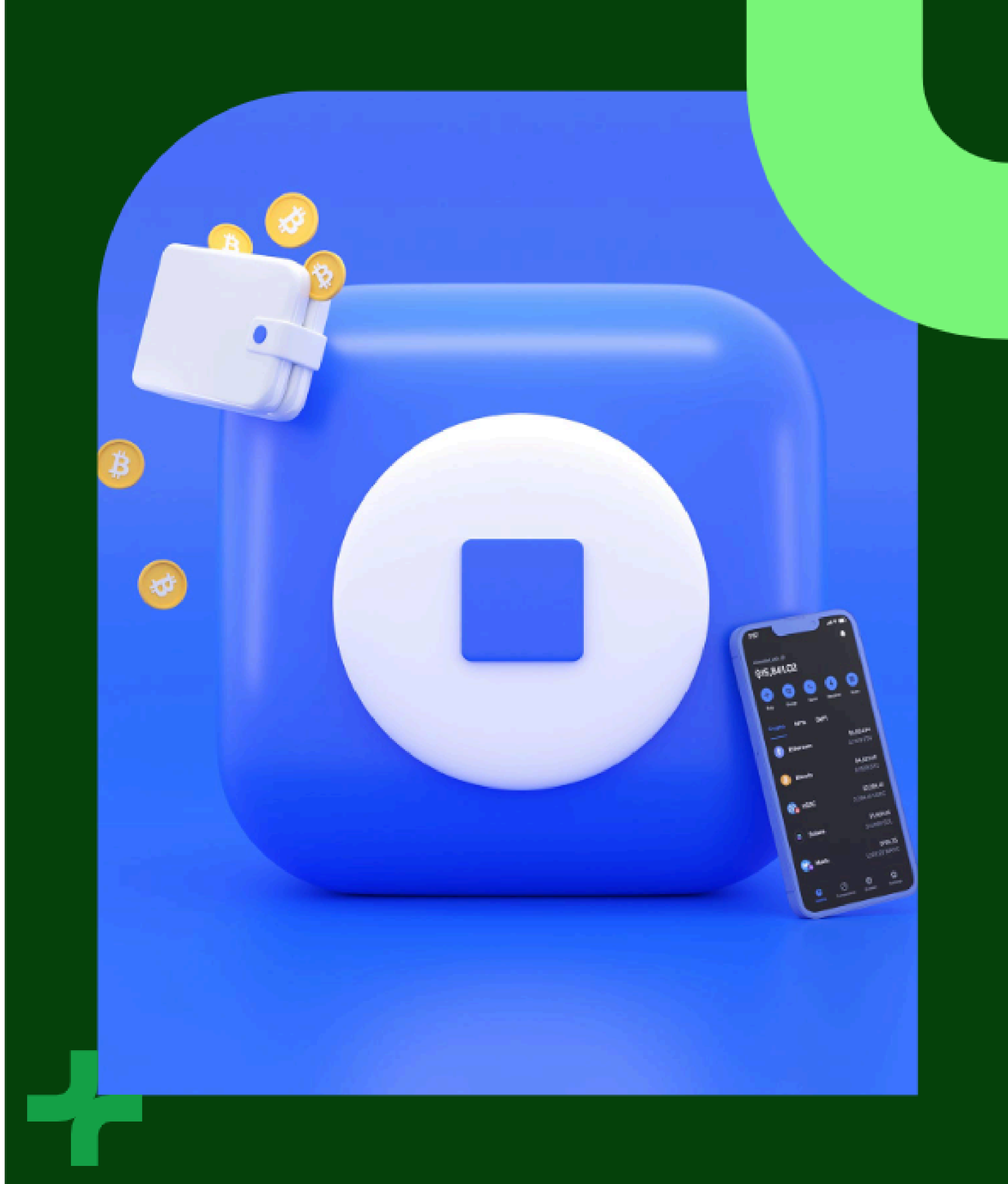
Designing a Simple Wallet for Secure Everyday Payments

Final Project - Software System Design (Without Coding)
Concise architecture, SDLC and Agile plan, data model, testing, CI/CD, and operations overview for a wallet app supporting top-up, transfer, history, and account management.

