FalsoPay System Documentation

Software Engineering 1 Project

Project Team Members:

- [Team Member Name 1]
- [Team Member Name 2]
- [Team Member Name 3]
- [Team Member Name 4]
- [Team Member Name 5]

Submission Date: [Current Date]

Project Overview:
FalsoPay is a university-level fintech web application designed to allow instant, secure, and 24/7 money transfers between users, with access to linked bank accounts, debit cards, and prepaid cards. The app mimics core features of modern financial apps like InstaPay Egypt, tailored for educational purposes.

Documentation Package Contents:

1. Software Requirements Specification (SRS)
2. System Design Documents
3. UML Diagrams
    - Use Case Diagrams
    - Class Diagrams
    - Sequence Diagrams
    - Activity Diagrams
    - Object Diagrams
4. Database Design
5. System Architecture
6. Test Documentation
7. Development Guidelines

Course Information:

- Course: Software Engineering 1
- Professor: [Professor Name]
- Institution: [University Name]
- Semester: [Current Semester]

Table of Contents

1. Introduction

1.1 Purpose

FalsoPay is a university-level fintech web application designed to allow instant, secure, and 24/7 money transfers between users, with access to linked bank accounts, debit cards, and prepaid cards. The app mimics core features of modern financial apps like InstaPay Egypt, tailored for educational purposes.

1.2 Project Scope

The system will:

- Provide user authentication via email or Google.
- Support balance inquiries, transaction history, and fund transfers.
- Allow onboarding of debit/prepaid cards.
- Offer real-time money transfers between registered users.
- Be accessible via a responsive web interface.

1.3 Stakeholders

The project stakeholders include:

- End Users: Individuals who will use the application to transfer money
- System Administrators: Technical staff responsible for maintaining the system
- Course Instructors: Professors who will evaluate the project
- Development Team: Students implementing the system

1.4 Glossary

- **FalsoPay**: The name of our fintech application
- **Fintech**: Financial Technology
- **API**: Application Programming Interface
- **WebSocket**: Communication protocol providing full-duplex communication
- **SPA**: Single Page Application
- **JWT**: JSON Web Token used for authentication

2. Software Requirements Specification

2.1 Functional Requirements

User Requirements Specification

Total: 30 functions | Implementation: 20 functions

#	Function Description	Priority	Implemented
1	Register and log in via email	Must have	☑
2	Log in via Google	Should have	☑
3	Send money to a user by phone number	Must have	☑
4	View transaction history	Must have	☑
5	Check account balance	Must have	☑
6	Link a debit card securely	Must have	☑
7	Link a prepaid card	Must have	☑
8	Receive success/failure confirmation for transfers	Must have	☑
9	Edit profile and update phone number	Should have	☑
10	Reset password via email	Should have	☑
11	Block/unblock a linked card	Should have	☑
12	Report suspicious activity	Should have	☒
13	Scan QR code to pay another user	Should have	☑
14	View and download monthly account statements	Should have	☑
15	Add money to prepaid card from bank	Should have	☒
16	Set transfer limits per day/week	Could have	☑
17	Enable/disable notifications	Could have	☒
18	Access customer support chat	Could have	☒
19	Get in-app help/tutorials	Could have	☑
20	View app usage statistics (e.g. total amount sent)	Could have	☒
21	Add recipients to favorites for quick transfer	Should have	☑
22	Set a PIN for confirming transfers	Must have	☑
23	Schedule recurring transfers	Could have	☒
24	Export transactions as CSV/PDF	Could have	☒
25	Use biometric authentication (fingerprint/face)	Should have	☒
26	Multi-language support (English/Arabic toggle)	Won't have now	☒
27	Push notifications for incoming transfers	Should have	☑
28	Search/filter transactions by date or amount	Should have	☑
29	Request money from other users	Should have	☒
30	Create payment links to share externally	Could have	☑

System Requirements Specification

Total: 6 functions | Implementation: 4 functions

#	Function Description	Priority	Implemented
31	Process transfer requests in < 1 second	Must have	☑
32	Store passwords securely using hashing (e.g., bcrypt)	Must have	☑
33	Maintain a secure audit trail of all actions	Should have	☑
34	Auto-backup all data every 24 hours	Could have	☒
35	Support high concurrency without data loss	Must have	☑
36	Encrypt all sensitive data at rest and in transit	Must have	☑

