Cover Page

Project Name: Student Lunch Payment Application

Developer: Amstrong Akendung

Submission Date: 05/02/2025

Table of Contents

1. Introduction

1.2 Purpose:

This document outlines the system requirements for the Online Student Lunch Payment System. This system aims to replace traditional cash and check-based lunch payment methods with a more efficient, secure, and convenient online solution for parents, students, and school administrators.

1.3 Scope:

The Online Student Lunch Payment System will provide the following functionalities:

- Secure online payments for student meals
- Account management for parents (add students, view balances, transaction history)
- Automated payment options
- Low-balance alerts
- Student balance checks
- Meal pre-ordering (optional)
- Reporting and administrative functions

2. Customer Problem Statements and System Requirements

2.2 Customer Problem Statement:

Schools, parents, and students face inefficiencies in managing lunch payments due to the lack of a centralized, digital payment system. Current lunch payment systems often involve cash or checks, leading to:

- **Inefficiency:** Time-consuming for school staff.
- **Security Risks:** Potential for errors and theft.
- **Inconvenience:** Difficult for parents to manage payments.
- **Lack of Visibility:** Limited access to real-time balance information.

2.3 System Requirements:

The Student Lunch Payment Application will address these problems by providing:

- Secure online payment options for parents.
- Easy account creation and management for parents and students.
- Allow parents to make lunch payments to their students' accounts.
- Provide real-time balance updates for student accounts.
- Send low balance alerts to parents.
- Track transaction history for transparency and accountability.
- Support multi-student management for families with more than one child.

3. Functional Requirements Specification

3.2 Stakeholders:

- **Parents:** Primary users of the system, interested in convenient payment options, account management, and balance tracking.
- **Students:** Users who need to view their balances and potentially pre-order meals.
- **School Administrators:** Managed accounts, generated reports, and configured system settings.
- **Cafeteria Staff:** Users who process payments, view student balances, and potentially manage meal pre-orders.
- **School District IT Staff:** Responsible for system installation, maintenance, and integration with existing school systems.
- **School District Finance Department:** Oversees financial aspects, including payment processing and reconciliation.

3.3 Actors and Goals:

Actor	Type	Goal
Parent	Human	Manage accounts, make payments, track balances, set up alerts.
Student	Human	View balance, pre-order meals (optional).
Administrator	Human	Manage accounts, generate reports, configure settings.
Cafeteria Staff	Human	Process payments, view balances, manage pre-orders (optional).
System	Software	Process payments, manage data, generate reports, send notifications.

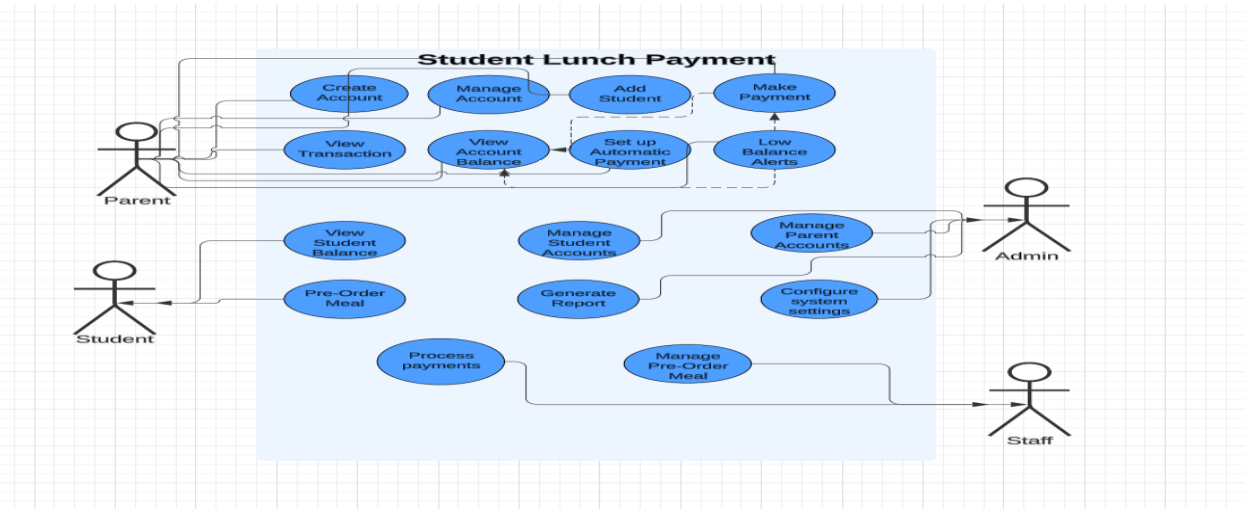
3.4 Use Cases:

Here's a breakdown of the use cases based on your functional requirements, along with estimated effort:

Use Case ID	Use Case Name	Description	Estimated Effort (Points)
UC-1	Create Account	Parent creates a new account with personal information and login credentials.	2
UC-2	Manage Account	Parent updates account information (e.g., contact details, password).	1
UC-3	Add Student	Parent adds a student to their account, linking them to their profile.	1
UC-4	Make Payment	Parent makes a payment to a student's account using various payment methods (credit/debit card, electronic check, etc.).	3
UC-5	View Transaction History	Parent views a history of transactions (payments and purchases) for a specific student.	2
UC-6	View Account Balance	Parent views the current lunch balance for a specific student.	1
UC-7	Set Up Automatic Payments	Parent configures automatic recurring payments to a student's account.	3
UC-8	Receive Low Balance Alerts	System sends an alert (email or SMS) to the parent when a student's balance falls below a defined threshold.	2
UC-9	View Student Balance (Student)	Student views their current lunch balance.	1
UC-10	Pre-order Meal (Student)	Student (or parent) pre-orders a meal for a future date (optional).	2
UC-11	Manage Student	Administrator creates, modifies, or deletes student	2