Raul Mahmud
Presenter



Project Vision and Scope

Simple, secure wallet app focusing on core payment flows

Vision: simple, reliable wallet app for top-up, transfer, and history

User-centered wallet emphasizing reliability and clarity

Scope: core flows for Customer and Admin roles with secure accounts

Defines what to build and who manages it

Main users: Customer and Admin

Primary personas controlling usage and operations

Main features: Top-up, Transfer, Transaction History, User Account Management

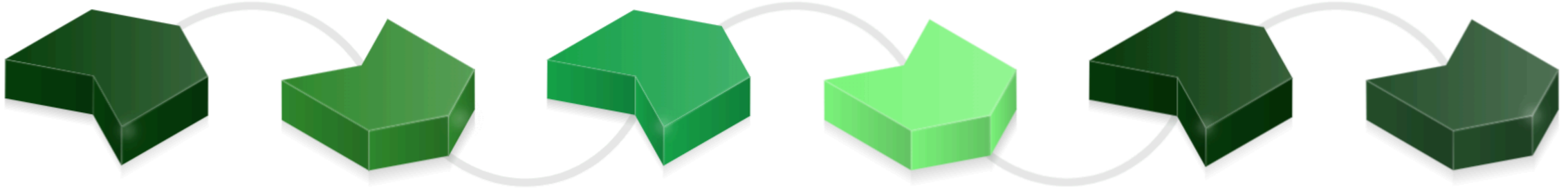
Essential payment and account capabilities

Constraints: define supported payment methods and regulatory requirements

Address security and compliance before implementation

SDLC Overview for the Wallet App

Linear process from requirements to maintenance



Requirements

Capture functional needs: top-up, transfer, history; non-functional: security, performance. Purpose: define what the system must deliver.

Design

Create architecture, ER diagrams, and API contracts. Purpose: turn requirements into a blueprint for implementation.

Implementation

Translate designs into code modules, interfaces, and follow coding standards. Purpose: build the system components.

Testing

Verify units, integrations, system behavior, and edge cases. Purpose: ensure quality and correctness.

Deployment

Package releases and deploy to environments with configuration management. Purpose: deliver working software to users.

Maintenance

Monitor, patch, and iterate based on user feedback. Purpose: sustain and improve the live system.

Agile and Scrum Plan

2-week sprints, rapid feedback, adaptive scope

Sprint cadence: 2-week sprints

Timeboxed iterations for frequent delivery and feedback

Product Owner: prioritizes backlog

Owns value and backlog ordering each sprint

Scrum Master: facilitates process

Removes impediments and coaches the team

Development Team: delivers increments

Cross functional team producing shippable work

Ceremonies: planning, daily stand-ups, review, retrospective

Regular cadence to plan, sync, inspect, and adapt

Backlog refinement each sprint

Allocate time to clarify and split user stories

Definition of Done: clear and consistent

Agree criteria so increments are shippable

Keep user stories small

Smaller stories reduce risk and increase predictability

Benefits: frequent feedback, early risk discovery, adaptive scope

Short cycles increase learning and reduce surprises

Product Backlog: User Stories

Six prioritized stories ready for refinement

User	Action	Goal
User	Top-up my wallet	Make payments
User	Transfer money to another user	Pay friends
User	View my transaction history	Track spending
User	Manage my account	Update personal details
Admin	View user activity	Monitor platform usage
Admin	Manage user accounts	Support customer issues

+ **Prioritize stories for next sprint**

  Assign business value and estimated effort

+ **Define acceptance criteria for each story**

  Specify conditions of satisfaction and test cases

Sprint Planning: Story Allocation

Two-sprint plan to deliver core customer flows then transfer and admin features

Sprint	Assigned User Stories	Capacity & Velocity	Objectives & Acceptance Focus
Sprint 1	Top-up; View Transaction History; Manage Account	Capacity: team sprint capacity estimate; Velocity: baseline estimate 20 story points	Objective: deliver user-facing flows and basic persistence. Acceptance: end-to-end customer flows, data persisted, basic error handling, unit and integration tests

1. **Estimate capacity before Sprint 1 and update after first velocity**
   Use Sprint 1 actuals to tune Sprint 2 forecast
2. **Ensure Sprint 1 covers persistence and core UX**
   Top-up, history, and account management fully integrated
3. **Allocate extra buffer for transfer complexity in Sprint 2**
   Include security, concurrency, and reconciliation tests
4. **Define acceptance criteria with clear test coverage**
   Unit, integration, and end-to-end tests for critical paths

