**SM CINEMA BOOKING SYSTEM**

A Maintenance 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

**MAINTENANCE DOCUMENTATION**

**INTRODUCTION**

The Web-Based Cinema Booking System for SM Cinema Grand Central is a full-stack web application designed to simplify the movie ticket reservation process for customers and provide operational management tools for cinema administrators. The system allows users to browse movie listings, select screening times, and choose specific seats in real-time while providing administrators with tools to manage movie schedules and monitor reservation activity.

System maintenance is critical to ensuring the long-term reliability, security, and performance of the application. Regular maintenance activities help prevent potential issues before they escalate, address bugs and errors as they are discovered, and adapt the system to changes in technology or user requirements. Without consistent maintenance, the system risks degradation in performance, security vulnerabilities, and decreased user satisfaction over time.

The scope of this maintenance documentation covers several key areas. Corrective maintenance addresses the identification and resolution of bugs, errors, and functional issues reported by users or discovered through monitoring. Perfective maintenance focuses on enhancements and optimizations that improve system performance, user experience, or code quality based on feedback and usage patterns. The maintenance activities outlined in this document are designed to keep the Cinema Booking System operating smoothly and meeting the needs of both customers and administrative staff.

**MAINTENANCE PLAN**

The maintenance strategy for the Cinema Booking System is structured around two primary maintenance types, each serving distinct purposes in the system's lifecycle.

**Corrective Maintenance**

Corrective maintenance involves identifying and fixing bugs, errors, and defects that affect system functionality or user experience. This includes addressing issues reported through GitHub issue tracking, problems discovered during routine monitoring, and errors logged by the application itself. Corrective maintenance is reactive in nature, responding to specific problems as they arise. Issues are prioritized based on severity, with critical bugs affecting core booking functionality receiving immediate attention while minor cosmetic issues are addressed during scheduled maintenance windows. The goal of corrective maintenance is to restore the system to its intended operational state as quickly as possible.

**Perfective Maintenance**

Perfective maintenance focuses on improving and optimizing the system beyond its baseline functionality. This includes performance enhancements such as query optimization, code refactoring for better maintainability, user interface improvements based on feedback, and the addition of minor features that enhance usability. Perfective maintenance is proactive and planned, taking place during scheduled maintenance periods when changes can be tested thoroughly before deployment. These improvements aim to increase user satisfaction, reduce system resource consumption, and maintain code quality as the application evolves.

The maintenance approach balances reactive response to issues with planned improvements, ensuring both immediate problems and long-term system health are addressed effectively. All maintenance activities follow a testing protocol before deployment to production, minimizing the risk of introducing new issues while resolving existing ones.

**MAINTENANCE SCHEDULE**

Regular maintenance tasks are scheduled to ensure consistent system health and reliability. The following table outlines key maintenance activities, their frequency, responsible parties, and current status.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task | Description | Frequency | Responsible Person | Status |
| Database Backup | Verify Supabase automatic backup completion and integrity | Weekly | System Developer | Ongoing |
| Security Updates | Review and apply security patches for Django, React, and dependencies | Quarterly | System Developer | Ongoing |
| Bug Fixes | Address reported issues and errors from GitHub issue tracker | As Needed | System Developer | Ongoing |
| System Performance Check | Monitor application metrics through Vercel Analytics and Render logs | Weekly | System Developer | Ongoing |
| Dependency Updates | Check for and apply non-breaking updates to libraries and frameworks | Quarterly | System Developer | Scheduled |
| Code Repository Maintenance | Review and clean up branches, update documentation in GitHub | Monthly | System Developer | Ongoing |

Table 1. Scheduled Maintenance Tasks

The maintenance schedule is designed to be manageable for a single developer while ensuring critical system functions remain reliable. Weekly tasks focus on monitoring and verification, while quarterly tasks address more substantial updates that require thorough testing. As-needed tasks like bug fixes are prioritized based on severity and impact on user experience.