	Accounts (Admin)	accounts.	
UC-12	Manage Parent Accounts (Admin)	Administrator creates, modifies, or deletes parent accounts, potentially linking/unlinking them to student accounts.	2
UC-13	Generate Reports (Admin)	Administrator generates various reports (payment summaries, low balance reports, meal counts, etc.) for specific periods.	4
UC-14	Configure System Settings (Admin)	Administrator configures system-wide settings (e.g., low balance thresholds, payment options, meal pricing).	3
UC-15	Process Payment (Cafeteria Staff)	Cafeteria staff processes a student's payment (online or offline).	2
UC-16	View Student Balance (Cafeteria)	Cafeteria staff views a student's current balance at the point of sale.	1
UC-17	Manage Meal Pre-orders (Cafeteria)	Cafeteria staff views and manages meal pre-orders for students (optional).	2

3.5 Use Case Diagram:



3.6 Use Case Description

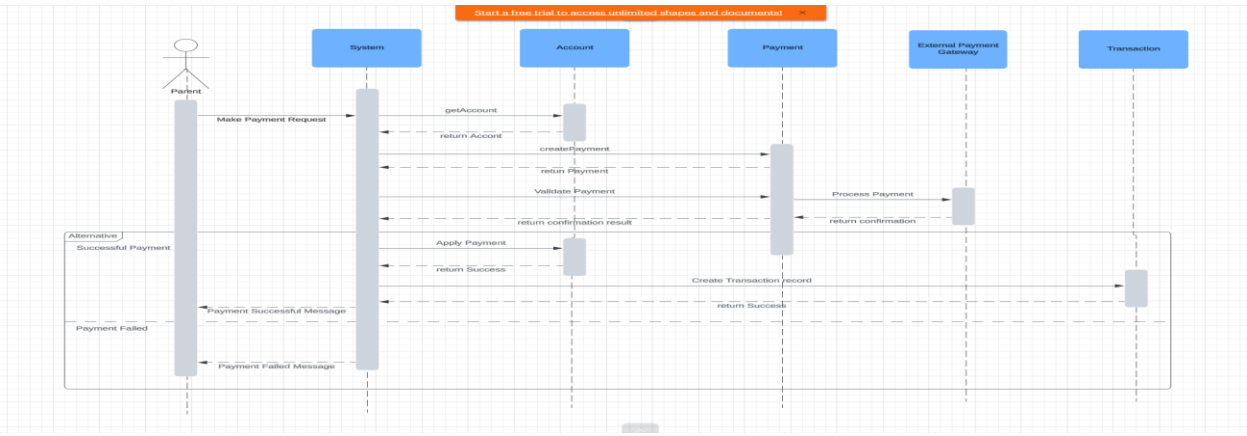
- **Make a Payment to a Student's Account**
    - Description: Allows parents to top up their child's lunch account.
    - Trigger: Parent initiates payment request on the app.
- **View Account Balance**
    - Description: Displays real-time balance information for each student account.
    - Trigger: User logs into their account.
- **Set Low Balance Alert**
    - Description: Notifies parents when the account balance falls below a defined threshold.
    - Trigger: Parent sets a threshold level.
- **View Transaction History**
    - *Description*: Provides access to all past transactions, including payment dates, amounts, and statuses.
    - *Trigger*: User navigates to the transaction history section.
- **Manage Multiple Student Accounts**
    - *Description*: Allows parents to manage lunch balances and payments for multiple students from a single account.

4. System Sequence Diagram

4.2 Make Payment:

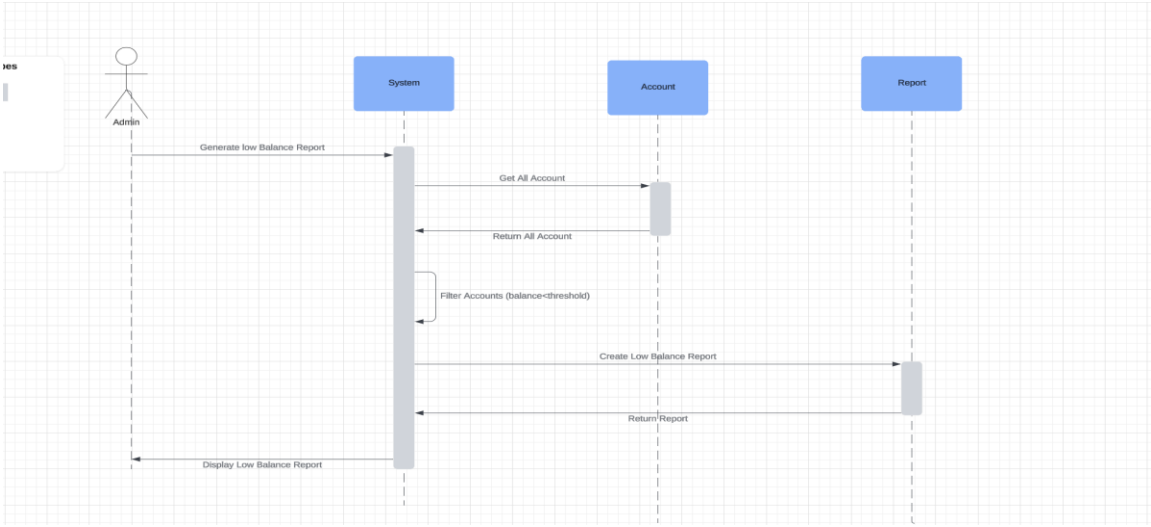
The below Steps explain what happens when a Parent makes a payment to a student's lunch account:

- The Parent actor sends a "Make Payment Request" message to the System, providing the studentID, amount, and paymentMethod.
- The System then sends a message to the Account object to retrieve the account associated with the studentID.
- The System creates a Payment object with the provided amount and paymentMethod.
- The System sends a validatePayment() message to the Payment object to ensure the payment details are valid.
- Alternative flows based on validation:
    - **Valid Payment:**
        - The Payment object interacts with an External Payment Gateway to process the payment.
        - If successful, the Payment object is applied to the Account, a Transaction record is created, and a success message is returned to the Parent.
    - **Invalid Payment:**
        - If the payment validation fails, a failure message with the reason is returned to the Parent.



