**SM CINEMA BOOKING SYSTEM**

A Technical Documentation 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.

October 2025

**TECHNICAL DOCUMENTATION**

**INTRODUCTION**

This technical documentation serves as a comprehensive reference guide for the Web-Based Cinema Booking System developed for SM Cinema Grand Central. The document is intended for developers, system administrators, and technical stakeholders who need to understand the system's architecture, components, and implementation details. It provides the necessary information for system maintenance, future development, troubleshooting, and potential scaling efforts.

The application is built using modern web technologies with a clear separation between frontend and backend components. The frontend provides a responsive user interface accessible from desktop and mobile browsers, while the backend handles business logic, data management, and API services. The system integrates with cloud-based services for database storage, media management, and caching to ensure reliable performance and scalability within the constraints of a free-tier deployment.

**Scope of the Technical Documentation**

This documentation covers the complete technical implementation of the Cinema Booking System as deployed in its production environment. The scope includes detailed descriptions of system architecture and component relationships, frontend implementation using React and supporting libraries, backend API development with Django REST Framework, database schema and data models, third-party service integrations, deployment configuration and infrastructure, and authentication and authorization mechanisms.

The documentation focuses on the technical aspects of the system rather than user-facing functionality or business processes. It provides sufficient detail for technical personnel to understand how the system operates, how components interact, and how to perform common maintenance tasks.

**SYSTEM OVERVIEW**

The Cinema Booking System follows a client-server architecture with clear separation between presentation, business logic, and data layers. The system is built on a three-tier architecture: the React-based frontend handles the presentation tier, the Django REST Framework backend manages the application tier, and PostgreSQL provides the data tier. Communication between frontend and backend occurs through RESTful API calls using JSON-formatted data.

The frontend is a single-page application that runs in the user's browser, handling interface rendering and user interactions. The backend processes all business logic, data validation, and database operations. External services handle specialized functions: Cloudinary manages media storage and delivery, Upstash Redis provides caching for performance optimization, and Supabase hosts the PostgreSQL database with managed backup and connection pooling.

**High-Level Components and Their Interactions**

The system comprises several interconnected components that work together to deliver complete functionality. Understanding how these components interact is essential for maintaining and extending the system.

**Frontend Application:**

The React-based frontend serves as the primary user interface for both customers and administrators accessing public-facing pages. It consists of multiple page components including the homepage displaying movie listings, movie detail pages showing screening information, seat selection interfaces for choosing specific seats, booking forms for capturing customer information, and various administrative views accessible through protected routes.

The frontend manages local state for user interactions such as seat selections and form inputs. It communicates with the backend by making HTTP requests to API endpoints using the Axios library. Responses from the backend are processed and used to update the user interface dynamically. The frontend also implements client-side routing using React Router, allowing seamless navigation between different pages without full page reloads.

**Backend API:**

The Django REST Framework backend provides a comprehensive API that handles all server-side operations. It exposes RESTful endpoints for retrieving movie listings and details, fetching showtime information, checking seat availability, creating and managing reservations, and performing administrative operations on movies and schedules.

The backend implements business logic to validate user inputs, enforce booking rules, prevent double-booking through transaction management, and maintain data consistency across concurrent requests. It serves as the intermediary between the frontend and the database, translating API requests into database queries and formatting query results into JSON responses suitable for frontend consumption.

Authentication and authorization are handled at the backend level, with Django's built-in authentication system protecting administrative endpoints. The backend also manages integration with external services, communicating with Cloudinary for media uploads and Upstash Redis for caching frequently accessed data.

**Database Layer:**

The PostgreSQL database hosted on Supabase stores all persistent application data. The database schema includes tables for movies containing film information and metadata, showtimes representing screening schedules, reservations tracking customer bookings, and seats defining theater layout and availability status.

Relationships between tables are enforced through foreign key constraints, ensuring referential integrity. For example, showtimes are linked to specific movies, and reservations reference both showtimes and selected seats. The database handles concurrent access through transaction isolation levels and locking mechanisms, preventing conflicts when multiple users attempt to book the same seats simultaneously.

**External Services:**

Cloudinary serves as the content delivery network for all media assets, primarily movie posters and promotional images. When administrators upload images through the admin interface, files are transmitted to Cloudinary's servers where they are stored and optimized. The service returns URLs that the backend stores in the database, and these URLs are used by the frontend to display images efficiently.

Upstash Redis provides a caching layer that reduces database load and improves response times for frequently accessed data. The backend caches movie listings and showtime information, serving subsequent requests from cache until the data is updated or the cache expires.