Summary

- **36 total functions designed** (30 user + 6 system)
- **24 total functions selected for implementation** (marked ☑)
- Prioritized using **MoSCoW scheme**

2.2 Non-Functional Requirements

Categories Followed

- Performance
- Security
- Usability
- Maintainability
- Availability

Specification with Types

Requirement	Category
System must process 95% of transfers within 5 second	Performance
Passwords must be encrypted using bcrypt	Security
UI must be responsive and mobile-friendly	Usability
Code must follow PSR standards and be modular	Maintainability
System must have 99.9% uptime	Availability

Fit Criteria (Testable)

- **Performance**: Benchmark tests show 95% of API responses are < 5000ms.
- **Security**: Manual tests confirm no sensitive data is stored in plaintext.
- **Usability**: User feedback scores average 8+/10 in internal testing.
- **Maintainability**: 80%+ code coverage with unit tests.
- **Availability**: Monitored by uptime tools with 99.9% SLA simulation.

Impact on Architecture

- System will use a modular MVC architecture to ensure maintainability.
- APIs will be optimized and rate-limited to guarantee high performance.
- Security layer will include middleware for JWT validation and role checks.
- Frontend components will be decoupled to reduce load time and increase reusability.

2.3 Constraints

- **Development Timeline**: The project must be completed within the academic semester timeline.

- **Technology Constraints**: The system must be built using PHP for the backend and React for the frontend.
- **Database Constraints**: MySQL database must be used for data storage.
- **Platform Constraints**: The system must function correctly on major browsers (Chrome, Firefox, Safari, Edge).
- **Connectivity Constraints**: The system must handle intermittent connectivity gracefully.
- **Legal Constraints**: As an educational project, the system must operate with mock financial data only.

2.4 Validation Criteria

Approach to Requirements Validation

- **Traceability Matrix**: Each requirement is mapped to its corresponding test case
- **User Acceptance Testing**: Feedback from potential users evaluates if requirements meet expectations
- **Peer Reviews**: Team members review each other's work to ensure functionality matches requirements
- **Stakeholder Validation**: Regular meetings with instructors to validate requirements are being met

Success Criteria

- 95% of the "Must Have" requirements are successfully implemented and tested
- All implemented features pass their respective test cases
- The system demonstrates stability under normal usage conditions
- The application load time meets performance requirements
- Security measures are verified through penetration testing

3. System Design

3.1 Architectural Patterns

FalsoPay implements a combination of architectural patterns to create a robust, scalable, and maintainable payment system. Below are the key architectural patterns used and the rationale behind each choice:

Microservices Architecture

FalsoPay is built using a microservices architecture, which divides the application into loosely coupled, independently deployable services.

Why this pattern?

- **Scalability**: Each service can be scaled independently based on demand
- **Resilience**: Failure in one service doesn't bring down the entire system
- **Technology Flexibility**: Different services can use different technologies based on specific requirements
- **Independent Development**: Teams can work on different services simultaneously without tight coordination
- **Easier Maintenance**: Smaller, focused codebases are easier to understand and maintain

API Gateway Pattern

An API Gateway serves as the single entry point for all client requests, routing them to appropriate microservices.

Why this pattern?

- **Simplified Client Interface**: Clients only need to know about a single endpoint
- **Security**: Centralized authentication and authorization
- **Cross-cutting Concerns**: Handles common functionality like logging, rate limiting, and monitoring
- **Protocol Translation**: Can translate between web protocols and internal protocols
- **Request Aggregation**: Can combine results from multiple services for a single client request

Event-Driven Architecture

FalsoPay uses event-driven architecture for handling transactions and notifications.

Why this pattern?

- **Decoupling**: Services communicate through events without direct dependencies
- **Asynchronous Processing**: Enables handling of high-volume transactions without blocking
- **Real-time Updates**: Facilitates immediate notifications for account activities
- **Audit Trail**: Events provide a natural audit log for all system activities
- **Scalability**: Easily scales to handle varying loads of transactions

CQRS (Command Query Responsibility Segregation)

The system separates read operations (queries) from write operations (commands).

Why this pattern?