4.3 Generate Low Balance Report

Below, the Use case diagram visually represents the interactions that happen when an Administrator generates a report that lists all Student accounts with low balances.



5. Activity Diagrams:

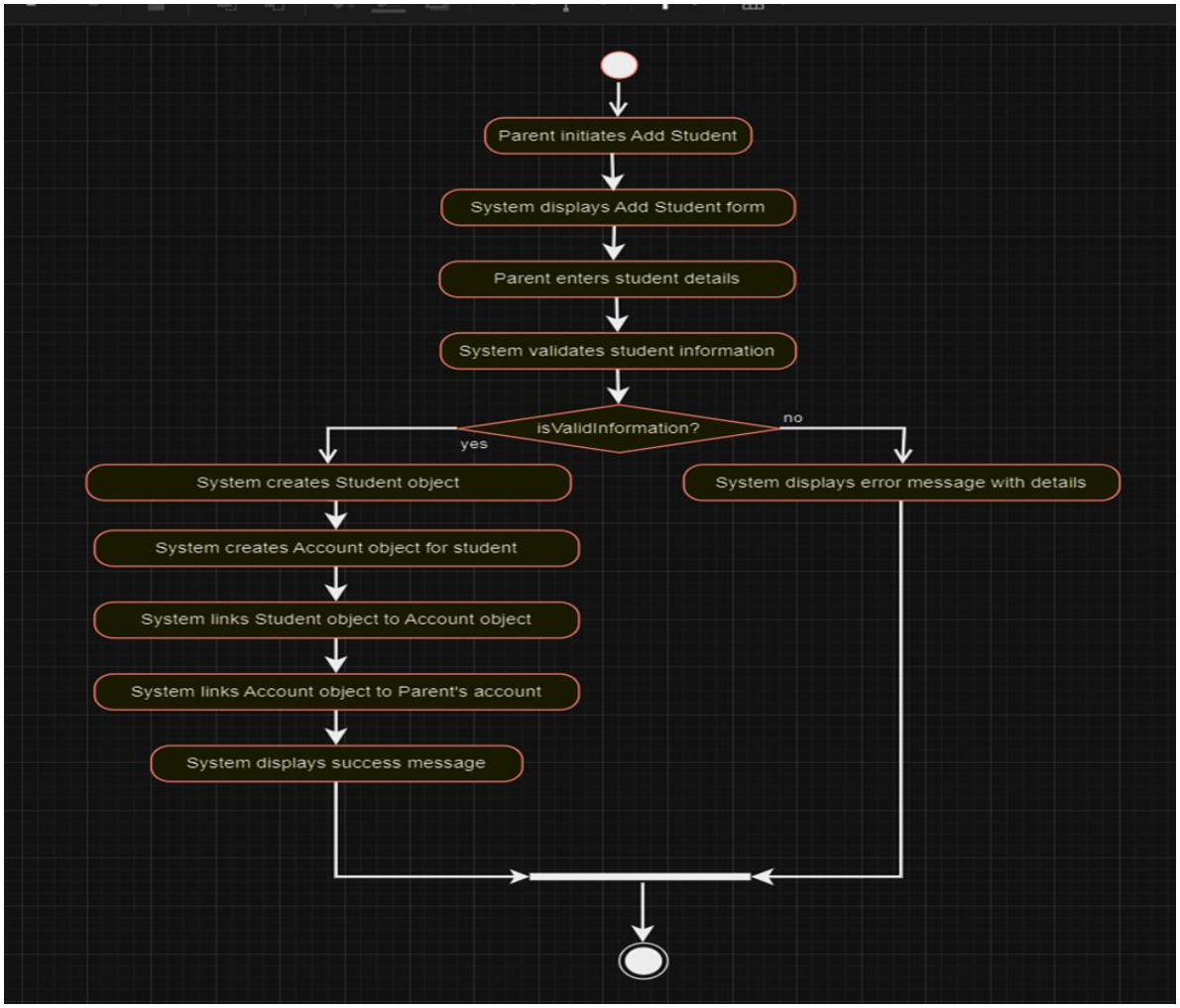
5.2 Adding Student:

This use case describes the process of a parent adding a new student to their account.

States:

- Initial State: The parent is logged in and can access their account
- Final State: System Add a student to the Parent Account

Actions: Parent login and can see his or her account information. Parents click on Add Student, and the system displays the Add Student form. Parents enter student information. Parents submit the form, and the system validates student information and creates Student and Account objects. Display success or error message.