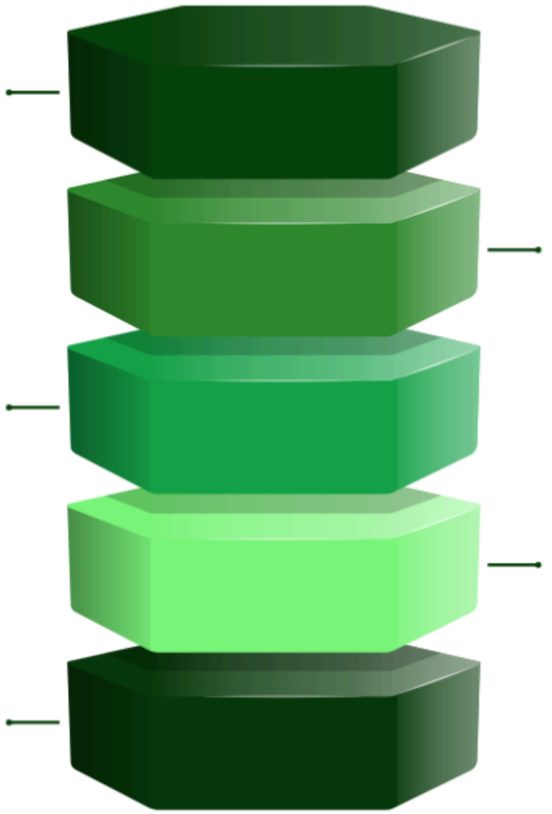
Use Case Diagram: User Actions

Customer and Admin interactions for wallet operations

Customer

Primary actor performing Top-up, Transfer, View History, Manage Account. Include happy and alternate flows.

1



Admin

2

Secondary actor with View Activity and Manage Accounts actions for oversight and administration.

Primary Flows

3

Happy path: successful top-up, successful transfer, view history retrieval, account updates.

Alternate Flows

4

Failed payment and insufficient balance branches to error handling and recovery steps.

Validation & APIs

5

Use diagram to validate requirements and identify required API endpoints for each use case.

High-Level Architecture Diagram

Client to Backend to Database with optional payment integration

Client

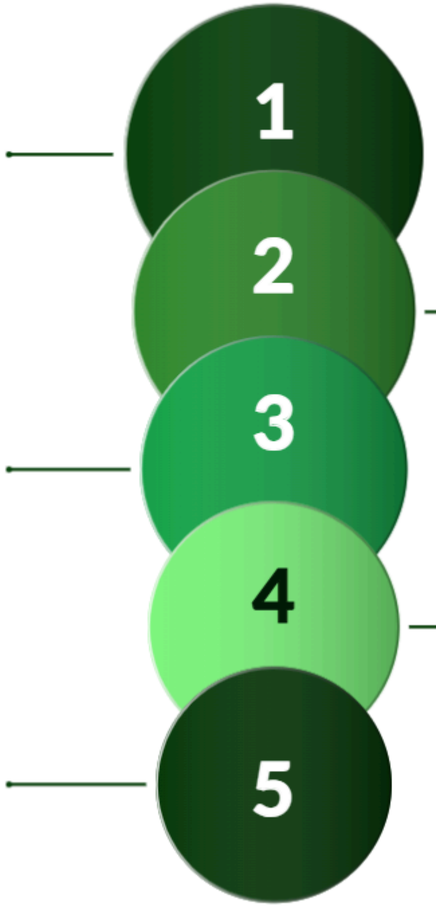
Mobile and web interfaces for users to top up, transfer, and view history.

Database

Persistent storage for accounts, transactions, and audit history.

Security & Auth

Authentication and authorization components to secure accounts and actions.



API Gateway / Backend Services

Stateless application services handling business logic, auth, and integrations.

External Payment API (optional)

Third-party payment provider used for top-up flows with secure credentials.

Database ER Design for a Wallet App

Core entities, keys, relationships, and performance notes

User

PK user_id, name, email, status, created_at, last_login; FK to Admin if role-based

Integrity & Performance

Timestamps, status fields, referential FKs; indexes on user_id and transaction timestamp

Admin

PK admin_id, user_id FK or separate credentials, role, audit_links, created_at

Wallet

PK wallet_id, user_id FK, balance, currency, status, created_at, updated_at

Transaction

PK transaction_id, wallet_id FK, amount, type, status, timestamp, balance_after

Testing Strategy for Wallet App

Layered testing with automation, data planning, and negative-case coverage

1 Unit testing: core business logic

- Verify top-up calculations and balance updates
- Validate transfer calculations and rounding
- Automate tests for fast CI feedback
- Cover edge cases like boundary amounts

2 Integration testing: backend and external services

- Test backend to database interactions
- Validate payment API integrations and error flows
- Automate integration suites in CI
- Use service stubs for unavailable endpoints

3 System testing: end-to-end flows

- Exercise full user journeys: top-up, transfer, history
- Verify data consistency across components
- Include performance and reliability checks
- Run in environment similar to production

4 API testing: contracts and error codes

- Validate contracts, response schemas, and status codes
- Test error codes for invalid requests
- Use contract tests to prevent regressions
- Automate as part of CI pipelines

5 UI testing: basic flows and error handling

- Smoke tests for top-up and transfer flows
- Verify UI error messages for invalid inputs
- Automate critical UI scenarios
- Keep UI tests focused to reduce flakiness

