**SM CINEMA BOOKING SYSTEM**

A Design Document Presented to the

Faculty of Datamex College of Saint Adeline, Inc.

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science in Information Technology

By:

Jerusalem, Auldey Glen A.

September 2025

**DESIGN DOCUMENT**

**INTRODUCTION**

This design document serves as a comprehensive technical blueprint for the development of the Cinema Booking System for SM Cinema. It outlines the system architecture, database design, user interface specifications, and implementation strategies required to deliver a robust web-based cinema ticket booking platform. This document will guide the development process and serve as a reference for future maintenance and enhancements.

The Cinema Booking System is an interactive web-based application designed to modernize and streamline the movie ticket reservation experience for SM Cinema customers and operational staff. The system addresses critical pain points in existing booking platforms, including unresponsive interfaces, limited user features, and inefficient concurrent seat reservation handling.

Built using modern web technologies including React for the frontend interface and Django REST Framework for the backend API, the system provides a comprehensive solution for cinema operations. The platform enables customers to browse current movie listings, view available showtimes, select specific seats through an interactive seating map, and complete reservations seamlessly. For administrative staff, the system includes a dedicated dashboard for managing movie schedules, monitoring reservation activities, and accessing detailed audit logs.

The system is architected as a client-server application with a PostgreSQL database backend, ensuring data integrity and scalability. All components are designed with responsiveness in mind, providing optimal user experience across desktop and mobile devices.

**Scope of the Design Document**

This design document encompasses the complete technical specification for the Cinema Booking System, covering all aspects from high-level system architecture to detailed component implementations

**Included System Features:**

* Customer-facing movie browsing and seat selection interface
* Real-time seat availability and reservation management
* Administrative dashboard for schedule and reservation oversight
* Email-based ticket confirmation and delivery system
* Responsive web design optimized for multiple device types
* Comprehensive audit logging for administrative actions
* User authentication and session management

The design focuses on core booking functionalities essential for cinema operations. The system handles movie listings, showtime management, seat reservations, and basic administrative controls. Integration points are defined for email services and potential future payment gateway implementations.

This document provides detailed specifications for database schema design, API endpoint definitions, user interface wireframes, security implementations, and deployment architecture. It serves as the authoritative reference for development decisions and system maintenance procedures.

The design document will be maintained throughout the development lifecycle, with revisions tracked to ensure accuracy and relevance as the system evolves to meet SM Cinema's operational requirements.

**SYSTEM ARCHITECTURE**

The Cinema Booking System follows a modern, cloud-native microservices architecture designed for scalability, maintainability, and cost-effectiveness. The system is built as a distributed application with clear separation of concerns across multiple cloud platforms, utilizing free-tier services to ensure production-ready deployment without infrastructure costs.

**High-Level Components and Their Interactions**

The system consists of five core components that communicate through well-defined interfaces. The React frontend serves as the user interface layer, consuming services from the Django REST Framework API. The PostgreSQL database handles persistent data storage, while Redis provides caching capabilities. Cloudinary manages media assets including movie posters and promotional images.

The frontend communicates with the backend exclusively through RESTful HTTP requests, maintaining a stateless architecture that supports scalability. The backend API handles all business logic, authentication, and data validation before interacting with the database layer.

**Deployment Architecture**

The application utilizes a multi-tier cloud deployment strategy. The React frontend is hosted on Vercel, providing global CDN distribution and automatic SSL certificate management. The Django REST Framework backend runs on Render, offering containerized deployment with environment variable management.

Data persistence is managed through Supabase's PostgreSQL service, while Upstash provides Redis caching functionality. Cloudinary handles all media storage and delivery through their CDN infrastructure. This distributed approach enables cross-domain operation with appropriate CORS configuration, allowing each component to scale independently based on demand.

**Communication Protocols and Interfaces**

All client-server communication operates over HTTPS using JSON as the primary data exchange format. The API follows REST conventions with a versioned namespace structure using /api/v1/ as the base URL.

Authentication is implemented using Django's token-based system, where clients include authentication tokens in the Authorization header for protected endpoints. The system maintains stateless sessions to support horizontal scaling.

The current API structure includes comprehensive movie and genre management endpoints with query parameter support for different response formats. Additional endpoints for user authentication, booking management, and administrative functions are planned for future implementation.