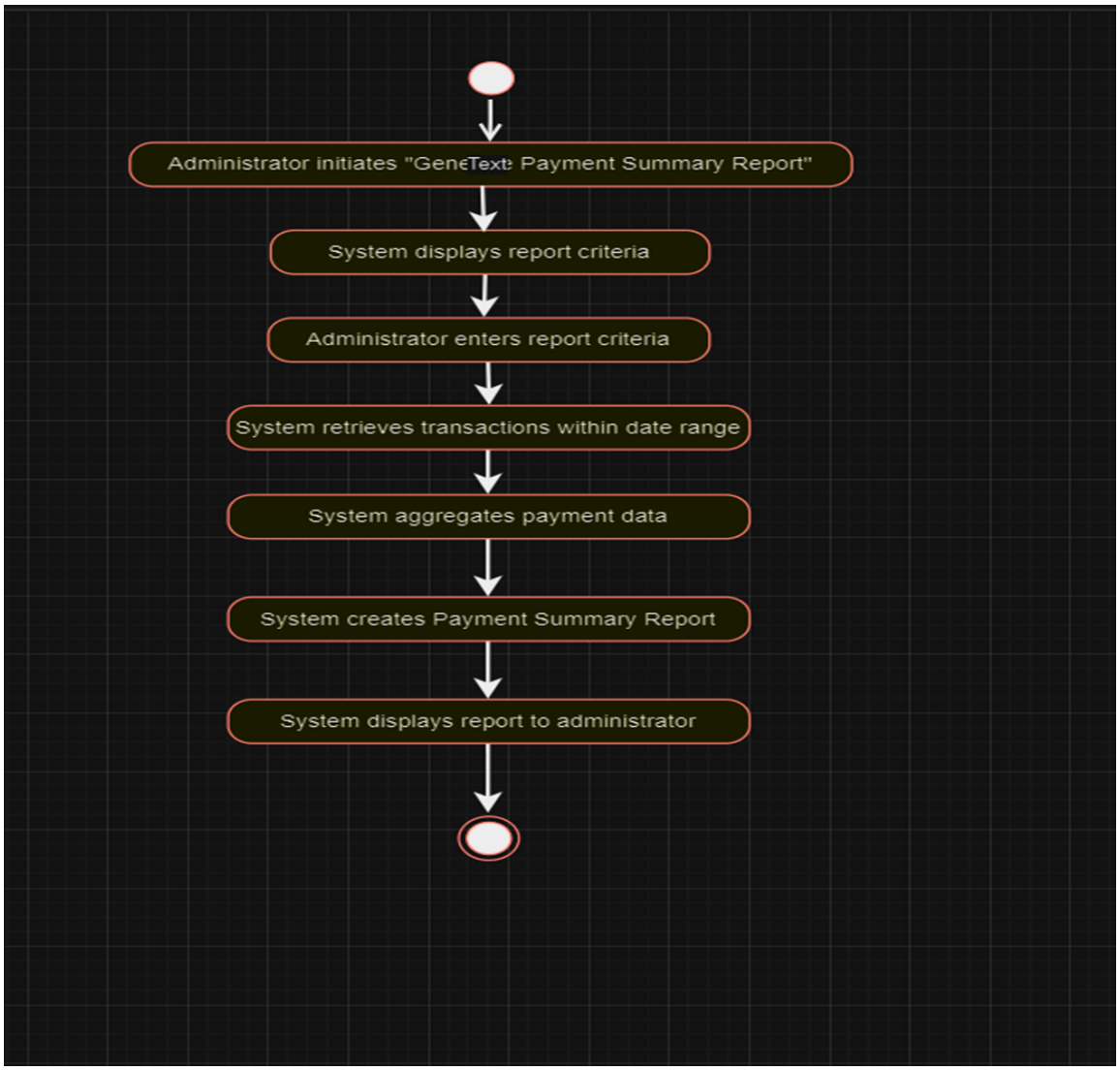


5.3 Generate Payment Summary Report:

States:

- Initial State: Administrator logged in and ready to generate a report.
- Final State: System Shows report to the administrator.

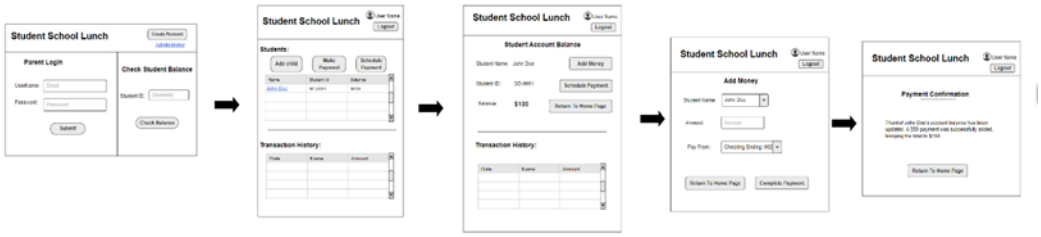
Actions: The administrator logs in and can see his or her account information. Enters report criteria, System generates the report, and Administrator views the generated report.



6. User Interface Specification

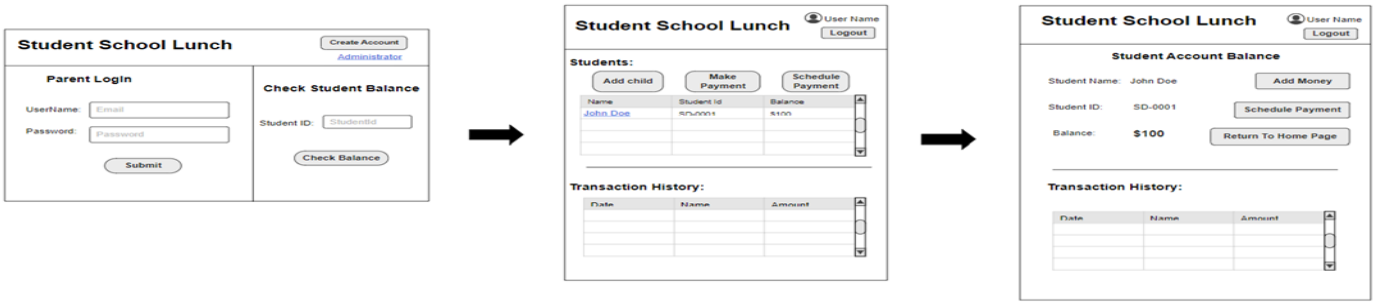
6.2 Make a Payment to a Student Account

- **Login Page**: Parent logs in securely.
- **Dashboard**: Shows all students, transaction history, and payment options.
- **Student Account**: Shows a Student Account
- **Payment Screen**: Input fields for amount and payment method, confirmation pop-up.