- **Performance Optimization**: Read and write operations can be optimized separately
- **Scalability**: Read-heavy operations can be scaled independently from write operations
- **Security**: Easier to implement different security models for reads vs. writes
- **Simplified Models**: Simpler domain models for specific operations

3.2 System Architecture

The FalsoPay system consists of the following major components:

1. **User Management Service**
    - Handles user registration, authentication, and profile management
    - Manages user roles and permissions
2. **Payment Processing Service**
    - Processes money transfers between accounts
    - Handles different payment methods
    - Implements transaction security measures
3. **Account Management Service**
    - Manages bank account linking
    - Handles balance inquiries
    - Processes transaction limits
4. **Notification Service**
    - Sends transaction confirmations
    - Delivers security alerts
    - Manages notification preferences
5. **Transaction History Service**
    - Stores and retrieves transaction records
    - Provides filtering and search capabilities
    - Generates transaction reports
6. **Security Service**
    - Implements PIN verification
    - Manages transaction blocking/unblocking
    - Detects suspicious activities
7. **Customer Support Service**
    - Handles support ticket creation and management
    - Provides system status information
    - Manages user reports
8. **API Gateway**
    - Routes client requests to appropriate services
    - Handles authentication and authorization
    - Implements rate limiting and monitoring
9. **Event Bus**

- Facilitates asynchronous communication between services
- Ensures reliable event delivery
- Supports event sourcing for audit trails

Integration Points

FalsoPay integrates with several external systems:

1. **Banking APIs**
   - For bank account verification
   - For processing transfers to/from bank accounts
2. **Payment Networks**
   - For processing card payments
   - For interoperability with other payment systems
3. **Regulatory Compliance Systems**
   - For KYC (Know Your Customer) verification
   - For AML (Anti-Money Laundering) checks

Deployment Architecture

FalsoPay uses a cloud-native deployment approach:

1. **Containerization**: All services are containerized using Docker
2. **Orchestration**: Kubernetes manages container deployment and scaling
3. **Infrastructure as Code**: Terraform scripts define and provision infrastructure
4. **CI/CD Pipeline**: Automated testing and deployment processes
5. **Multi-region Deployment**: For high availability and disaster recovery

Security Architecture

Security is implemented at multiple levels:

1. **Network Level**: Firewalls, VPNs, and network segmentation
2. **Application Level**: Input validation, output encoding, and secure coding practices
3. **Data Level**: Encryption at rest and in transit
4. **Identity Level**: Multi-factor authentication and role-based access control
5. **Monitoring Level**: Intrusion detection and security information and event management

3.3 Class Classification

FalsoPay's classes are categorized into the following classifications:

1. **Entity Classes**
   - User
   - Transaction
   - BankAccount
   - Card
   - SupportTicket
   - Notification
2. **Boundary Classes**
   - UserInterface
   - APIGateway
   - PaymentProcessor
   - NotificationSender
   - ReportGenerator
3. **Control Classes**
   - AuthenticationController
   - TransactionController
   - AccountController
   - NotificationController
   - SecurityController
4. **Service Classes**
   - UserService
   - TransactionService
   - AccountService
   - NotificationService
   - SecurityService
5. **Repository Classes**
   - UserRepository
   - TransactionRepository
   - AccountRepository
   - NotificationRepository
   - SecurityRepository

3.4 Interaction Patterns

The system employs several key interaction patterns:

1. **Request-Response**: Used for synchronous operations like balance inquiries
2. **Publish-Subscribe**: Used for notifications and event propagation
3. **Command-Query**: Implementing CQRS for optimized data operations
4. **Observer Pattern**: For real-time updates of transaction status
5. **Chain of Responsibility**: For processing transaction validations

3.5 Design Patterns

FalsoPay implements several design patterns:

1. **Factory Method**: For creating different types of payment methods
2. **Singleton**: For global services like authentication and logging
3. **Strategy**: For implementing different transaction processing algorithms
4. **Observer**: For notification handling
5. **Decorator**: For adding security layers to transactions

6. **Adapter**: For integrating with different banking APIs
7. **Facade**: For simplifying complex subsystems
8. **Command**: For encapsulating transaction operations

4. UML Diagrams

4.6 Object Diagrams

Object diagrams show instances of classes and their relationships at a specific point in time, illustrating the system's state before and after key operations. They differ from class diagrams in that they show specific instances rather than general structures. Each object is an instance of a class with specific attribute values.