**Component Interaction Flow:**

A typical user interaction follows a predictable flow through the system components. When a customer accesses the homepage, the frontend makes a GET request to the backend's movie listing endpoint. The backend checks the Redis cache for recent movie data. If cached data exists and is still valid, it is returned immediately. Otherwise, the backend queries the PostgreSQL database, retrieves current movie information, stores it in the cache, and returns it to the frontend. The frontend processes the JSON response and renders movie cards with information and poster images loaded from Cloudinary.

When a customer selects a movie and chooses a showtime, the frontend requests seat availability data from the backend. The backend queries the database for the specific showtime's seating information, checking which seats are already reserved. This data is returned to the frontend, which renders an interactive seat grid showing available and reserved seats in different colors.

As the customer selects seats, the frontend maintains this selection in local state without immediately communicating with the backend. When the customer proceeds to book, the frontend sends a POST request to the reservation endpoint with the selected seat information and customer details. The backend validates the request, checks that selected seats are still available, creates a database transaction to reserve the seats, and stores the reservation record. Upon successful completion, the backend returns a confirmation response that the frontend displays to the user.

**Deployment Architecture**

The deployment architecture leverages multiple cloud platforms to host different components of the system, taking advantage of free-tier offerings while maintaining separation of concerns and service reliability.

**Frontend Deployment:**

The React application is deployed on Vercel, a platform optimized for frontend frameworks and static site hosting. The deployment process is automated through GitHub integration, where pushing code to the main branch triggers an automatic build and deployment. Vercel runs the build command to compile the React application using Vite, generating optimized static files. These files are then distributed across Vercel's content delivery network, ensuring fast load times for users regardless of geographic location.

Environment variables configured in the Vercel dashboard provide the frontend with the backend API URL and other configuration values needed at runtime. The platform handles HTTPS certificates automatically, ensuring secure connections for all users.

**Backend Deployment:**

The Django REST Framework backend is deployed on Render as a web service. Similar to the frontend, deployment is automated through GitHub integration. When code is pushed to the main branch, Render pulls the latest changes, installs Python dependencies, runs database migrations, collects static files, and starts the application server using Gunicorn.

The backend runs in a containerized Linux environment managed by Render. Environment variables stored in Render's dashboard provide sensitive configuration including database connection strings, API keys for external services, and Django's secret key. The free tier includes automatic sleep after periods of inactivity, which may cause initial requests after idle periods to take longer as the service restarts.

**Database Hosting:**

The PostgreSQL database is hosted on Supabase, a managed database platform that handles server maintenance, backups, and connection pooling automatically. The database is configured with appropriate connection limits and security settings. Connection to the database from the backend is established using the connection URL stored as an environment variable in Render.

Supabase provides a web-based interface for direct database administration when needed, though most database operations occur programmatically through the Django ORM. The platform maintains automated backups and provides point-in-time recovery capabilities in case of data issues.

**Media Storage:**

Cloudinary hosts all uploaded media files in cloud storage with global distribution through their CDN. Files are uploaded directly from the backend when administrators add or update movies with poster images. Cloudinary returns permanent URLs for uploaded files, which are stored in the database and used by the frontend to display images. The service handles image optimization, format conversion, and responsive delivery based on the requesting device.

**INSTALLATION GUIDE**

**System Requirements**

**Hardware Requirements:**

* Processor: Intel Core i3 or AMD Ryzen 3 (or equivalent)
* Memory: Minimum 4GB RAM (8GB recommended for development)
* Storage: At least 2GB of free disk space for project files and dependencies
* Network: Stable internet connection for downloading dependencies and accessing external services

**Software Requirements:**

Operating System:

* Windows 10/11, macOS 10.15+, or Linux (Ubuntu 20.04+ recommended)
* Required Software:
* Node.js: Version 18.x or higher
* Python: Version 3.12.x
* Web Browser: Latest version of Chrome, Firefox, Edge, or Safari
* Package Managers:
* npm (comes with Node.js)
* pip (comes with Python)

**External Service Accounts:**

The following cloud service accounts are required for full system functionality:

* Vercel account (for frontend hosting)
* Render account (for backend hosting)
* Supabase account (for PostgreSQL database)
* Upstash account (for Redis caching)
* Cloudinary account (for media storage)

All services offer free-tier plans sufficient for development and small-scale deployment.

**Step-by-Step Installation Instructions**

**Prerequisites Setup**

1. **Install Node.js and npm**

Download and install Node.js from the official website (https://nodejs.org/). The installation includes npm automatically.

Verify installation by opening a terminal or command prompt and running:

* node --version
* npm –version

1. **Install Python**

Download and install Python 3.12 from the official website (https://www.python.org/downloads/).