6.3 Views transactions History of a student.

- **Login Page**: Parent logs in securely.
- **Dashboard**: shows all students, transaction history, and payment options.
- **Student Account**: shows a student account with a balance.



7. Project Plan

7.2 Software:

- **Node.js:** This is used to run JavaScript on the server side. It's the backbone for running an Express server.
- **Express:** A framework for Node.js that helps build RESTful APIs, handling routing and middleware for server-side operations.
- **React:** A JavaScript library for building the user interface, allowing for a dynamic and component-based front end.
- **MySQL:** A relational database management system (RDBMS) used to store and manage data such as student and parent accounts.
- **JWT (JSON web token):** A library for generating and validating JSON Web Tokens, essential for implementing authentication and authorization.
- **Cookie-parser:** A middleware for Express to handle cookies, enabling HttpOnly and Secure cookies to store tokens.
- **Bcrypt:** Used to hash and validate passwords securely.
- **Npm:** Package managers for installing dependencies and managing libraries in Node.js project.

7.3 Hardware:

- Web server: Hosting environment for Node.js backend and API
- Database server: XAMP provides an easy-to-set-up environment, allowing me to manage MySQL locally.

7.4 Network:

- Secure internet connection
- Firewall protection.

7.5 Framework

- **Backend:** Node.js with the Express framework for handling server-side routing and API endpoints.

- **Database:** MySQL for reliable data storage
- **Front-end:** React for building the user interface.

7.6 Timeline:

- **Phase 1: Requirements Gathering and Design (2 weeks)**
    - Gather detailed requirements from stakeholders (parents, school staff, administrators).
    - Design the system architecture, database schema, and user interface.
- **Phase 2: Development and Testing (6-8 weeks)**
    - Develop the core functionality (user accounts, payments, reporting).
    - Implement integration with existing school systems.
    - Conduct thorough testing to ensure functionality and security.
- **Phase 3: Deployment and Training (1-2 weeks)**
    - Deploy the system to a production environment.
    - Provide training to school staff and parents on how to use the new system.
- **Phase 4: Ongoing Maintenance and Support**
    - Monitor system performance and address any issues.
    - Provide ongoing support to users.
    - Implement enhancements based on feedback and evolving needs.

7.7 Deliverables: Detailed mock-ups, functional requirements document, system prototype, user feedback, final product, and user training materials.

8. Traceability Matrix

Req't	PW	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11	UC12	UC13	UC14	UC15	UC16	UC17
REQ1	5	X	X	X														
REQ2	5	X	X	X														
REQ3	5	X	X		X		X											
REQ4	3	X	X			X												
REQ5	2							X										
REQ6	4	X	X						X									
REQ7	3									X								
REQ8	2										X							
REQ9	4											X	X					
REQ10	5													X				
REQ11	3														X			
REQ12	5															X		
REQ13	5																X	
REQ14	1																	X
Max PW		5	5	5	5	3	5	2	4	4	3	2	2	5	3	5	5	1
Total PW		22	22	10	5	3	5	2	4	4	3	2	2	5	3	5	5	1

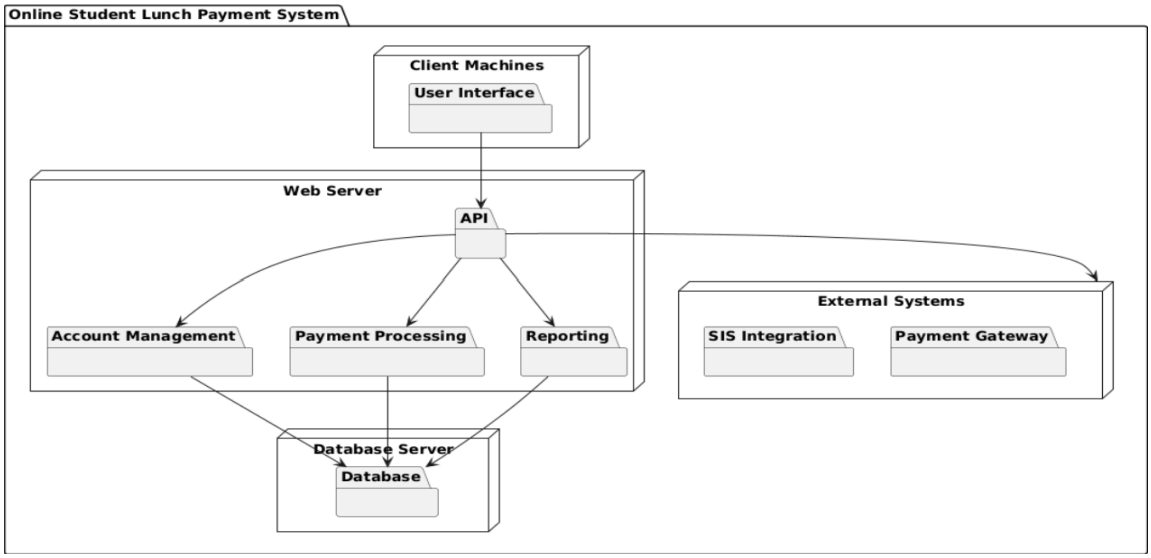
9. System Architecture and System Design:

9.2 Architectural Styles: The Online Student Lunch Payment System incorporates several architectural styles in its design to achieve its goals of security, scalability, and maintainability. The system follows a classic client-server model where the web browser (or potentially a mobile app) acts as the client, providing the user interface and handling user interactions. This is the back end of the system, which handles the processing of requests, data storage, and business logic. In this case, the server would run Node.js with Express.js to handle API requests and interact with the MySQL database. XAMP provides an easy-to-set-up environment, allowing me to manage MySQL locally.

