined approval rules.

Actors:

- Student

Preconditions:

- Student is logged into the system.
- The event is currently accepting registrations.

Postconditions:

- Registration is stored in the database.
- Student receives a confirmation or denial notification.

Usage Steps:

- **Basic Flow:**
  - Student browses the event list.
  - Student clicks "Register" on an event.

  - System verifies capacity and records the registration if space is available.
  - Student is notified of successful registration.
- **Alternative Flow (Event Full):**
  - System prevents registration and offers a waitlisting option.
- **Exception Handling:**
  - If a network failure occurs during submission, the transaction is rolled back and the student is shown a retry option.

2.1.4 Use Case: Approve Registrations

Description:
Provides Admins with tools to approve or reject student registrations, ensuring they align with event-specific policies and capacity.

Actors:

- Admin

Preconditions:

- There are pending registration requests.
- Admin has valid RBAC privileges.

Postconditions:

- Registration status is updated.
- Student receives an approval or rejection notification.

Usage Steps:

- **Basic Flow:**
  - Admin navigates to pending registrations.
  - Admin reviews each request and selects Approve or Reject, with optional comments.
  - System updates registration status and sends notification to the student.
- **Alternative Flow (Event Capacity Reached):**
  - If capacity is exceeded, system auto-rejects new requests with the message:
    "Event is full. Registration denied."

3 Design Overview

This section provides a high-level overview of the entire software design and establishes the foundation upon which detailed components are built. It outlines the system architecture, major modules, their responsibilities, and how they interact with one another. The design complies with the interface contracts and requirements defined in the Software Requirements Specification (SRS) and adheres to the architectural principles established in the high-level design.

The system follows a modular design approach to ensure scalability, maintainability, and ease of integration. Each module is responsible for a specific set of functions and communicates with other modules through clearly defined interfaces. This decomposition facilitates parallel development, testing, and reusability.

The design incorporates the following key elements:

- **System Architecture:** A layered architecture separating presentation, business logic, and data access layers.
- **Module Decomposition:** High-level modules such as User Management, Data Processing, Reporting, and Admin Control are broken down into smaller, manageable components.
- **Interface Contracts:** Defined APIs and service interfaces govern communication between modules and external systems.
- **Design Principles:** The design follows SOLID principles, design patterns where appropriate, and aims to reduce coupling while increasing cohesion across components.

3.1 Design Goals and Constraints

This section outlines the key design objectives and constraints that significantly influence the structure, strategy, and implementation of the software system. These factors shape how the design is approached, the tools and technologies chosen, and the priorities set during development.

Design Goals

The primary goals of the software design include:

- **Modularity:** The system should be composed of discrete, well-defined modules to promote reusability and ease of maintenance.
- **Scalability:** The design should support future growth in terms of user load, data volume, and feature enhancements.
- **Security:** Ensure secure handling of sensitive data through authentication, authorization, and data protection mechanisms.
- **Performance:** The system should provide fast response times and efficient resource utilization under expected workloads.
- **Maintainability:** The system should be easy to understand, modify, and extend by the development team.
- **Usability:** The user interface should be intuitive and accessible, minimizing user training and errors.

Constraints

Several constraints impact the design and implementation of the system:

- **Technology Stack:** The system must be built using [Java, ReactJS, MongoDB], as required by organizational standards or existing infrastructure.
- **Development Tools:** Tools such as [Git, VS code, Jira] are mandated for version control, deployment, and project tracking.
- **Team Structure:** The development team is divided into frontend, backend, and QA sub-teams, influencing how modules are assigned and integrated.
- **Schedule Constraints:** The project must be completed within a strict timeline, requiring careful prioritization and iterative delivery.
- **Legacy Code:** Integration with or reuse of existing legacy components is required in specific modules.