**ISSUE TRACKING & BUG REPORTS**

All bugs and issues are tracked through the project's GitHub repository issue tracker. Users and administrators can report problems by creating new issues with detailed descriptions, steps to reproduce, and any relevant screenshots or error messages. Each reported issue is assigned a unique identifier, severity level, and status tracking its resolution progress.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Issue ID | Description | Severity | Reported By | Date Reported | Status |
| BUG001 | Scheduled movies showing in listings despite having no active showtimes | Medium | Admin User | 09/15/2025 | Fixed |
| BUG002 | Seat selection not refreshing properly after another user books same showtime | High | Customer | 09/18/2025 | Fixed |
| BUG003 | Movie poster images loading slowly on mobile devices | Low | Admin User | 09/20/2025 | In Progress |
| BUG004 | Admin dashboard shows incorrect count of total reservations | Medium | Admin User | 09/22/2025 | In Progress |
| BUG005 | Browser back button causes seat selection to desync from server state | Medium | Customer | 09/22/2025 | Pending |

Table 2. Issue Tracking Log

Issues are categorized by severity: Critical (system unusable), High (major functionality impaired), Medium (functionality affected but workarounds available), and Low (minor issues or cosmetic problems). Critical and High severity issues are addressed immediately, while Medium and Low priority issues are batched and resolved during scheduled maintenance windows.

**BACKUP & RECOVERY PLAN**

Data backup and recovery procedures are essential to protect against data loss and ensure business continuity in the event of system failures or data corruption.

**5.1 Backup Procedures**

The backup strategy utilizes multiple layers to protect different components of the system. Database backups are handled automatically by Supabase, which performs daily backups of the PostgreSQL database and retains them for seven days on the free tier. These backups include all movie data, showtimes, seat configurations, and reservation records. The developer verifies backup completion weekly by checking the Supabase dashboard backup status and occasionally testing restoration to a development environment.

Source code is backed up continuously through Git version control hosted on GitHub. Every code change is committed to the repository, creating a complete history of the application's development. The main branch represents the production state, while development and feature branches allow for safe experimentation without affecting the live system. GitHub serves as both a backup and a collaboration platform, ensuring code is never lost even if local development machines fail.

Media assets stored on Cloudinary are maintained by the platform's infrastructure, which includes redundancy and backup systems as part of their service. Movie posters and other uploaded images remain accessible even during local system failures. Configuration files and environment variables are documented separately in secure, encrypted storage outside of the code repository to prevent accidental exposure of sensitive credentials.

**5.2 Recovery Steps**

In the event of data loss or system failure, recovery procedures vary depending on the affected component. For database recovery, the process involves accessing the Supabase dashboard, navigating to the backup section, selecting the appropriate backup point based on the timestamp before the failure, and initiating the restore operation. This process typically completes within minutes for the database size used by this application. After restoration, the developer verifies data integrity by checking recent reservations and ensuring all movie listings appear correctly.

If the application code is corrupted or an update causes critical failures, recovery involves reverting to a previous Git commit. The developer identifies the last known working commit from the GitHub repository history, checks out that commit, and redeploys to Render and Vercel. Both platforms support rollback features that allow reverting to previous deployments without needing to manually redeploy old code versions.

For frontend deployment failures on Vercel, the platform's dashboard provides a deployment history with one-click rollback functionality. If a recent deployment breaks the frontend, the developer can instantly revert to the previous working version while investigating the cause of the failure. Similarly, Render maintains deployment history for the backend API, allowing quick rollback if server errors occur after an update. In cases of complete platform failure where hosting services themselves become unavailable, the recovery strategy involves deploying to alternative platforms.

The source code in GitHub can be deployed to different hosting services, and the database can be exported from Supabase and imported into another PostgreSQL provider. While this scenario is unlikely given the reliability of established cloud platforms, maintaining platform-agnostic code and standard technologies ensures the system is not permanently locked to any single provider.