9.3 Identifying Subsystems: The UML package diagram provides a high-level view of the system's architecture, showing the key subsystems and their relationships. It helps in understanding the overall structure and organization of the Online Student Lunch Payment System. The client machines represent the computers used by parents, students, and Admins to access the system within their machine web browser. The Web Server hosts the core application logic, and the API, Account Management, Payment Processing, and Reporting subsystems reside on this server. The Database subsystem, typically a MySQL database, runs on this machine. External systems represent external services or systems that the application interacts with, like the Student Information System and Payment Gateway. (not implemented in this project). The arrows in the package diagram indicate dependencies between subsystems. For example, the User Interface subsystem depends on the API

subsystem to access back-end functionalities. The API subsystem, in turn, depends on the Account Management, Payment Processing, Reporting, and External Systems subsystems to fulfill user requests.

9.4 UML Package Diagram:



9.5 Persistent Data Storage: The Online Student Lunch Payment System definitely needs to store data that outlives a single execution. A relational database like MYSQL is the most suitable storage management strategy for this system. Here are the key persistent objects in the system User, Account, Transaction, Meal, and Order.

9.6 Network Protocol: Since the Online Student Lunch Payment System will be a web-based application accessed by users on different machines, HTTP is the primary communication protocol that will be used in this system. Web browsers (clients) will use HTTP to send requests to the web server (Node.js with Express.js) and receive responses. The front end will make API calls to the back end using HTTP methods like GET, POST, and PUT. Data will be exchanged between the client and server in formats like JSON (JavaScript Object Notation)

Global Control Flow

9.7 Execution Order: The Online Student Lunch Payment System is primarily event driven. The system's back-end (Node.js with Express.js) is designed to wait for events, such as User requests, schedule tasks, or external systems. When an event occurs, the system reacts by executing the appropriate code to handle that event. Many operations in the system are

asynchronous, meaning they don't necessarily happen in a predefined order. For example, a parent might make a payment while an administrator is generating a report.

9.8 Time Dependency: The Online Student Lunch Payment System is primarily event-driven, meaning it reacts to events like user requests, but it also incorporates periodic tasks using timers. A timer can be used to periodically check account balances and trigger alerts to parents when balances fall below a threshold. This could be done daily or at other defined intervals. The system can also have scheduled tasks to generate reports (e.g., daily payment summaries and weekly low-balance reports) automatically.

9.9 Hardware Requirements: The Online Student Lunch Payment System will need some hardware and software to operate effectively and meet the needs of the school:

- Device Type: Desktops, laptops, tablets, and smartphones.
- Operating System: Any operating system with a modern web browser.
- Web Browser: A recent version of a standard web browser with JavaScript enabled.
- Internet Connection: A stable internet connection with sufficient bandwidth to access web pages and transmit data.
- Processor: A multi-core processor with a clock speed of at least 2 GHz.
- Memory (RAM): Minimum 4 GB RAM, 8 GB or more recommended for optimal performance, especially with high user traffic.
- Storage: Minimum 100 GB of hard disk space for the operating system, application files, and database. SSD storage is recommended for better performance.

10. Implementation Planning:

10.2 Implementation Strategy: A pilot launch enables a limited number of users, such as a school or department, to utilize the system in a real environment to identify any last-minute bugs or usability issues. After feedback and adjustments, a full transition from the old to the new system ensures complete deployment across the organization.

10.3 Training Strategy:

- **Target Group Training:** Parents, school staff (admin), IT support team
- **Training Tools:**
    - Step-by-step user manual (PDF)
    - Short video tutorials on login, payment, and account management

- Onsite or virtual demo session.