3.2 Design Assumptions

The software design is based on several key assumptions. It is assumed that the core requirements will remain stable throughout the development process and that the system will operate in an environment with reliable internet connectivity. User roles and access levels are predefined and limited to known categories. The design also assumes that any third-party APIs or external services will be consistently available and function as documented.

3.3 Significant Design Packages

The software design is organized into layered packages, each handling a specific responsibility. The **Presentation Layer** manages user interfaces, the **Business Logic Layer** handles core processing, and the **Data Access Layer** interacts with the database. Additional packages support external integrations and utility functions. Dependencies are structured hierarchically to maintain modularity and ensure clean separation of concerns.

3.4 Dependent External Interfaces

This section outlines the external interfaces the system depends on, along with the internal modules that use them. These interfaces enable integration with essential services such as authentication, email notifications, and reporting. Proper interaction with these interfaces ensures that key system features function as expected.

The table below lists the public interfaces this design requires from other modules or applications.

External Application	Module Using the Interface	Functionality/ Description

and Interface Name		
AuthService – AuthAPI.login()	Authentication Module	Used to validate user credentials during login and maintain secure access control.
EmailService – EmailAPI.sendMail()	Notification Module	Sends automated emails for registration confirmation, alerts, and other notifications.
ReportingService – ReportAPI.generate()	Report Generation Module	Generates user-specific reports based on submitted data and predefined templates.

3.5 Implemented Application External Interfaces (and SOA web services)

This section lists the public interfaces developed and exposed by the application for use by other systems or external modules. These interfaces are implemented within specific internal modules and support functionalities such as data access, reporting, and user management. The table below identifies each interface, the module responsible for its implementation, and a brief description of how it supports external integration.

The table below lists the implementation of public interfaces this design makes available for other applications.

Interface Name	**Module Implementing the Interface**	**Functionality/ Description**
UserAPI.getProfile()	User Management Module	Provides user profile information to external systems upon request.
ReportAPI.generateReport()	Report Generation Module	Generates reports based on user input and makes them accessible to authorized external systems.
NotificationAPI.sendAlert()	Notification Module	Sends alert messages or status updates to subscribed external applications.

4 Logical View

This section describes the detailed design of the Online Shopping Management System using a layered architecture. It focuses on how different components of the system interact to complete the main functionalities described in Section 2 (Use Case View), such as user registration, product browsing, order placement, and admin management.

Layered Architecture Overview:

1. **Presentation Layer**
   This layer manages all interactions between the users (customers or administrators) and the system. It provides the user interface for activities like registration, login, browsing products, and managing inventory. It sends user actions to the business logic and displays appropriate responses.
2. **Business Logic Layer**
   This layer contains the core logic that implements the functional requirements. It processes user inputs received from the presentation layer and coordinates necessary operations such as validating users, placing orders, managing product data, and applying business rules.
3. **Data Access Layer**
   Responsible for interacting with the underlying database. It handles the storage, retrieval, and updating of data related to users, products, categories, and orders. This layer ensures that all data transactions are abstracted from the business logic.
4. **Database Layer**
   This is the system’s persistent data store. It includes structured tables for managing essential entities like users, products, orders, and categories. This layer ensures reliable data storage and retrieval, supporting all other layers with the necessary data.

4.1 Design Model

The design model provides a class-based perspective of the Online Shopping Management System. It outlines the major components (modules) of the system and the key classes within each module. This structure enables a clear understanding of how the system is organized in terms of object-oriented principles like encapsulation, inheritance, and modularity.

Module Decomposition

The system is logically divided into the following modules:

- **User Management**
- **Product Management**
- **Order Management**
- **Category Management**
- **Authentication**
- **Admin Controls**

Each module contains classes that collaborate to fulfill the corresponding functionality.