During installation on Windows, ensure "Add Python to PATH" is checked.

Verify installation:

* python --version
* pip –version

**Backend Dependencies Installation**

Navigate to the backend directory and create a virtual environment to isolate project dependencies:

* cd backend
* python -m venv venv

**Activate the virtual environment:**

* venv\Scripts\activate

**Install required Python packages:**

* pip install -r requirements.txt

The requirements.txt file includes the following dependencies:

* Django: Web framework for building the backend API
* Django REST Framework: Toolkit for building RESTful APIs
* psycopg2-binary: PostgreSQL database adapter for Python
* gunicorn: Production-grade WSGI HTTP server
* django-cors-headers: Handles Cross-Origin Resource Sharing (CORS)
* python-decouple: Manages environment variables
* cloudinary: SDK for media storage and management
* redis: Python client for Redis caching
* requests: HTTP library for external API calls

**Frontend Dependencies Installation**

Open a new terminal window and navigate to the frontend directory:

* cd frontend
* Install required npm packages:
* npm install

This will install all dependencies listed in package.json:

**Main Dependencies:**

* react (19.1.0): JavaScript library for building user interfaces
* react-dom (19.1.0): React package for DOM rendering
* react-router-dom (7.8.2): Routing library for React applications
* axios (1.10.0): Promise-based HTTP client for API requests
* tailwindcss (4.1.11): Utility-first CSS framework for styling
* react-icons (5.5.0): Icon library for React components

**External Service Configuration**

Cloudinary Setup: Sign up at https://cloudinary.com and navigate to the dashboard. Locate your Cloud Name, API Key, and API Secret for backend configuration.

Upstash Redis Setup: Sign up at https://upstash.com and create a new Redis database. Select the REST API option and copy the provided REST URL.

Deployment Platforms: Create accounts on Vercel (https://vercel.com) and Render (https://render.com) for hosting the application.

**Running the Application Locally**

**Start the Backend Server:**

* python manage.py runserver

The backend API will be accessible at http://localhost:8000. The Django admin panel is available at http://localhost:8000/admin.

**Start the Frontend Development Server:**

* npm run dev

The frontend application will be accessible at http://localhost:5173.

**CONFIGURATION GUIDE**

**Detailed Instructions for Configuring the Software**

The Cinema Booking System requires proper configuration of both frontend and backend components to function correctly. Configuration is managed through environment variables stored in .env files.

**Backend Configuration Process**

Navigate to the backend directory and create a new file named .env in the root folder. Add the following configuration variables with your actual values: SECRET\_KEY, DEBUG, ALLOWED\_HOSTS, DATABASE\_URL, CLOUDINARY\_CLOUD\_NAME, CLOUDINARY\_API\_KEY, CLOUDINARY\_API\_SECRET, USE\_CLOUDINARY, UPSTASH\_REDIS\_URL, and FRONTEND\_URL.

**Frontend Configuration Process**

Navigate to the frontend directory and create a new file named .env in the root folder.

Add the following configuration variables: VITE\_API\_URL, VITE\_FAVICON\_URL, VITE\_LOGO\_URL, and VITE\_REF\_COMPANY.

**Configuration File Formats and Parameters**

**Backend Configuration Parameters**

* SECRET\_KEY: Django's cryptographic signing key used for security features. Generate a secure random string of at least 50 characters. Never share this publicly.
* DEBUG: Controls debug mode. Set to True during development and False in production.
* ALLOWED\_HOSTS: Comma-separated list of domain names permitted to access the application. Include localhost and 127.0.0.1 for development.
* DATABASE\_URL: Complete PostgreSQL connection string from Supabase in the format postgresql://username:password@hostname:port/database\_name
* CLOUDINARY\_CLOUD\_NAME: Your Cloudinary account's cloud name from the dashboard.
* CLOUDINARY\_API\_KEY: API key for Cloudinary authentication.
* CLOUDINARY\_API\_SECRET: API secret for Cloudinary. Keep this secure.
* USE\_CLOUDINARY: Set to True to use cloud storage or False for local storage.
* UPSTASH\_REDIS\_URL: REST API URL for Redis caching service. Optional but improves performance.
* FRONTEND\_URL: Complete URL where frontend is hosted. Used for CORS configuration.

**Frontend Configuration Parameters**

* VITE\_API\_URL: Base URL for all backend API requests. Include the full path with API version.
* VITE\_FAVICON\_URL: URL or path to site favicon. Optional.
* VITE\_LOGO\_URL: URL or path to company logo. Optional.
* VITE\_REF\_COMPANY: Company name displayed throughout the interface.