**API Design Coverage**

The system provides comprehensive movie and genre management capabilities with search functionality and flexible response formatting. Administrative endpoints enable content management operations, while user authentication ensures secure access control.

Core booking functionality includes reservation handling with seat selection, showtime scheduling and management, and administrative dashboard operations for system oversight. The API architecture supports both customer-facing operations and administrative workflows through distinct endpoint groupings.

**System Integration Points**

The system integrates with external services for email delivery, utilizing Django's built-in email backend for ticket confirmations. Version control is managed through GitHub with automated deployment pipelines to the respective cloud platforms.

Application monitoring and logging are handled through Django's logging framework combined with platform-specific monitoring tools provided by the hosting services.

**DATABASE DESIGN**

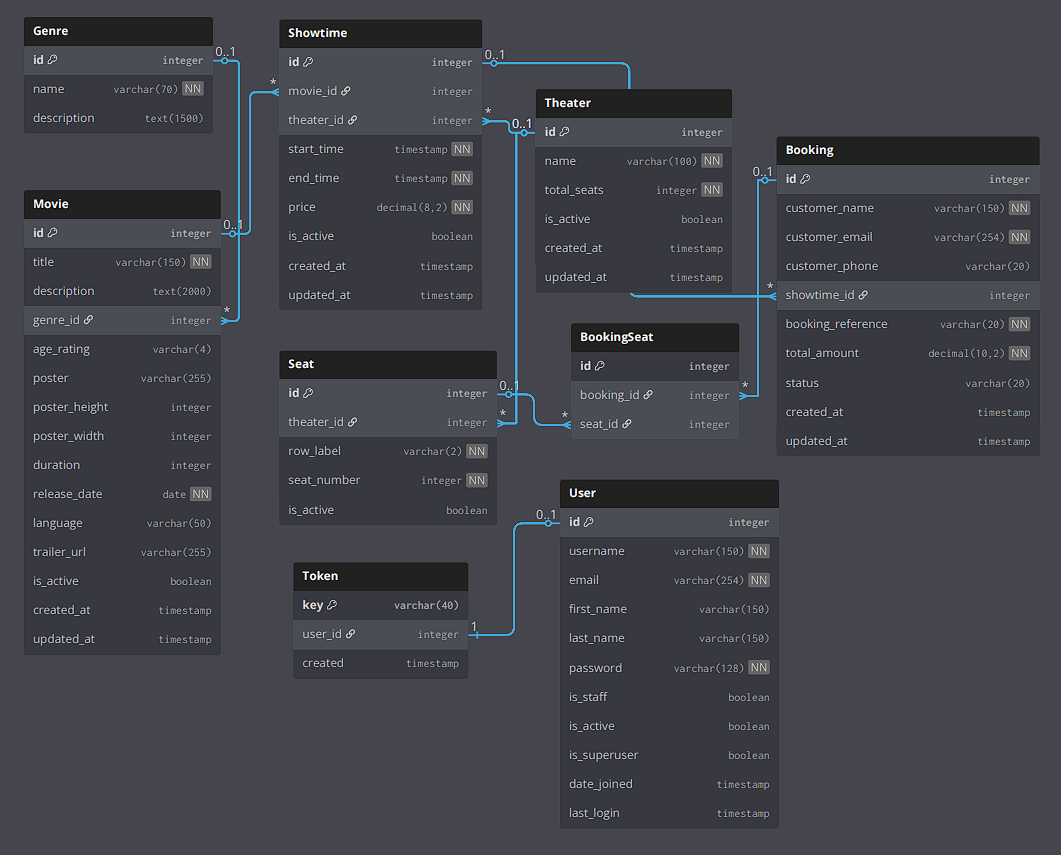
**Entity-Relationship Diagram (ERD)**

Figure 1. Entity Relationship Diagram (ERD)

The database schema follows a relational model designed to support efficient cinema booking operations while maintaining data integrity and performance. The ERD depicts eight core entities with well-defined relationships that support both customer booking workflows and administrative management functions.