Key Classes and Their Responsibilities

1. **User**
   a. **Attributes**: userID, name, email, password, address
   b. **Responsibilities**: Represents a registered user; stores and retrieves user details.
2. **Product**
   a. **Attributes**: productID, name, description, price, categoryID, stock
   b. **Responsibilities**: Holds product-related data; supports operations like search, update, and display.
3. **Admin**
   a. **Attributes**: adminID, name, email, password
   b. **Responsibilities**: Performs backend operations like adding or updating products and categories.
4. **AuthenticationService**
   a. **Responsibilities**: Handles login validation and access control for both users and admins.

Relationships

- A **User** can place multiple **Orders**.
- An **Order** contains multiple **Order of Items**.
- Each **OrderItem** is associated with one **Product**.
- A **Product** belongs to one **Category**.
- An **Admin** manages **Products** and **Categories**.

Class Diagram

If diagrams are required for submission, a class diagram can visually show:

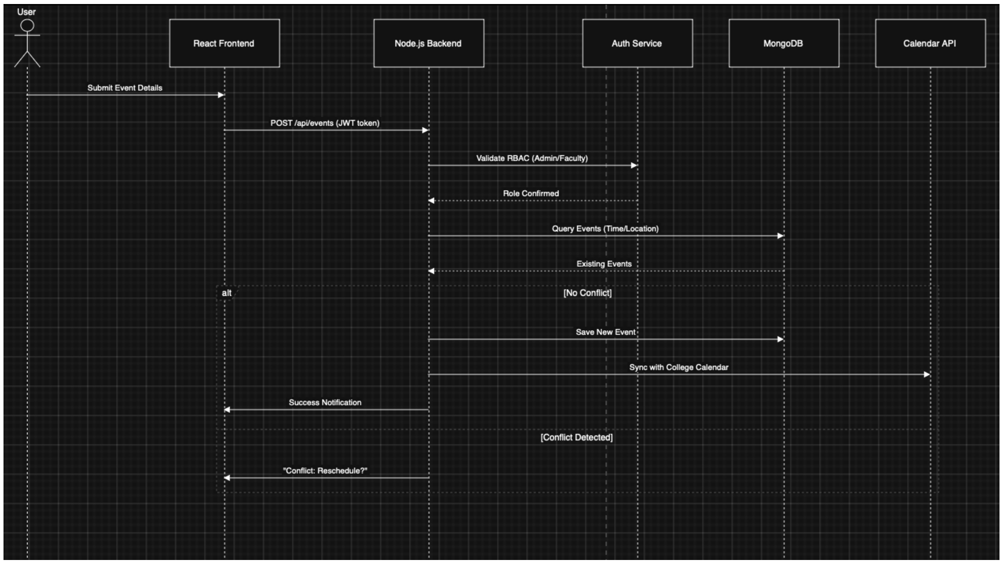
- Associations (e.g., User ↔ Order)
- Aggregations (e.g., Order → OrderItem)
- Inheritance if any
- Attributes and methods in each class

4.2 Use Case Realization

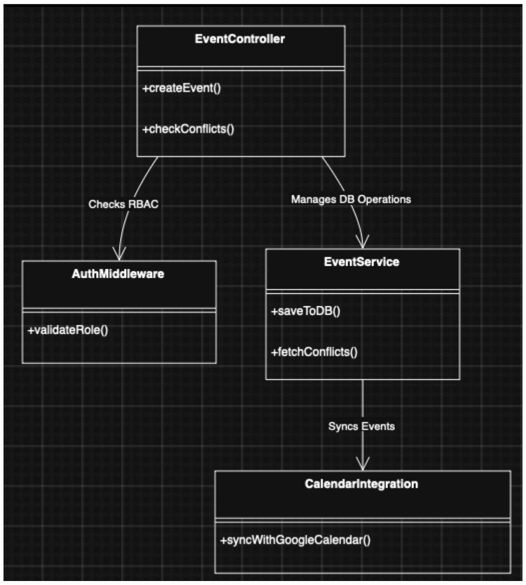
USE CASE: Schedule Events

SRS References: 3.3.3 (Use Case Model), P2 (Performance), S3 (RBAC)