**Best Practices for Customization**

**Security Best Practices**

* Never commit .env files to version control
* Use different SECRET\_KEY values for development and production
* Always set DEBUG to False in production
* Use strong passwords for database connections
* Rotate API keys periodically

**Performance Optimization**

* Enable Redis caching to reduce database load
* Use Cloudinary for automatic image optimization
* Configure appropriate cache timeouts based on data update frequency

**Development vs Production Configuration**

* Maintain separate .env files for different environments
* Development should use localhost URLs and enable debug mode
* Production must use actual domain names and disable debug mode
* Test with production-like settings before deploying

**Environment Variable Management**

* Document all required variables in a .env.example file
* Use descriptive variable names
* Group related variables together
* Add comments explaining usage

**CORS Configuration**

* Ensure FRONTEND\_URL exactly matches your frontend deployment URL
* Do not include trailing slashes in URLs
* Verify protocol matches (http vs https)

**Database Configuration**

* Use PostgreSQL version 12 or higher
* Monitor connection usage on free-tier plans
* Always run migrations before starting the application

**Troubleshooting Configuration Issues**

* Check that all required environment variables are present
* Verify proper formatting of all values
* Test database connectivity using Django shell
* Check browser developer tools for CORS error details

**API DOCUMENTATION**

The Cinema Booking System exposes a RESTful API for communication between frontend and backend. The API is versioned and follows standard HTTP methods. All endpoints accept and return JSON-formatted data unless specified otherwise.

**Authentication and Authorization Requirements**

**Authentication Process**

* The API uses token-based authentication for protected endpoints
* After successful login, server returns an authentication token
* Token must be included in Authorization header as: Token your-token-here
* Tokens remain valid until logout or expiration

**Token Storage**

* Frontend stores tokens in localStorage or sessionStorage
* Tokens are automatically attached to requests through Axios interceptors

**Authorization Levels**

* Public Endpoints: Accessible without authentication (movie listings, showtimes, cinema details)
* Authenticated Endpoints: Require valid token (booking creation, user profile)
* Staff Endpoints: Require staff privileges (content creation, user management)
* Superuser Endpoints: Require superuser privileges (full system access)

**API Endpoints**

**User Management Endpoints**

**POST /users/create/**

* Description: Register a new user account
* Authentication: Public
* Request: username, email, password, first\_name, last\_name
* Response: User object with ID and account details

**POST /users/login/**

* Description: Authenticate user and receive token
* Authentication: Public
* Request: username, password
* Response: Authentication token and user information

**POST /users/logout/**

* Description: Invalidate current token
* Authentication: Required
* Response: Success message

**GET /users/me/**

* Description: Get current authenticated user information
* Authentication: Required
* Response: Current user details

**GET /users/**

* Description: List all users
* Authentication: Required (Superuser only)
* Response: Array of user objects

**GET /users/{user\_id}/**

* Description: Get specific user details
* Authentication: Required (Superuser only)
* Response: User object

**PUT /users/{user\_id}/update/**

* Description: Update user information
* Authentication: Required (Superuser only)
* Request: User fields to update
* Response: Updated user object

**DELETE /users/{user\_id}/delete/**

* Description: Delete user account
* Authentication: Required (Superuser only)
* Response: No content

**PATCH /users/{user\_id}/toggle-active/**

* Description: Toggle user active status
* Authentication: Required (Superuser only)
* Response: Updated user status

**PUT /users/me/update/**

* Description: Update own user information
* Authentication: Required (Staff only)
* Request: User fields to update
* Response: Updated user object

**Movie Endpoints**

**GET /movies/**

* Description: List all active movies
* Authentication: Public
* Query Parameters: detail (summary or full), genre (filter by genre ID)
* Response: Array of movie objects

**GET /movies/{id}/**

* Description: Get detailed movie information
* Authentication: Public
* Query Parameters: detail (summary or full)
* Response: Single movie object with full details

**GET /movies/search/**

* Description: Search movies by title
* Authentication: Public
* Query Parameters: search (search query string)
* Response: Array of matching movies

**POST /movies/**

* Description: Create new movie
* Authentication: Required (Staff only)
* Request: title, description, genre, duration, rating, release\_date, poster file
* Response: Created movie object

**PUT /movies/{id}/**

* Description: Update movie (full update)
* Authentication: Required (Staff only)
* Request: All movie fields
* Response: Updated movie object

**PATCH /movies/{id}/**

* Description: Partially update movie
* Authentication: Required (Staff only)
* Request: Fields to update
* Response: Updated movie object

**DELETE /movies/{id}/**

* Description: Delete movie
* Authentication: Required (Staff only)
* Response: No content