Technical support contact information is maintained in the project documentation. For issues beyond the developer's expertise, support channels for each platform (Vercel, Render, Supabase) are available through their respective documentation and support portals.

**PERFORMANCE MONITORING**

System performance is continuously monitored to ensure acceptable user experience and identify potential issues before they impact operations. Key performance indicators are tracked through platform-provided analytics and logging tools.

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Description | Threshold | Monitoring Tool |
| Frontend Uptime | Percentage of time the Vercel-hosted frontend is accessible | > 99% | Vercel Analytics Dashboard |
| Backend Uptime | Percentage of time the Render-hosted API is responsive | > 95% | Render Metrics and Status |
| API Response Time | Average time for backend API requests to complete | < 3 seconds | Render Logs and Metrics |
| Page Load Time | Time for complete page render on frontend | < 4 seconds | Vercel Analytics and Browser DevTools |
| Database Query Performance | Average execution time for database queries | < 500ms | Supabase Dashboard Query Statistics |
| Error Rate | Percentage of requests resulting in errors | < 2% | Vercel and Render Error Logs |
| Memory Usage | Backend application memory consumption | < 450MB | Render Metrics (512MB limit on free tier) |

Table 3. Performance Metrics and Monitoring

Performance monitoring is conducted weekly through manual review of platform dashboards and automated alerts configured in Vercel and Render. The developer checks for anomalies in response times, spikes in error rates, or unusual resource consumption patterns. When performance degradation is detected, investigation focuses on recent code changes, database query efficiency, or external service dependencies that may be causing slowdowns.Render's free tier includes automatic sleep after 15 minutes of inactivity, which causes the first request after a sleep period to experience a cold start delay of 30-60 seconds. This behavior is documented and communicated to users as a trade-off for free hosting. Performance thresholds account for this limitation, with response time targets reflecting typical active-state performance rather than cold-start scenarios.

Vercel Analytics provides detailed insights into frontend performance, including Core Web Vitals metrics such as Largest Contentful Paint, First Input Delay, and Cumulative Layout Shift. These metrics help identify user experience issues related to rendering performance and interactivity. When metrics fall outside acceptable ranges, optimizations such as code splitting, lazy loading, or image optimization are implemented.

Database performance is monitored through Supabase's query statistics panel, which tracks slow queries and connection usage. If query times exceed thresholds, optimization efforts focus on adding appropriate indexes, refining query logic, or implementing caching strategies to reduce database load.

**SECURITY MEASURES**

Security is maintained through multiple layers of protection, access controls, and best practices to safeguard user data and system integrity.

**Authentication and Access Control**

The administrative dashboard is protected by Django's built-in authentication system, which requires valid credentials to access management functions. Only authorized staff members have admin accounts, and each account is secured with strong, unique passwords. Session management uses secure cookies with httpOnly and secure flags enabled in production to prevent cross-site scripting attacks. Administrative sessions expire after a period of inactivity, requiring re-authentication to continue access.

**API Security**

Backend API endpoints implement selective authentication based on sensitivity and function. Public endpoints such as movie listings and showtime viewing are accessible without authentication to allow customer browsing. Sensitive endpoints including reservation creation, modification, and administrative functions require valid authentication tokens. Django REST Framework's permission classes enforce these access rules, preventing unauthorized access to protected resources.

Request throttling is implemented using Upstash Redis to prevent abuse and denial-of-service attacks. Anonymous users are limited to a reasonable number of requests per minute, while authenticated users receive higher rate limits appropriate for legitimate use. Throttling configurations balance user experience with security, ensuring normal usage is unaffected while preventing automated attacks.

Cross-Origin Resource Sharing (CORS) is configured to allow requests only from the known frontend domain hosted on Vercel. This prevents malicious websites from making unauthorized requests to the API. The ALLOWED\_HOSTS setting in Django restricts which domains can serve the application, further limiting attack vectors.