High-Level Sequence Diagram:



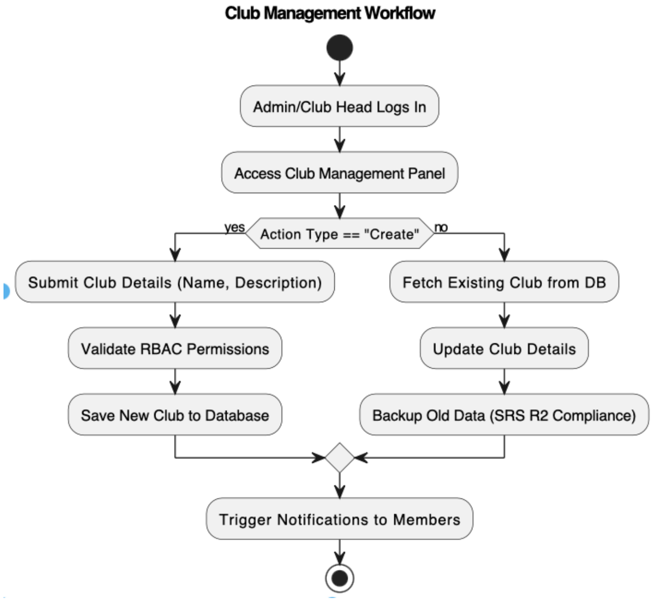
Low level class interactions:



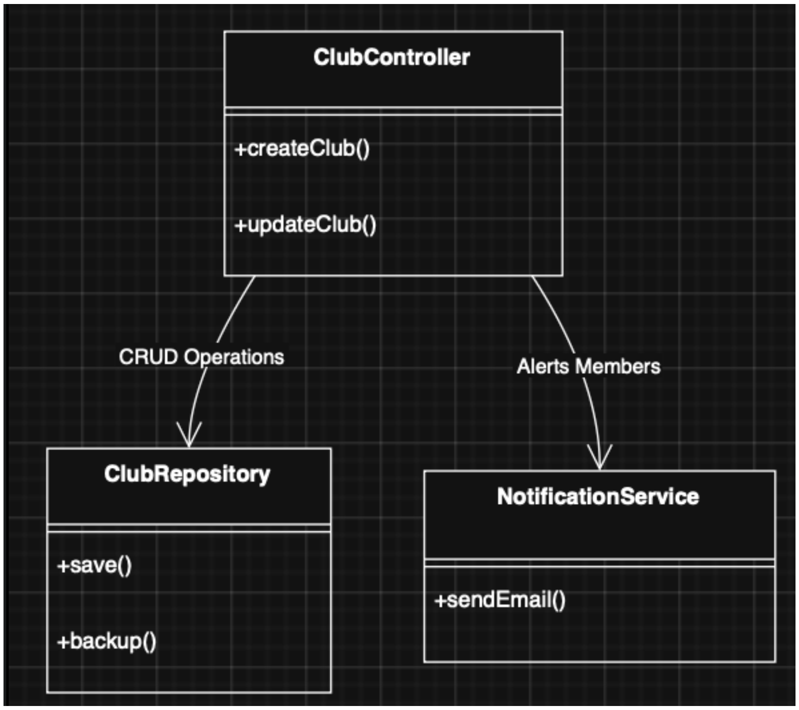
Key Design Elements:

- **Conflict Detection**: EventService queries MongoDB with { location, time } filters (optimized indexes for **SRS P2**).
- **RBAC**: AuthMiddleware validates USER_ROLE against the **Data Dictionary**.
- **Real-Time Sync**: CalendarIntegration uses Google Calendar API.