6 Test data and negative-case strategy

- Create realistic transaction datasets
- Include negative tests: insufficient balance
- Include invalid inputs and malformed requests
- Refresh and isolate test data per run

7

- R
- F
- S
- R

Sample Test Cases

Key positive and negative flows for wallet operations

Scenario	Input	Expected Result
Top-up Success	Top-up $10 for User A	Wallet balance increases by $10
Transfer Success	User A transfers $25 to User B with sufficient balance	Sender balance decreases by $25; recipient balance increases by $25; transaction recorded
Transfer Insufficient Funds	User A attempts to transfer $100 with $20 balance	Transfer rejected; balances unchanged; error message returned

1. **Test data setup: create users and seed balances**
   Create User A and User B with controlled balances; seed wallet for top-up and transfers.

2. **Post-conditions to verify after each test**
   Check balances, transaction log entries, and UI notifications match expected results.

Error Handling and Logging

Capture, centralize, monitor, and audit errors for reliable operations

1. **Types of errors to log: failed payments, invalid requests, system exceptions**

   Cover transactional, input validation, and runtime failures

2. **Recommended log fields: timestamp, service, user_id, correlation_id, error code, message**

   Use structured fields to enable search and tracing

3. **Storage and monitoring: centralize logs in aggregation system and set alerts for critical failures**

   Enable dashboards, anomaly detection, and alerting

4. **Audit trails for financial actions and retention policies**

   Record immutable audit records and enforce retention rules

5. **Use correlation IDs to trace requests across services**

   Link distributed logs for end-to-end debugging

6. **Graceful user-facing messages while excluding sensitive details from logs**

   Keep users informed and logs compliant with privacy

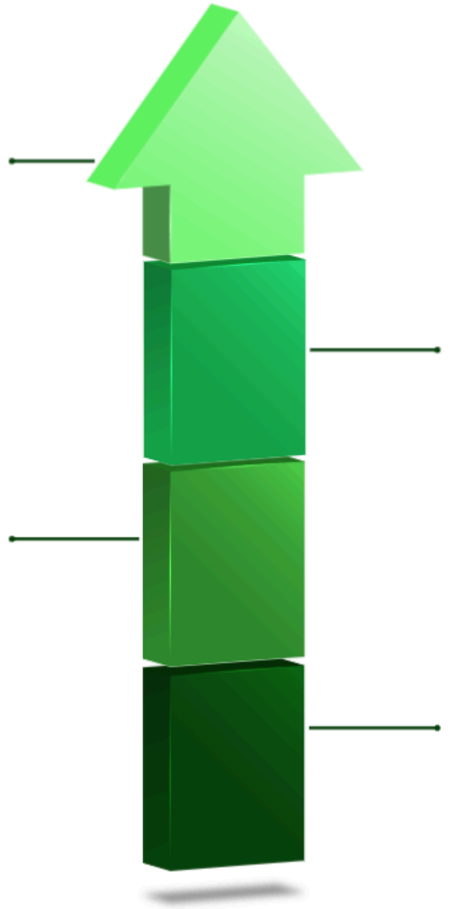
Git and CI/CD Concept

Feature branches, gated pipeline, and smoke tests

Deploy

Deploy to staging, run smoke tests, then manual approval to production

4



3

Build

Create artifact or container image after successful tests

Tests

Run automated unit and integration tests; require passing results

2

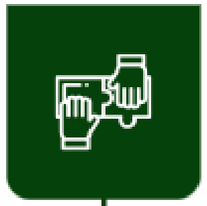
1

Push

Developer pushes feature branch or pull request to repository

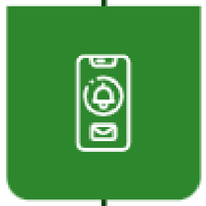
Thank You

Feedback, prioritization, and extenion topics



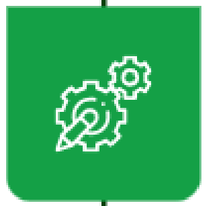
1 **Feedback on design artifacts**

Collect feedback on proposed design artifacts.



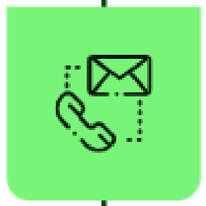
2 **Prioritization guidance for implementation**

Decide which features and fixes to prioritize.



3 **Potential extension topics: security hardening, compliance, scaling**

Discuss next areas to expand the solution.



4 **Contact: Raul Mahmud— preferred contact channel**

Include presenter name and preferred contact method.

5 **Invite questions and discussion on trade-offs in the proposed design**

Open floor for questions and trade-off discussions.