**Description of Database Tables, Fields, and Relationships**

**Core Content Tables:**

* Genre stores movie categories with name and description fields
* Movie contains film details including title, description, duration, rating, poster URL, and release information with foreign key to Genre
* Theater defines screening venues with name and seat capacity
* Seat models individual theater seating using row labels and seat numbers, linked to specific theaters

**Scheduling and Booking:**

* Showtime connects movies to theaters with scheduling and pricing information
* Booking captures customer reservations with embedded customer details (name, email, phone) and booking references
* BookingSeat junction table links bookings to specific seats for multi-seat reservations

**Authentication:**

* User implements Django's authentication model for administrative staff only
* Token provides API authentication with one-to-one relationship to User table

Key relationships include Movie-to-Genre (many-to-one), Showtime-to-Movie and Showtime-to-Theater (many-to-one), and Booking-to-Seat (many-to-many through BookingSeat).

**Data Normalization Techniques Used**

The database follows Third Normal Form (3NF) principles to eliminate redundancy while maintaining performance. Tables satisfy 1NF with atomic values, 2NF with full functional dependencies, and 3NF by eliminating transitive dependencies.

Customer information is embedded directly in the Booking table rather than normalized to a separate Customer table, as the system supports guest bookings without persistent user accounts. This design choice reduces complexity while maintaining data integrity for email-based ticket delivery.

Pricing is stored at the Showtime level to support flexible pricing strategies including time-based and special event pricing. The design uses soft deletion through is\_active fields to maintain referential integrity while hiding inactive content from customer views.

**USER INTERFACE DESIGN**

**Wireframes and Mockups**

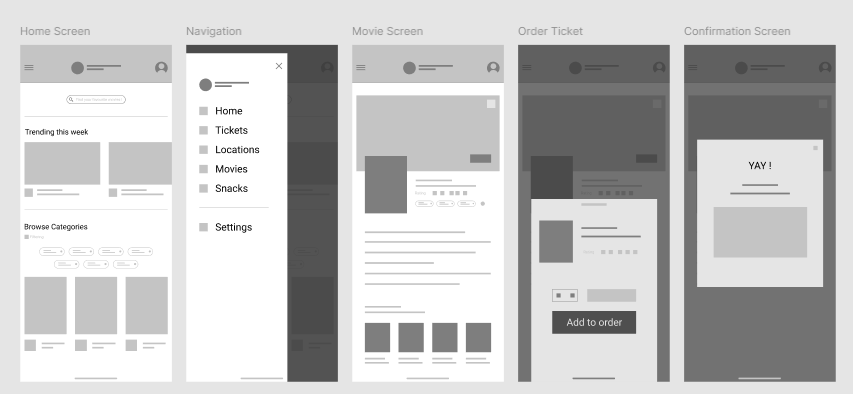
The user interface design prioritizes mobile-first responsive design, recognizing that customers primarily access the booking system through mobile browsers. The wireframes illustrate a streamlined customer journey optimized for touch interaction and quick booking completion.

Image 1. System User Interface Design Mockups

**Description of User Interface Elements, Layout, and Navigation**

**Customer Interface**

The home screen presents movies in a responsive grid layout with poster thumbnails, titles, and ratings. Sidebar navigation provides access to Movies, Tickets, and support sections.

Movie detail pages feature large poster displays, comprehensive film information, and prominent "Order Ticket" buttons. The seat selection interface uses an interactive theater layout with color-coded availability indicators. The booking form captures customer details through mobile-optimized input fields, concluding with popup confirmation screens.

**Administrative Interface**

The admin dashboard utilizes standard CRUD interface patterns for desktop usage, providing movie management capabilities, schedule oversight, and booking administration tools. Navigation follows a horizontal menu structure with role-based access controls.

**Design Principles and Usability Considerations**

The design prioritizes mobile browser experience with touch-friendly elements and responsive layouts that scale across device sizes. Interface elements follow accessibility standards with appropriate contrast and semantic structure.

The booking flow maintains logical progression from movie discovery to confirmation, minimizing cognitive load while providing clear progress indicators and visual feedback for user actions.

**COMPONENT DESIGN**

**Description of Key System Components / Modules**

**Frontend Components (React)**

The React application consists of core components including MovieList for displaying available films, MovieDetail for individual movie information, SeatSelection for interactive theater layout, BookingForm for customer data capture, and AdminDashboard for administrative functions. Each component maintains state management through React hooks and communicates with the backend via API calls.