USE CASE: Club Management
SRS References: 3.3.4 (Use Case Model), S3 (RBAC), R2 (Backups)
High Level Activity Diagram:


Club Management Workflow
Admin/Club Head Logs In
Access Club Management Panel
Action Type == "Create"
yes
no
Submit Club Details (Name, Description)
Validate RBAC Permissions
Save New Club to Database
Fetch Existing Club from DB
Update Club Details
Backup Old Data (SRS R2 Compliance)
Trigger Notifications to Members

Class Collaborations:

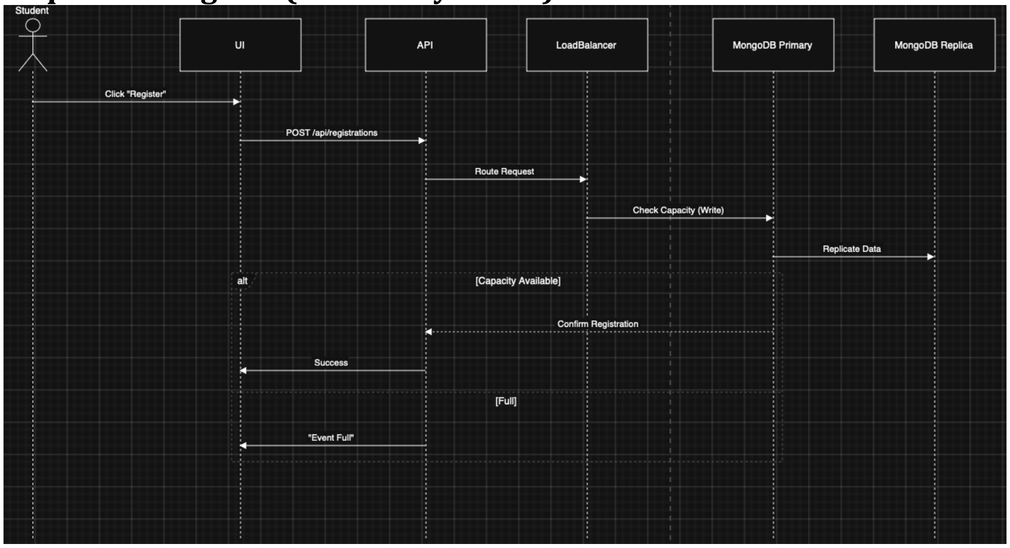


Key Design Elements:

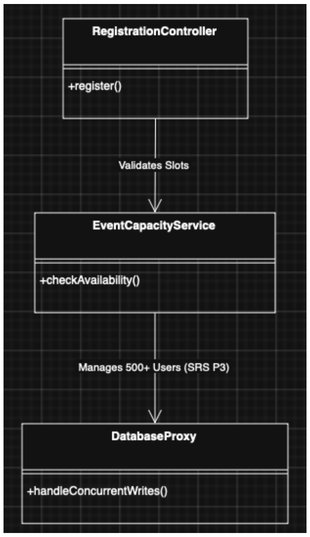
- **Data Integrity**: ClubRepository implements daily backups (**SRS R2**).
- **Notifications**: NotificationService triggers emails/SMS via AWS SNS.

USE CASE: Register For Event
SRS References: 3.3.6 (Use Case Model), P3 (Concurrent Users)

Sequence Diagram (Scalability Focus)



Class Diagram:



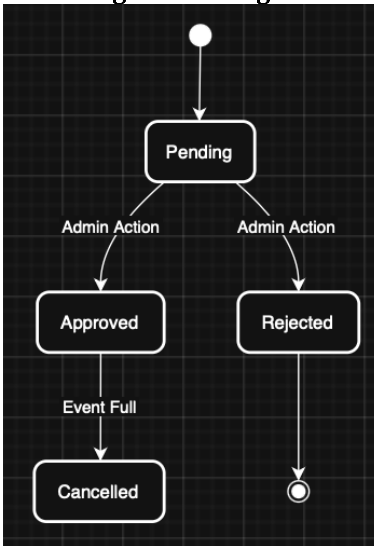
Key Design Elements:

- **Concurrency**: DatabaseProxy uses connection pooling and optimistic locking.
- **Capacity Checks**: EventCapacityService caches event slots to reduce DB load.