- **Duration:** 2–3 days per group

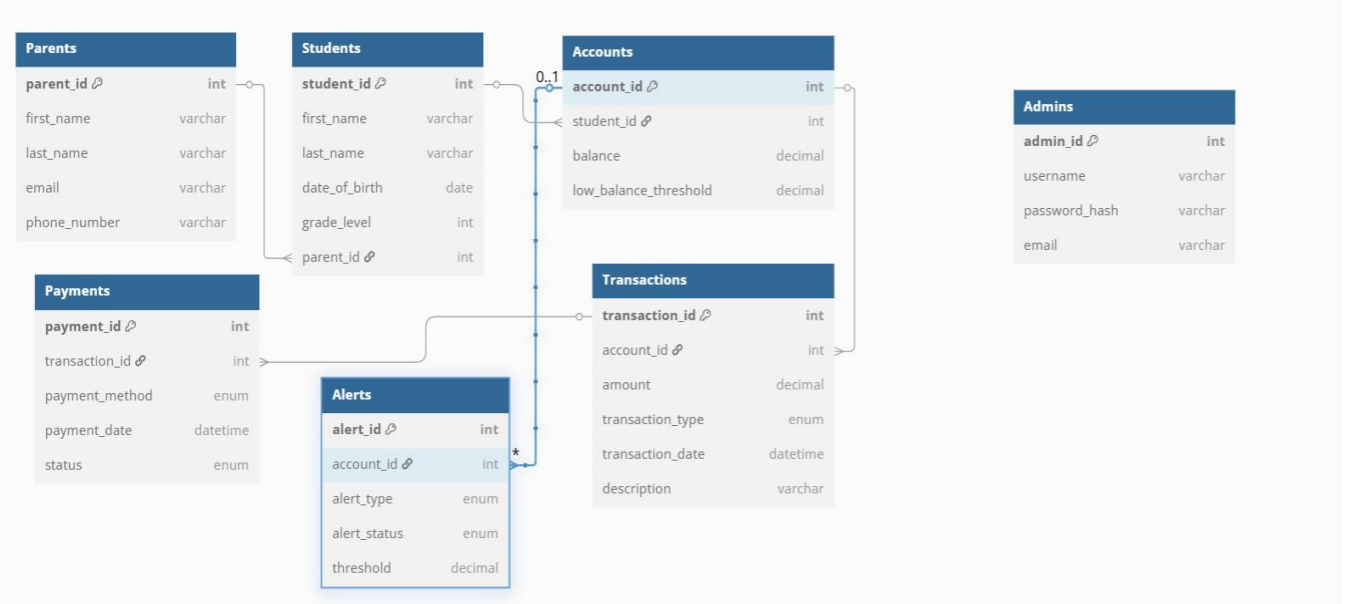
10.4 Support Strategy:

- **Ongoing Support:**
  - **A helpdesk ticketing system or shared support email**
  - **Scheduled monthly maintenance check-ins**
  - **Quick reference FAQ on the system homepage**
- **Implementation Timeline**

Task ID	Task Name	WBS	Duration	Start Date	End Date	Predecessors	Resources	Milestones
1.1	Gather Stakeholder Requirements	1	2 weeks	1/10/2025	1/24/2025	None	Project Manager, Business Analyst	M1: Requirements Gathering Complete (2024-12-27)
1.2	Analyze Existing System	1	1 week	1/10/2025	1/14/2025	None	Business Analyst, System Analyst	
1.3	Define Functional Requirements	1	1 week	1/17/2025	1/21/2025	1.1, 1.2	Project Manager, Business Analyst, Development Team	
1.4	Define Non-Functional Requirements	1	1 week	1/17/2025	1/21/2025	1.1, 1.2	Project Manager, Business Analyst, Development Team	
1.5	Document Use Cases	1	1 week	1/24/2025	1/28/2025	1.3	Business Analyst	
1.6	Create System Requirements Document	1	1 week	1/31/2025	2/4/2025	1.4, 1.5	Business Analyst, Project Manager	
2.1	Design System Architecture	2	3 weeks	2/7/2025	2/21/2025	1.6	System Architect, Development Team Lead	M2: System Design Complete (2025-01-31)
2.2	Design User Interface	2	2 weeks	2/7/2025	2/18/2025	1.6	UI/UX Designer	
2.3	Design Database Schema	2	2 weeks	2/10/2025	2/21/2025	2.1	Database Administrator	
2.4	Develop Security and Privacy Plan	2	1 week	2/24/2025	2/28/2025	2.1	Security Engineer	
2.5	Create Design Documents	2	1 week	3/3/2025	3/7/2025	2.2, 2.3, 2.4	System Architect, Development Team Lead	
3.1	Develop Front-End Application	3	6 weeks	3/10/2025	4/14/2025	2.5	Front-End Developers	
3.2	Develop Back-End Application	3	6 weeks	3/10/2025	4/14/2025	2.5	Back-End Developers	
3.3	Develop Payment Gateway Integration	3	2 weeks	3/24/2025	4/4/2025	3.2	Back-End Developers, Integration Specialist	
3.4	Develop Reporting Module	3	3 weeks	3/27/2025	4/16/2025	3.2	Back-End Developers	
3.5	Develop Integration with SIS	3	4 weeks	4/7/2025	5/8/2025	3.2	Back-End Developers, Integration Specialist	M3: Development Complete (2025-04-25)
4.1	Unit Testing	4	3 weeks	4/14/2025	5/2/2025	3.1, 3.2, 3.3, 3.4	QA Testers	
4.2	Integration Testing	4	2 weeks	5/5/2025	5/16/2025	4.1	QA Testers	
4.3	System Testing	4	2 weeks	5/19/2025	5/30/2025	4.2	QA Testers	
4.4	User Acceptance Testing (UAT)	4	2 weeks	6/2/2025	6/13/2025	4.3	School Staff, Parents (selected)	M4: Testing and UAT Complete (2025-06-06)
5.1	Set up Production Environment	5	1 week	6/16/2025	6/20/2025	4.4	System Administrator	
5.2	Migrate Data from Existing System	5	1 week	6/23/2025	6/27/2025	5.1	Database Administrator	
5.3	Deploy Application to Production	5	1 week	6/30/2025	7/4/2025	5.2	Development Team, System Administrator	M5: System Deployed (2025-06-27)
5.4	Configure System Settings	5	1 week	7/7/2025	7/11/2025	5.3	System Administrator	
6.1	Develop Training Materials	6	2 weeks	7/14/2025	7/25/2025	5.4	Training Specialist, Project Manager	
6.2	Conduct Training Sessions	6	2 weeks	7/28/2025	8/8/2025	6.1	Training Specialist	M6: Training Complete (2025-08-01)
6.3	Provide Ongoing Support	6	Ongoing	8/11/2025		6.2	Support Team	

11. Data Model Implementation:

11.2 ERD:



11.3 Normalization Strategy:

- **1NF (First Normal Form):** Each table contains atomic values and a unique primary key.
- **2NF (Second Normal Form):** All non-key attributes fully depend on the primary key.
- **3NF (Third Normal Form):** No transitive dependencies exist. For example, student and parent data are separated.