**Genre Endpoints**

**GET /genres/**

* Description: List all genres
* Authentication: Public
* Query Parameters: include\_count (include movie count)
* Response: Array of genre objects

**GET /genres/{id}/**

* Description: Get genre details
* Authentication: Public
* Response: Single genre object

**GET /genres/{id}/movies/**

* Description: List movies in specific genre
* Authentication: Public
* Query Parameters: limit (max movies), detail (summary or full)
* Response: Array of movie objects

**POST /genres/**

* Description: Create new genre
* Authentication: Required (Staff only)
* Request: name, description
* Response: Created genre object

**PUT /genres/{id}/**

* Description: Update genre (full update)
* Authentication: Required (Staff only)
* Request: name, description
* Response: Updated genre object

**PATCH /genres/{id}/**

* Description: Partially update genre
* Authentication: Required (Staff only)
* Request: Fields to update
* Response: Updated genre object

**DELETE /genres/{id}/**

* Description: Delete genre
* Authentication: Required (Staff only)
* Response: No content

**Showtime Endpoints**

**GET /showtimes/**

* Description: List all showtimes
* Authentication: Public
* Query Parameters: detail (summary or full), movie (filter by movie ID)
* Response: Array of showtime objects

**GET /showtimes/{id}/**

* Description: Get showtime details
* Authentication: Public
* Query Parameters: detail (summary or full)
* Response: Single showtime object

**POST /showtimes/**

* Description: Create new showtime
* Authentication: Required (Staff only)
* Request: movie, cinema, screening\_room, start\_time, price
* Response: Created showtime object

**PUT /showtimes/{id}/**

* Description: Update showtime (full update)
* Authentication: Required (Staff only)
* Request: All showtime fields
* Response: Updated showtime object

**PATCH /showtimes/{id}/**

* Description: Partially update showtime
* Authentication: Required (Staff only)
* Request: Fields to update
* Response: Updated showtime object

**DELETE /showtimes/{id}/**

* Description: Delete showtime
* Authentication: Required (Staff only)
* Response: No content

**Cinema Endpoints**

**GET /cinemas/**

* Description: List all cinemas
* Authentication: Public
* Query Parameters: detail (summary or full)
* Response: Array of cinema objects

**GET /cinemas/{id}/**

* Description: Get cinema details
* Authentication: Public
* Query Parameters: detail (summary or full)
* Response: Single cinema object

**GET /cinemas/{cinema\_id}/showtimes/**

* Description: List showtimes for specific cinema
* Authentication: Public
* Query Parameters: date (YYYY-MM-DD), movie (movie ID)
* Response: Array of showtime objects

**POST /cinemas/**

* Description: Create new cinema
* Authentication: Required (Staff only)
* Request: name, location, description
* Response: Created cinema object

**PUT /cinemas/{id}/**

* Description: Update cinema (full update)
* Authentication: Required (Staff only)
* Request: All cinema fields
* Response: Updated cinema object

**PATCH /cinemas/{id}/**

* **Description: Partially update cinema**
* **Authentication: Required (Staff only)**
* **Request: Fields to update**
* **Response: Updated cinema object**

**DELETE /cinemas/{id}/**

* Description: Delete cinema
* Authentication: Required (Staff only)
* Response: No content

**Screening Room Endpoints**

**GET /rooms/**

* Description: List all screening rooms
* Authentication: Public
* Response: Array of screening room objects

**GET /rooms/{id}/**

* Description: Get screening room details
* Authentication: Public
* Response: Single screening room object

**POST /rooms/**

* Description: Create new screening room
* Authentication: Required (Staff only)
* Request: name, cinema, total\_rows, seats\_per\_row
* Response: Created screening room object

**PUT /rooms/{id}/**

* Description: Update screening room (full update)
* Authentication: Required (Staff only)
* Request: All room fields
* Response: Updated screening room object

**PATCH /rooms/{id}/**

* Description: Partially update screening room
* Authentication: Required (Staff only)
* Request: Fields to update
* Response: Updated screening room object

**DELETE /rooms/{id}/**

* Description: Delete screening room
* Authentication: Required (Staff only)
* Response: No content

**Booking Endpoints**

**POST /bookings/**

* Description: Create new booking with instant confirmation
* Authentication: Public
* Request: showtime, customer\_name, customer\_email, seats (array of row and number)
* Response: Complete booking object with reference number

**GET /bookings/{booking\_reference}/**

* Description: Get booking details by reference
* Authentication: Public
* Response: Complete booking object