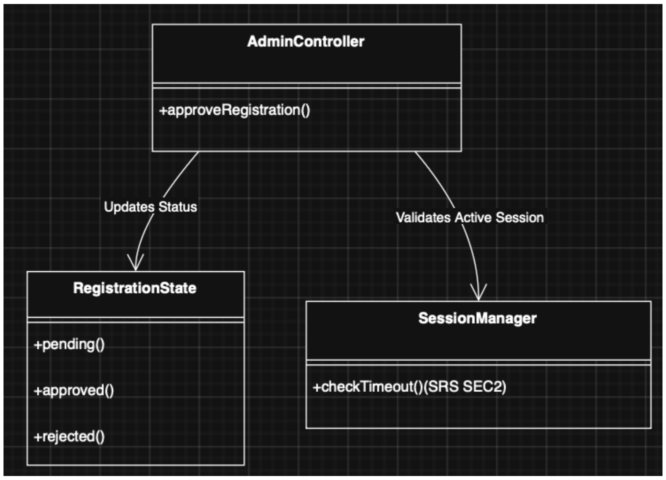
USE CASE: Approve Registrations

SRS References: 3.3.7 (Use Case Model), S3 (RBAC), SEC2 (Session Timeout)

State Diagram for Registration Workflow



Class Interactions:



Key Design Elements:

- **Session Security**: SessionManager enforces 15-minute inactivity logout (**SRS SEC2**).
- **State Pattern**: RegistrationState handles transitions (e.g., Pending → Approved).

5 Data View

The application maintains persistent data using a document-oriented NoSQL database (MongoDB). Core entities such as Users, Roles, Devices, Transactions, and Event Logs are stored as BSON documents, reflecting the structure defined in the domain model.

Indexes are applied to optimize query performance for frequently accessed fields. Data consistency is managed through application-level validation and update mechanisms. Sensitive data fields (e.g., credentials, tokens) are encrypted at rest using field-level encryption standards.

Backup, retention, and archival policies are enforced to support data durability and compliance requirements.

5.1 Domain Model

The domain model defines the core entities and their relationships, representing the system’s persistent data structure. Each entity corresponds to a collection in MongoDB and is modeled as a document with embedded or referenced relationships as appropriate.

Primary Entities:

- **User**: Represents system users; includes profile, credentials, and role assignments.
- **Role**: Defines access levels and permissions; referenced by User.
- **Device**: Represents connected hardware; includes metadata and status.
- **Transaction**: Logs user or device interactions; includes timestamps and context.
- **EventLog**: Captures system and device events; supports audit and troubleshooting.

Relationships:

- A **User** can have multiple **Roles** (1:N).
- A **Device** may have multiple **EventLogs** and **Transactions** (1:N).
- **Transactions** reference both **User** and **Device** for traceability.

5.2 Data Model (persistent data view)

The persistent data model is designed around core domain entities stored in a document-oriented database (MongoDB). Each collection represents a major system entity, and relationships are handled through referencing or embedding based on access and performance considerations.

5.2.1 Data Dictionary

Entity	Field	Data Type	Description
User	`userId`	String	Unique identifier for the user
	`username`	String	Login name of the user
	`email`	String	Contact email address
	`roles`	Array[String]	List of assigned role identifiers
	`createdAt`	DateTime	Account creation timestamp
Role	`roleId`	String	Unique role identifier

	`name`	String	Descriptive name of the role
	`permissions`	Array[String]	List of feature access permissions
Device	`deviceId`	String	Unique identifier for the device
	`type`	String	Device type/category
	`status`	String	Current operational status
	`lastSeen`	DateTime	Timestamp of last communication
Transaction	`transactionId`	String	Unique ID for transaction
	`userId`	String	Reference to user initiating the transaction
	`deviceId`	String	Reference to target device
	`action`	String	Action performed
	`timestamp`	DateTime	When the transaction occurred
EventLog	`eventId`	String	Unique event identifier
	`deviceId`	String	Reference to associated device
	`level`	String	Severity level (info, warning, error)
	`message`	String	Event message content
	`timestamp`	DateTime	Time of event occurrence

