VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Programming Integration Project

Project Proposal

Group 2

Advisor(s):	PhD Lê Xuân Bách	
Student(s):	Nguyễn Anh Duy	2352181 (2A)
	Phạm Quang Anh Duy	2352192 (2B)
	Hoàng Đức Hiếu Anh	2352030 (2C)
	Nguyễn Thế Anh	2352052 (2D / Leader)
	Huỳnh Nhật Huy	2352381 (2E)
	Đoàn Thiên Quý	2353035 (2F)

HO CHI MINH CITY, SEPTEMBER 2025

1 Requirements

1.1 Motivation

Students often struggle when trying to buy or sell stuff like textbooks, electronics, or even basic campus supplies. Current options aren't great: Facebook groups are messy and hard to search, while big platforms like Shopee or Lazada feel too bloated for quick student-to-student deals. Our idea, the **Campus Shop Assistant**, is meant to fill that gap by giving students a simple marketplace made for campus life. On top of that, it comes with a chatbot that can handle natural queries like "Find me a calculus book under 200k VND," making the whole process faster and easier.

1.2 Scope

Core Features

- **User Management:** Students sign up with their university email, log in securely, and manage a basic profile.
- **Item Listing:** Users can post items with details like title, category, price, and description.
- **Smart Search:** Search items either through filters (category, price, keywords) or by asking the chatbot in plain language.
- **Communication:** Show seller contact info and support simple buyer-seller messaging.

Boundaries

For now, we're only aiming for an MVP web app focused on campus trading. Things like real payment integration, a mobile app, push notifications, or full-scale deployment are left for future versions.

1.3 Related Work / Analysis

- **Facebook Marketplace:** Lots of reach, but too cluttered and not student-focused.
- **Shopee / Lazada:** Feature-rich but overkill for small peer-to-peer campus trades.
- **University Facebook Groups:** Easy to use but messy, with no real search or organization.

What makes the **Campus Shop Assistant** stand out is that it keeps things lightweight and campus-focused, while adding modern features like a chatbot that understands how students actually talk and search for items.

1.4 Functional Requirements

- **Authentication:** Register with university email (example@hcmut.edu.vn), secure login, session management, and basic password recovery functionality.
- **Item Management:** Create, edit, delete listings with title, category, price, description; optionally with images, condition, location; plus listing status control (active/sold/expired).
- **Search:** Manual search with multiple filters (category, price, keywords, date) combined with sorting (price, newest, relevance); plus chatbot interface for natural language queries.
- **Communication:** Display seller contact info (phone/email) and support basic in-app messaging.
- **UI:** Responsive design for desktop/mobile, simple navigation, clear item display.

1.5 Non-functional Requirements

- **Performance:** Search queries must return results in under 2s, supports ~100 concurrent users.
- **Usability:** Clean, student-friendly interface with clear navigation and error handling.
- **Security:** Secure password hashing, input validation, data protection, proper access control.
- **Reliability:** Aim for 99% uptime in demo/testing, with error handling and feedback.
- **Maintainability:** Modular, documented code with basic unit tests and clear API docs.
- **Compatibility:** Cross-browser support and responsive design for multiple devices.

1.6 Process and Enhancements

- **Development Methodology:** Follow Agile development with 4-week sprints, including daily stand-up meetings, sprint planning sessions, and regular code reviews through GitHub.

- **Sprint 1 (Weeks 1-4):**
  - **Goal:** Establish core user management and basic item listing functionality.
  - **Deliverables:**
    - User registration/login with university email and database setup (Users, Items, Categories).
    - Basic item posting and browsing interface with responsive design.
  - **Success Criteria:** Users can successfully register, login, post items, and browse existing listings
- **Sprint 2 (Weeks 5-8):**
  - **Goal:** Implement search functionality and enhance user experience.
  - **Deliverables:**
    - Advanced search with filtering options (price range, category, ...) and basic chatbot interface.
    - User profile management and personal listing dashboard
    - Item status management (active, sold, expired) and editing capabilities
  - **Success Criteria:** Search functionality performs within 2-second requirement, chatbot can process basic queries
- **Sprint 3 (Weeks 9-12):**
  - **Goal:** Complete MVP with communication features and system polish.
  - **Deliverables:**
    - In-app messaging system and enhanced chatbot capabilities (better natural language processing)
    - System testing, bug fixes, and performance optimization
    - User documentation and deployment preparation
  - **Success Criteria:** Complete system functions reliably, meets all performance requirements, ready for user testing

- **Quality Assurance:** Weekly testing sessions, peer code reviews, and documentation updates with GitHub issue tracking for bug management.
- **Future Enhancements:** Database query optimization through indexing, caching mechanisms for improved performance, and expanded chatbot vocabulary for campus-specific terminology.