**Data Protection**

All data transmission occurs over HTTPS, encrypting communication between clients and servers to prevent eavesdropping or man-in-the-middle attacks. Both Vercel and Render enforce HTTPS by default, automatically redirecting HTTP requests to secure connections. Supabase database connections use encrypted channels, ensuring data remains protected during transit between the application and database servers.

Environment variables containing sensitive information such as database credentials, API keys, and secret keys are stored securely in platform configuration dashboards. These variables are never committed to the code repository, preventing accidental exposure through version control history. Access to these configuration settings is restricted to authorized developers only.

**Dependency Security**

Third-party dependencies and libraries are periodically reviewed for known security vulnerabilities. GitHub's Dependabot alerts notify the developer when security advisories affect packages used in the project. Critical security updates are applied promptly, while lower-severity updates are bundled into quarterly maintenance cycles. Dependencies are kept reasonably current to benefit from security patches while avoiding breaking changes that require extensive retesting.

**Ongoing Security Practices**

Security monitoring includes regular review of application logs for suspicious activity patterns such as repeated failed login attempts, unusual API access patterns, or unexpected error rates. While automated intrusion detection is not implemented in this project scope, manual log review during weekly maintenance provides basic security oversight. Any identified security concerns are documented and addressed according to their severity and potential impact.

**CONCLUSION & RECOMMENDATIONS**

**Summary of Maintenance Tasks Completed**

The maintenance framework for the Cinema Booking System has been successfully established and implemented. Regular monitoring activities are ongoing, with weekly performance checks conducted through Vercel Analytics and Render metrics dashboards. Database backup verification is performed weekly to ensure Supabase's automatic backup system is functioning correctly. Several bugs have been identified and resolved since initial deployment, including issues with movie scheduling logic and seat selection synchronization. Code repository maintenance activities are conducted monthly, keeping the GitHub repository organized and documentation current.

Security measures have been implemented and are actively maintained, including API endpoint authentication, request throttling through Upstash Redis, and CORS configuration to prevent unauthorized access. Dependency reviews are conducted quarterly to identify and apply security updates. Performance monitoring has identified the expected cold start delays from Render's free tier but confirmed that active-state performance meets acceptable thresholds for user experience.

**Recommendations for Future Improvements**

Several enhancements could further improve system reliability and user experience if additional resources become available.

Implementing automated testing would reduce the risk of introducing bugs during maintenance activities and speed up the validation process before deploying updates. Continuous integration and deployment pipelines could streamline the update process and ensure consistent deployment procedures.

Upgrading to paid hosting tiers would eliminate cold start delays on the backend and provide additional resources for handling higher traffic volumes**.**

Paid Supabase plans offer longer backup retention periods and point-in-time recovery, which would enhance data protection capabilities. Similarly, paid Cloudinary plans would provide more storage and bandwidth for media assets as the movie library grows.

Implementing automated monitoring alerts would enable faster response to performance degradation or system errors. Currently, monitoring relies on manual dashboard reviews, which could miss time-sensitive issues. Alert systems could notify the developer immediately when error rates exceed thresholds or when services become unavailable.

Adding comprehensive logging and analytics would provide better insights into user behavior and system usage patterns. This data could inform future optimizations and help prioritize development efforts based on actual user needs. Session replay tools could help diagnose user-reported issues by showing exactly what happened during problem scenarios.

Establishing a staging environment separate from production would allow safer testing of updates before they reach end users. Currently, testing occurs in local development environments, which may not perfectly replicate production conditions. A staging environment hosted on similar infrastructure would catch deployment-specific issues before they affect the live system.

A proper change management process would also help keep track of updates as the system grows. This means documenting what changes are being made, checking how they might affect other parts of the system, and having a plan to undo changes if something goes wrong. Right now, the informal approach works fine since it's just one person handling everything, but having a more organized process would make things easier if the project eventually needs more developers or gets handed off to someone else down the line.