**Backend Modules (Django REST Framework)**

The backend architecture includes Movies application handling film management, Bookings application managing reservations, Authentication module for admin access, and API serializers for data transformation. Core modules include MovieViewSet, GenreViewSet, BookingViewSet, and custom authentication middleware.

**Interface Specifications for Each Component**

**API Endpoints**

Movie management utilizes RESTful endpoints with GET /api/v1/movies/ for listing, POST for creation, and PATCH/PUT/DELETE for modifications. Genre management follows similar patterns with /api/v1/genres/ base URL. Booking endpoints include POST /api/v1/bookings/ for reservations and GET for retrieval operations.

**Component Props and Methods**

React components accept props for data passing and implement methods for user interactions. MovieList component accepts movies array and loading state props. SeatSelection component receives theater layout data and callback functions for seat selection handling.

**Dependency Management and Interaction Between Components**

Frontend dependencies are managed through npm with React as the core framework. Backend dependencies utilize pip with Django REST Framework, PostgreSQL adapter, and Redis client. Components interact through well-defined API contracts, with frontend components consuming backend services through HTTP requests and response handling.

**DATA FLOW DIAGRAMS**

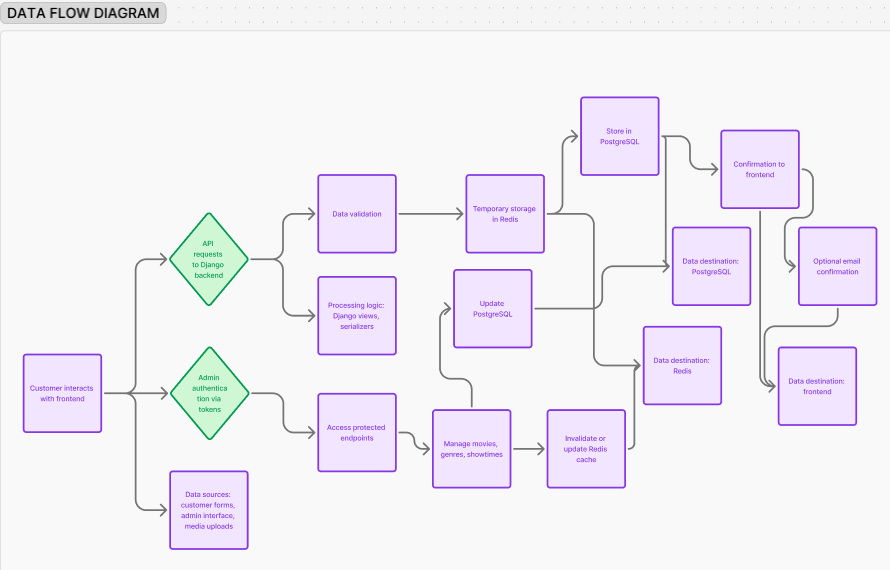


Figure 2. Data Flow Diagram (DFD)

**Data Flow Through the System**

Customer interactions on the React frontend send API requests to the Django backend, where data is validated, processed according to business logic, temporarily stored in Redis for seat holding, and persisted in PostgreSQL, with confirmations returned to the frontend and optionally sent via email.

Admins authenticate via tokens to access protected endpoints, perform CRUD operations on movies, genres, and showtimes, and update the database while managing cache invalidation as needed. Data sources include customer forms, admin interfaces, and external media uploads, while processing is handled by.

**SECURITY DESIGN**

**Security Requirements and Considerations**

The system implements comprehensive security measures including input validation, SQL injection prevention through Django ORM, XSS protection via content security policies, and HTTPS enforcement across all communications. Rate limiting prevents abuse, while secure headers protect against common web vulnerabilities.

**Authentication and Authorization Mechanisms**

Administrative access utilizes Django's built-in authentication with token-based API authentication. Tokens are generated upon login and included in Authorization headers for protected endpoints. Role-based access control differentiates between staff and superuser permissions for appropriate feature access.

**Data Encryption and Protection Measures**

All data transmission occurs over HTTPS with TLS encryption. Database connections utilize encrypted connections to Supabase. User passwords are hashed using Django's PBKDF2 algorithm. API tokens are securely generated and stored with appropriate expiration policies.