11.4 Indexing Strategy:

- Primary keys are automatically indexed.
- Foreign keys (like student_id, parent_id, account_id) will be indexed to improve JOIN operations.
- Columns frequently queried, such as account_id (in transactions), and alert_status (in alerts), should have secondary indexes for fast lookups.
- Date fields (like transaction_date, payment_date) can be indexed for time-range queries or reports.

12. Testing Strategy:

12.2 Unit Testing

- **Goal:** Verify that individual components and functions work as intended.
    - **Tools:** Jest (for React components), Mocha/Chai (for backend logic)
    - **Scope:**
        - Component rendering (e.g., AddStudentForm, PaymentForm)
        - Function outputs (e.g., calculateNewBalance, validateEmail)
        - API request/response mocking

12.3 Integration Testing:

- **Goal:** Test how components or modules interact with each other.
    - **Tools:** React Testing Library, Supertest (Node.js)
    - **Scope:**
        - Logging in and storing the JWT token
        - Adding a student and verifying account creation
        - Making a payment and updating the balance

12.4 System Testing:

- **Goal:** Validate the complete system's functionality against the business requirements.
    - **Tools:** Manual testing, Selenium (optional for automated UI testing)
    - **Scope:**
        - Full parent workflow from login to payment
        - Error handling (e.g., insufficient balance, invalid input)
        - Database consistency checks

12.5 Sample Test Cases:

- **Test Case 1:** Add Student Successfully

Test Step	Expected Result
Parent logs into the system	Dashboard loads with "Welcome" message
Clicks "Add Student"	The student form is displayed
Enter valid student info & submit	Student appears in the dashboard with a $0 balance

- Test Case 2: Make Payment to Student Account

Test Step	Expected Result
Navigate to Student Profile	Student info and balance are shown
Click “Add Funds” and enter $20	The success message appears
Balance updates from $0 to $20	The database reflects the new balance

13. Error Handling and Debugging

13.2 Error Logging

- **Backend (Node.js):**
  - Use Winston or Morgan for structured logging.
  - Log API errors, database connection issues, and unexpected behavior.
  - Store logs in separate files (e.g., error.log, access.log).
- **Frontend (React):**
  - Console warnings and error boundaries.
  - Optional integration with Sentry or LogRocket for error reporting.

13.3 Exception Handling

- Catch and respond to exceptions gracefully to avoid application crashes.
- Return meaningful HTTP status codes (e.g., 400 for bad requests, 500 for server errors).
- Use try-catch blocks for async operations.

13.4 Debugging Tools

- **Backend:**
  - Node.js Inspector (Chrome DevTools)
  - console.log() for quick debugging
- **Frontend:**
  - React Developer Tools
  - Chrome DevTools

14. Collaboration and Code Integration

14.2 Version Control System

Git (hosted on GitHub) to tracks changes, supports branching, enables team collaboration, rollback, and issue tracking.

14.3 Team Collaboration

- Slack / Microsoft Teams for daily check-ins
- Jira for task tracking

- GitHub repository with protected branches and pull request workflows

14.4 Code Reviews

- Pull requests (PRs) are required for all new features and bug fixes.
- At least one team member must approve the PR before merging.
- Use GitHub's PR comment system to suggest changes

14.5 Git Branching Workflow

- main
    - dev
        - feature/add-student
        - feature/add-parent
        - feature/payment-integration

15. Performance Optimization Plan

15.2 Potential Bottlenecks

- As the number of Transactions, Accounts, and Payments grows, querying balances or transaction histories may slow down.
- Foreign key fields (e.g., student_id, account_id) could result in full table scans without proper indexing.
- Repeated database hits for the same data (e.g., checking balance before every payment) can degrade performance under high usage.

15.3 Optimization Strategies

- **Indexing**
    - Foreign keys: student_id, account_id, parent_id, transaction_id
    - Frequently filtered/sorted fields: transaction_date, email
    - Why: Reduces query execution time, especially on joins and searches.
- **Caching**
    - Cache frequently accessed data like student balances, account details, and parent login sessions.
    - How: Use React state management or local storage to cache balance information temporarily.
    - Use in-memory caching tools like Redis for recent queries or authentication tokens.

16. Maintenance and Support Strategy

16.2 Types of Maintenance

- Corrective Maintenance: Fixing bugs and defects found after deployment.
- Adaptive Maintenance: Modifying the system to adapt to new environments (e.g., browser updates, operating system changes).
- Perfective Maintenance: Enhancing functionality or improving performance based on user feedback.

16.3 Change Management & Version Control

- **Version Control: Git + GitHub**

    - Use Git branching strategies (feature branches, main, dev) to manage updates.
    - All code changes must go through pull requests and peer reviews before merging.
- **Change Management Workflow:**
    - Submit change requests (via GitHub issues or project board).
    - Evaluate and assign changes based on urgency and impact.
    - Test in staging before merging to production.
    - Log changes in a CHANGELOG.md file.

16.4 Documentation and Support

- **Internal Documentation:**
    - Maintain a /docs folder with API endpoints, data models, and deployment steps.
    - Use comments and consistent code style for maintainability.
- **User Documentation:**
    - Provide a User Guide PDF or online help section within the app.
    - Include FAQ and troubleshooting tips.
- **Feedback Mechanisms:**
    - In-app feedback form or link to a Google Form.
    - Track user-reported issues using a GitHub Issue Tracker.

17. References