**GET /bookings/{booking\_reference}/download-ticket/**

* Description: Download booking ticket as PDF
* Authentication: Public
* Response: PDF file

**GET /bookings/overview/**

* Description: Get booking statistics
* Authentication: Required (Staff only)
* Response: Summary statistics object

**GET /bookings/summary/**

* Description: Get detailed booking analytics
* Authentication: Required (Staff only)
* Response: Detailed analytics object

**Common Response Codes**

* 200 OK: Request successful
* 201 Created: Resource successfully created
* 204 No Content: Request successful, no response body
* 400 Bad Request: Invalid request format
* 401 Unauthorized: Authentication required
* 403 Forbidden: Lacking required permissions
* 404 Not Found: Resource does not exist
* 500 Internal Server Error: Server-side error

**Error Response Format**

All errors return a consistent format with error message, details, and field-specific errors when applicable.

**USER MANUAL**

**Getting Started**

No installation required. Open any modern web browser and navigate to the Cinema Booking System URL. The system works on Windows, macOS, Linux, iOS, and Android.

Browser Requirements: Chrome, Firefox, Safari, or Edge (latest versions) System Requirements: 4GB RAM minimum, stable internet connection

**Customer interface**

Movie Browsing: The main page displays available movies with posters, titles, genres, and showtimes. Use the date selector to browse movies for different dates. Click on a movie to see detailed information and available screening times.

Seat Selection: After selecting a showtime, the theater layout appears showing available and reserved seats. Green seats are available, gray or red seats are reserved. Click seats to select them. Selected seats highlight in yellow or blue. Review your selection in the summary panel.

**Booking Process:**

1. Browse and select a movie
2. Choose your preferred showtime and date
3. Click on available seats to select them
4. Review selected seats and movie details
5. Click confirm to complete reservation
6. Receive booking confirmation with details

Navigation: Use the menu bar to access different sections. Home button returns to movie listing. Search bar allows finding specific movies. Mobile devices show a hamburger menu for compact navigation.

**Admin interface**

Admin Login: Navigate to admin portal and enter credentials. Successful authentication opens the admin dashboard with full management tools.

Movie Management: Add new movies by entering title, description, genre, runtime, rating, and poster image URL. Edit existing movies to update any information. Delete movies to remove them from the system. Mark movies as active or inactive to control visibility.

Showtime Management: Create new showtimes by selecting a movie, date, and time. Update existing showtimes to change scheduling. Mark showtimes as available or unavailable for booking. View reservation status for each showtime.

Reservation Monitoring: View all current reservations in a list format. Filter reservations by date, movie, or showtime. Click individual reservations to see detailed information including seat numbers and booking time. Monitor theater capacity and occupancy rates.

**Frequently asked questions**

Q: Do I need an account to book? A: No, customer accounts are not required. Simply browse and book without registration.

Q: Can I modify my booking after confirmation? A: Contact customer support for booking modifications. System does not allow self-service changes.

Q: Does the system accept online payments? A: This version focuses on seat reservation only. Payment is handled separately by the cinema.

Q: Is the system mobile-friendly? A: Yes, the system is fully responsive and works on all devices and screen sizes.

Q: What if I encounter an error? A: Refresh the page and try again. Clear browser cache if issues persist. Contact support if problems continue.

**TROUBLESHOOTING GUIDE**

**Common issues**

Seat Cannot Be Selected Problem: Selected seat shows as available but cannot be clicked. Cause: Another customer just reserved the seat. Real-time availability updates as others book. Solution: Refresh the page to see current seat status. Select alternative available seats.

Booking Confirmation Fails Problem: Seat selection completes but confirmation does not process. Cause: Network interruption or temporary backend issue. Solution: Wait 30 seconds and refresh the page. Attempt booking again. Try different browser if issue persists.

Page Not Loading Problem: Page displays blank or takes very long to load. Cause: Slow internet connection or backend service starting up (cold start). Solution: Wait 30-60 seconds for page to fully load. Refresh the page. Clear browser cache and cookies. Try different browser.

Admin Cannot Login Problem: Admin credentials rejected despite correct password. Cause: Account not yet activated or permissions not set. Browser cache issues. Solution: Verify credentials are entered correctly. Clear browser cookies and try again. Contact system administrator to verify account activation.

Movie Not Appearing Problem: New movie added by admin does not show in customer interface. Cause: Movie marked as inactive. Browser cache displaying old data. Solution: Verify movie is marked as active in admin settings. Refresh customer website to load new data. Clear browser cache.

**General troubleshooting**

Clear Browser Cache: Access browser settings, find privacy or data section, select clear browsing data, confirm.

Update Browser: Download latest version from official browser website. Restart browser after updating.