1.7 Software Architecture

1. **Frontend Layer:** Modern web application using HTML5, CSS3, and JavaScript (React.js or Vue.js framework) with responsive design principles for cross-device compatibility.
2. **Backend Layer:** RESTful API server using Node.js with Express framework or Python with Flask, handling user authentication, business logic, and database operations.
3. **Database Layer:** Relational database (PostgreSQL or MySQL) with optimized schema including core tables:
   - `users` (id + email + name + phone + password_hash + created_at)
   - `categories` (id + name + description + parent_id)
   - `items` (id + user_id + category_id + title + description + price + status + images + created_at)
   - `messages` (id + sender_id + receiver_id + item_id + content + timestamp) [Future implementation]
4. **Integration Layer:** Simple chatbot service using natural language processing library (such as NLTK for Python) to parse user queries and convert them into database search parameters.

1.8 Team Roles

- **Project Lead (2D):** Overall project coordination, sprint planning facilitation, stakeholder communication, integration testing, and final deliverable assembly.
- **Frontend Developers (2B, 2C):** User interface design and implementation, responsive web development, user experience optimization, and chatbot interface creation.
- **Backend Developers (2A, 2E):** Server-side API development, database design and implementation, user authentication system, business logic programming, and chatbot query processing.
- **Quality Assurance Engineer (2F):** Test case development, system testing execution, documentation writing, performance testing, and bug tracking management.

2 Appendix

2.1 MoMo Online Payment Integration

This integration enables students to pay orders securely via MoMo E-Wallet (Sandbox) with clear payment status, while keeping Cash On Delivery (COD) as a fallback option. The system supports desktop and mobile websites, offline POS and QR payments, and app-to-app transfers within the MoMo application.

Requirements: Java 8 or higher, Maven.

Environment Setup: MoMo provides two environments, development and production, configurable via `selectEnv(String target, String process)` in the Environment model for proper setup during processes.

Integration Approach: The provided library supports transactions via All-In-One (AIO) Payment Gateway and other payment options (App-In-App, POS, Dynamic QR Code). Developers can use the ready processors in the `Processors` folder or extend base models in the `Models` folder. It is recommended to explore and run example code in `PayGate` and `NonAIOPay` for understanding.

Core User Stories:

- US1: Buyer selects MoMo at checkout, completes payment, and receives clear success or failure feedback.
- US2: The system exclusively trusts MoMo webhooks with verified signatures and handles them idempotently to prevent duplicate charges.
- US3: If MoMo transactions fail or become unavailable, COD payment remains an alternative for the buyer.

Main Flow:

1. Frontend chooses MoMo payment and sends a POST to `/payments/init` with HMAC signing; server creates a Payment with status `PENDING` and returns `payUrl/qrCodeUrl`.
2. Buyer completes payment on MoMo platform and is redirected to a `returnUrl` showing a “processing” state.
3. MoMo sends a webhook to `/payments/callback/momo`, where the system verifies the signature, checks idempotency, and updates Payment and Order statuses (`PAID` or `FAILED`).
4. Order confirmation is displayed only after payment status changes to `PAID` from the webhook.

Data Model:

- `Payment`: orderId, method=`MoMo`, status=`PENDING|PAID|FAILED`, amount, requestId, transId, signature, idempotencyKey, rawCallback, timestamps.
- `Order`: mirrors `paymentStatus`; state changes only via PaymentService.

APIs:

- `POST /payments/init`: initiates payment; returns `{payUrl, qrCodeUrl, requestId, orderId}`; requires `Idempotency-Key` header.
- `POST /payments/callback/momo`: webhook endpoint; verifies HMAC, updates status, returns 200 OK.
- `GET /payments/orderId/status` (optional): fetches payment status (`PAID|FAILED|PENDING`).

Security and Integrity:

- Verify HMAC signatures from MoMo; never trust client-modified totals or returnUrl.
- Implement idempotent `init` and webhook handling keyed on `orderId`, `requestId`, and `transId`.
- Recompute totals server-side and block client tampering.
- Use environment variables for secrets: `MOMO_PARTNER_CODE`, `MOMO_ACCESS_KEY`, `MOMO_SECRET_KEY`, `MOMO_ENDPOINT`, `MOMO_RETURN_URL`, `MOMO_IPN_URL`.

Acceptance Criteria:

- AC1: Order is marked `PAID` only after receiving a valid webhook.
- AC2: Late or duplicate callbacks do not cause repeated state changes.
- AC3: Callbacks with invalid signatures or amount mismatches are rejected and logged.
- AC4: Initialization failures respond with user-friendly errors while leaving COD as an option.

Testing:

- Happy path: `init` → `payUrl` → webhook success.
- Error cases: init timeout, MoMo cancel, bad-signature webhook, replayed webhook, amount mismatch.
- End-to-end test covering browsing, cart, checkout, MoMo sandbox payment, and order confirmation.

Deployment and Operations:

- Feature flag: `payments.momo.enabled`.
- Logging and metrics captured include `orderId`, `requestId`, `transId`, `resultCode`, `traceId` and KPIs such as payment success rate.
- Optional light reconciliation reports to identify `PENDING`/`PAID` mismatches over 7–14 days.