**PERFORMANCE DESIGN**

**Performance Requirements and Objectives**

The system targets sub-2-second page load times for movie listings, real-time seat availability updates, and efficient handling of concurrent booking requests. Database queries are optimized through indexing strategies, while CDN delivery ensures fast media loading.

**Performance Optimization Strategies**

Caching strategies include Redis for frequently accessed data, Cloudinary CDN for image delivery, and browser caching for static assets. Database optimization utilizes indexed fields for common queries and connection pooling for efficient resource usage. Frontend optimization includes code splitting and lazy loading for improved initial load times.

**Performance Testing Plan**

Performance testing includes load testing for concurrent user scenarios, database performance monitoring under peak conditions, and API response time benchmarking. Monitoring tools track system metrics and identify performance bottlenecks for optimization.

**ERROR HANDLING AND LOGGING**

**Error Handling Mechanisms and Strategies**

The system implements comprehensive error handling with try-catch blocks in critical code paths, graceful degradation for non-critical failures, and user-friendly error messages. API errors return appropriate HTTP status codes with descriptive error responses.

**Logging Requirements and Specifications**

Application logging captures system events, user actions, and error conditions through Django’s logging framework. Log levels include DEBUG for development, INFO for general operations, WARNING for potential issues, and ERROR for system failures. Logs are structured for easy parsing and analysis.

**Error Codes and Messages**

Standardized error codes include 400 for bad requests, 401 for authentication failures, 403 for authorization errors, 404 for not found resources, and 500 for server errors. User-facing messages provide clear guidance for resolution while technical details are logged for debugging.

**THIRD-PARTY INTEGRATIONS**

**Third-Party Services and APIs**

* **Supabase -** Managed PostgreSQL database service providing data persistence, backup, andmonitoring capabilities.
* **Upstash -** Redis caching service for performance optimization and temporary data storage
* **Cloudinary** - Media management service for movie poster storage, optimization, and CDN delivery
* **Email Service -** SMTP integration for ticket confirmation and delivery to customers

**Integration Points and Data Exchange Formats**

Database integration utilizes PostgreSQL connection strings with JSON data exchange for complex data types. Cloudinary integration uses REST API for image uploads with URL responses for frontend consumption. Redis integration employs standard Redis protocol for caching operations. Email integration uses SMTP with HTML-formatted message templates.

**DEPLOYMENT PLAN**

**Deployment Process Overview**

The deployment strategy utilizes automated CI/CD pipelines through GitHub integration. Frontend deployment to Vercel occurs automatically on repository updates, while backend deployment to Render follows similar automation. Database migrations are handled through Django's migration system with proper staging procedures.

**Hardware and Software Requirements**

* **Frontend**: Vercel hosting with automatic scaling and CDN distribution
* **Backend**: Render containerized deployment with minimum 512MB RAM allocation
* **Database**: Supabase managed PostgreSQL with automated backups and monitoring
* **Caching**: Upstash Redis with appropriate memory allocation for concurrent sessions

**Configuration Management and Version Control**

Environment variables manage configuration across deployment stages with secure handling of sensitive data. Git-based version control maintains code history and enables rollback capabilities. Database schema changes are versioned through Django migrations with proper testing procedures.

**MAINTENANCE AND SUPPORT**

**System Maintenance Guidelines**

Regular maintenance includes database optimization, security patch applications, dependency updates, and performance monitoring. Scheduled maintenance windows minimize service disruption while ensuring system reliability and security compliance.

**Software Updates and Bug Fix Procedures**

Updates follow a structured process including development testing, staging deployment, production deployment with rollback capabilities. Critical bug fixes receive expedited deployment procedures while maintaining system stability. Version control enables easy rollback for problematic updates.

**Issue Resolution and Escalation Process**

Support procedures include issue categorization by severity, automated monitoring alerts for system failures, and escalation procedures for critical issues. Documentation maintenance ensures knowledge preservation and efficient problem resolution. Regular system health checks proactively identify potential issues before customer impact.

**REVISION HISTORY**

|  |  |
| --- | --- |
| **Date** | **Description of Changes** |
| August 26, 2025 | Initial creation of the System Design Document detailing the architecture, components, data flows, interfaces, and design considerations for the Cinema Booking System. |
| September 10, 2025 | Document formatting improvements and table structure refinements. |

Table 1. Document Revision History