Check Internet Connection: Test connection with another website. Verify adequate bandwidth available.

Restart Browser: Close browser completely and reopen. Navigate back to system URL.

Test Different Browser: Try Chrome, Firefox, Safari, or Edge to isolate browser-specific issues.

**Performance issues**

Slow Loading: Clear cache and cookies. Close unused browser tabs. Use wired internet instead of WiFi. Access during lower-traffic times.

High Memory Usage: Close other applications. Restart browser. Close multiple open tabs.

Application Crashes: Force close browser. Restart device. Update browser to latest version.

**Support contact**

Email: customercare@smcinema.com Phone: (+632) 8470 2222 Address: J.W. Diokno Boulevard, Pasay City, Metro Manila, Philippines Response Time: Within 24 business hours

**TROUBLESHOOTING GUIDE**

**Project structure**

root/

* frontend-vite/ (React application)
* backend-django/ (Django API)
* docs/ (Documentation)

Frontend: src/ contains components, pages, services, utilities. public/ contains static files. dist/ contains production build.

Backend: config/ contains settings. apps/ contains features. manage.py is Django utility.

**Code organization**

Frontend Components: Movie cards display movie information. Seat selection component handles theater layout and seat interaction. Navigation component manages menu. Pages organize complete screens.

Backend Models: Movie represents cinema films. Showtime represents screening times. Seat represents individual seats. Reservation represents customer bookings.

Services: Movie service handles movie API calls. Booking service handles reservations. Seat service manages seat availability.

**Key functions**

get\_available\_movies: Retrieves active movies with showtimes for specified date.

get\_theater\_layout: Returns seat availability for specific showtime.

create\_reservation: Processes seat selection and creates booking. Prevents double-booking using database locking.

get\_showtimes\_for\_movie: Returns all showtimes for specified movie and date.

update\_movie\_status: Admin function to activate or deactivate movies.

**Naming conventions**

Variables and Functions: Use camelCase format (example: getMovieDetails, userSelection)

Classes: Use PascalCase (example: MovieSerializer, ReservationViewSet)

Constants: Use UPPERCASE\_WITH\_UNDERSCORES (example: MAX\_SEATS)

Database Fields: Use lowercase\_with\_underscores (example: movie\_title, seat\_availability)

API Endpoints: Use lowercase-with-hyphens (example: /api/movies, /api/reservations)

**Comments and documentation**

Add clear comments explaining why code does something. Document all functions with purpose, parameters, and return values. Use inline comments for complex logic. Frontend uses JSDoc format. Backend uses Python docstrings.

INDENTATION

Use 2 spaces consistently. Never use tabs. Maximum line length 100 characters. Wrap long lines appropriately.

TECHNOLOGIES

Frontend: React 18.x, Vite, React Router, Axios, Tailwind CSS, Lucide React

Backend: Django REST Framework, PostgreSQL, Cloudinary, Upstash Redis, Gunicorn

Languages: JavaScript, Python, HTML, CSS

**Environment variables**

Frontend: VITE\_API\_URL (backend URL), VITE\_LOGO\_URL (logo image), VITE\_FAVICON\_URL (browser icon), VITE\_REF\_COMPANY (cinema name)

Backend: DATABASE\_URL (Supabase), SECRET\_KEY (Django key), DEBUG (true/false), ALLOWED\_HOSTS (domain list), CLOUDINARY credentials, UPSTASH\_REDIS\_URL (cache)

**TESTING DOCUMENTATION**

**Test plan objectives**

Verify all features work correctly according to requirements. Identify and document defects. Ensure acceptable performance. Validate responsive design works across devices. Confirm cross-browser compatibility.

**Testing strategy**

* Unit Testing: Test individual functions and components in isolation.
* Integration Testing: Test communication between components and services.
* System Testing: Test complete workflows from start to finish.
* Performance Testing: Monitor response times and resource usage.
* Compatibility Testing: Test across Chrome, Firefox, Safari, Edge and various devices.

**Test scope**

Included Features: Movie browsing and search, showtime display and filtering, seat selection and availability, reservation creation and confirmation, admin dashboard access, movie and showtime management, responsive design functionality.

Excluded: Payment processing (not in scope), SMS notifications (not in scope), loyalty programs (not in scope), customer accounts (not in scope).