6 Exception Handling

The application defines structured exception types for predictable error scenarios. Key exceptions include:

Exception Type	Description / Trigger	Logging	Follow-up Action
`ValidationException`	Thrown when input data fails validation rules	Logged as warning	Prompt user to correct input
`AuthenticationException`	Triggered on failed login or expired session	Logged as warning	Redirect to login
`AuthorizationException`	Access to unauthorized resource	Logged as warning	Display access denied message
`DatabaseException`	DB connection/query failure	Logged as error	Retry or alert admin
`ExternalServiceException`	Failed integration with external APIs	Logged as error	Retry or fallback to alternate flow
`SystemException`	Unhandled or critical failure	Logged as critical	Alert ops, initiate failover if needed

7 Configurable Parameters

This table describes the simple configurable parameters (name / value pairs).

Configuration Parameter Name	Definition and Usage	Dynamic?
`MAX_CONCURRENT_USERS`	Defines the upper limit of simultaneously active user sessions	No
`API_TIMEOUT_MS`	Sets timeout duration (in milliseconds) for external API calls	Yes
`LOG_LEVEL`	Controls the granularity of application logging (e.g., INFO, DEBUG, ERROR)	Yes
`CACHE_EXPIRY_SECONDS`	Determines how long cached data is retained before	Yes

	expiration	
BATCH_JOB_INTERVAL_MIN	Time interval (in minutes) between execution of scheduled background jobs	Yes
DB_CONNECTION_POOL_SIZE	Sets the maximum number of database connections in the pool	No

8 Quality of Service

This system ensures high availability, robust security, optimal performance, and effective monitoring:

- **Availability**: 99.9% uptime with failover support and automatic recovery.
- **Security**: OAuth 2.0 with MFA, RBAC, TLS 1.3 for data in transit, and AES-256 for data at rest.
- **Performance**: Supports 1,000 concurrent users with <2s response time; APIs <500ms latency.
- **Monitoring**: Real-time metrics and alerts, centralized logging and audit trails.

8.1 Availability

The application is designed to meet the business requirement of 99.9% uptime. High availability is achieved through redundancy, automated failover, and load balancing. The system supports zero-downtime deployments and includes self-healing mechanisms.

Scheduled maintenance (e.g., mass data loads, housekeeping tasks) will be conducted during off-peak hours and communicated in advance. These activities are optimized to minimize service disruption and will not exceed 0.1% downtime per month.

8.2 Security and Authorization

The system enforces strict access control aligned with business requirements for data confidentiality and feature security. Role-Based Access Control (RBAC) is implemented along with Multi-Factor Authentication (MFA) for sensitive roles.

Authorization is integrated at both the feature and data levels. Custom qualifiers support fine-grained access, ensuring users only access permitted resources. Administrative interfaces allow designated personnel to manage roles, permissions, and user lifecycle securely.

8.3 Load and Performance Implications

The system is designed to support up to 1,000 concurrent users with a peak transaction rate of 100 TPS (transactions per second). API response times are expected to remain under 500 ms under typical load, and batch jobs should complete within defined SLAs.

Message queues and asynchronous processing are used for high-volume operations. Database tables are expected to grow by ~5GB/month; indexing, partitioning, and archival strategies are incorporated to maintain query performance.

8.4 Monitoring and Control

The application includes controllable components such as background daemons, message handlers, and scheduled jobs. These processes can be started, stopped, or restarted via an admin interface.
Key metrics (CPU, memory, message queue length, job execution time, error rates) are published to a centralized monitoring system. Alerts are configured for threshold breaches, and logs are captured in structured format for traceability and diagnostics.