**Test cases**

**TC-001: Movie Browsing**

* Objective: Verify movies display with correct information
* Steps: Open system, view movie list, click movie details
* Expected: Movie information displays accurately with title, genre, description
* Status: Pass

**TC-002: Showtime Selection**

* Objective: Verify showtimes display correctly
* Steps: Select movie, view available showtimes, select date
* Expected: Showtimes display for selected date, times are accurate
* Status: Pass

**TC-003: Seat Selection**

* Objective: Verify seat selection interface works correctly
* Steps: Select showtime, view theater layout, click available seat
* Expected: Seat highlights when clicked, status updates correctly
* Status: Pass

**TC-004: Complete Booking**

* Objective: Verify complete booking workflow
* Steps: Select movie, showtime, seats, confirm reservation
* Expected: Reservation completes successfully, confirmation displays
* Status: Pass

**TC-005: Admin Login**

* Objective: Verify admin authentication works
* Steps: Navigate to admin page, enter credentials, submit
* Expected: Admin dashboard displays upon successful login
* Status: Pass

**TC-006: Add Movie**

* Objective: Verify admin can add movies
* Steps: Go to admin, add new movie with details, save
* Expected: Movie appears in customer interface, details are accurate
* Status: Pass

**TC-007: Responsive Design**

* Objective: Verify interface works on mobile devices
* Steps: Access system on phone and tablet, test functionality
* Expected: Layout adapts properly, buttons clickable, interface usable
* Status: Pass

**TC-008: Cross-Browser**

* Objective: Verify system works in different browsers
* Steps: Test in Chrome, Firefox, Safari, Edge
* Expected: Functionality consistent across all browsers
* Status: Pass

**Test results summary**

Total Test Cases: 8 Passed: 8 Failed: 0 Blocked: 0 Success Rate: 100%

**Defect reports**

No critical defects identified. System functions correctly with all core features operational. Minor performance optimizations noted for future releases but not blocking deployment.

**MAINTENANCE GUIDE**

This guide outlines procedures for maintaining, updating, and managing the Cinema Booking System throughout its operational life.

**Regular maintenance tasks**

* Daily: Monitor system performance and logs. Review error reports. Check for critical security alerts.
* Weekly: Backup all data and configurations. Review user feedback and support tickets. Monitor theater occupancy and booking rates.
* Monthly: Review and apply security patches. Assess dependency updates. Test backup restoration procedures.

**Version control**

Repository: GitHub manages all code versions. Main branch contains production-ready code. Develop branch contains development work. Feature branches for new features. Hotfix branches for emergency fixes.

Branching Strategy: Create feature branches from develop for new work. Create hotfix branches from main for urgent issues. Merge back to both main and develop after fixes.

Commit Messages: Use clear descriptive messages. Format: type: description (example: feat: add seat filtering, fix: resolve booking timeout).

Pull Requests: Require code review before merging. Test all changes before submission. Document changes in pull request description.

**Release management**

Versioning: Follow Semantic Versioning format MAJOR.MINOR.PATCH. Major version for breaking changes. Minor version for new features. Patch version for bug fixes.

Release Cycle: Plan features and fixes for release. Develop features in feature branches. Test thoroughly before release. Create release branch for final testing. Merge to main and tag with version number.

Release Checklist: All tests passing. Documentation updated. Version number bumped. Release notes written. Backup created. Deployment plan reviewed.

**Bug fix process**

Report: Create issue ticket with description and priority. Assign to developer for investigation.

Fix: Create hotfix branch from main. Develop and test fix. Submit for code review.

Deploy: Merge to main and develop branches. Deploy fix to production. Monitor for issues.

**Deployment procedures**

Pre-Deployment: Run full test suite. Verify all tests pass. Create backup of database. Review deployment plan.

Deployment: Deploy backend to Render. Deploy frontend to Vercel. Verify services are running. Test core functionality.

Post-Deployment: Monitor system metrics. Check error logs. Verify database integrity. Perform smoke tests.

**Rollback plan**

If critical issues occur after deployment: Revert to previous version from version control. Restore database from backup. Redeploy previous stable version. Investigate issue in non-production environment.

**Security updates**

Apply immediately for security vulnerabilities. Test in staging before production. Deploy during low-traffic times. Monitor closely after deployment

.

**Critical updates**

Deploy within 24-48 hours for critical bugs. Test thoroughly before deployment. Have rollback plan ready.

**Regular updates**

Schedule during maintenance windows. Deploy during non-peak hours. Notify users of planned maintenance. Have support staff available during update.

**Software updates**

Monitor dependency updates monthly. Test updates in staging environment. Update production after verification. Document all updates made.

**Monitoring**

Track response times and performance metrics. Monitor error rates and exceptions. Alert on performance degradation. Review logs regularly for issues

.

**Documentation maintenance**

Update manuals when features change. Keep troubleshooting guide current. Maintain version history. Review documentation annually. Archive old versions.