FalsoPay uses pre-condition and post-condition object diagrams to visualize the system state changes during key operations:

4.6.1 User Registration

Pre-condition: Shows the system state before a new user registers with FalsoPay.

- The system contains only the base User type
- No user account exists for the email being registered
- The authentication system is prepared to validate the registration

Post-condition: Shows the system after successful user registration.

- A new User object exists with the provided information
- The system has generated a unique ID for the user
- Default account settings are established
- Authentication credentials are stored securely

4.6.2 Send Money

Pre-condition: Depicts the system state before a money transfer.

- Sender user has sufficient balance
- Both sender and recipient accounts are active
- Transaction limits have not been exceeded

Post-condition: Shows the system after a successful money transfer.

- Sender balance has decreased by the transfer amount
- Recipient balance has increased by the transfer amount
- A transaction record has been created
- Notifications have been triggered for both parties

4.6.3 Link Bank Account

Pre-condition: Illustrates the system state before linking a bank account.

- User exists in the system
- Bank account is not yet linked to the user
- Bank verification system is ready

Post-condition: Shows the state after successful bank account linking.

- Bank account is now associated with the user
- Bank verification status is updated
- User has access to the linked account functionality

4.6.4 Request Money

Pre-condition: Shows the system state before a money request.

- Requester and recipient users exist
- No pending request exists between these users for this amount

Post-condition: Depicts the system after a money request is created.

- A new MoneyRequest object exists
- Notification has been sent to the recipient
- Request appears in both users' activity logs

4.6.5 Support Ticket

Pre-condition: Shows the system before a support ticket is created.

- User exists in the system
- Support system is operational

Post-condition: Illustrates the system after a support ticket is submitted.

- A new SupportTicket object exists with the user's issue
- Ticket has been assigned a unique ID
- Initial status is set to "Open"
- Notification has been sent to support staff

4.6.6 Block App Transactions

Pre-condition: Shows the system state before a user blocks transactions.

- User account is active
- Transaction capability is currently enabled
- User is properly authenticated

Post-condition: Depicts the system after transactions are blocked.

- User's TransactionStatus is set to "Blocked"
- SecurityLog entry has been created documenting the change
- Notification has been sent to the user confirming the action
- Any pending outgoing transactions are canceled

These object diagrams serve as valuable tools for:

1. Validating that operations produce the expected results
2. Understanding the complex relationships between objects at runtime
3. Documenting the concrete effects of system operations
4. Guiding implementation of business logic

The diagrams are created using PlantUML and can be viewed using the PlantUML Web Server, IDE plugins, or the included HTML viewer.

6. Implementation Details

6.1 Backend Architecture

The FalsoPay backend is built using PHP with a custom MVC architecture. The backend handles all business logic, data persistence, and API endpoints.

Key Components:

- **Router**: Manages request routing to appropriate controllers
- **Controllers**: Handle request processing and response generation
- **Services**: Contain business logic and domain rules
- **Repositories**: Manage data access and persistence
- **Middleware**: Provide cross-cutting concerns like authentication

Server Optimization

The `server.php` file has been optimized to improve performance and handle parallel requests more efficiently:

1. **Resource Management**:
    - Implemented connection pooling for database connections
    - Added memory limits and garbage collection optimization
    - Configured timeout handling for long-running operations
2. **Session Handling**:
    - Implemented efficient session storage with Redis
    - Added session validation and cleanup mechanisms
    - Optimized session data structure
3. **Request Tracing**:
    - Added unique request IDs for tracing
    - Implemented structured logging for request lifecycle
    - Created performance monitoring points throughout the request flow
4. **Memory Optimization**:
    - Reduced unnecessary object instantiation
    - Implemented response streaming for large payloads
    - Added memory usage tracking
5. **Performance Metrics**:
    - Added timing metrics for critical operations
    - Implemented performance counters
    - Created monitoring endpoints for system health checks

6.2 Frontend Architecture

The FalsoPay frontend is built using React with a component-based architecture. It provides a responsive, user-friendly interface for all payment operations.

Key Features:

- **Component-Based Design**: Reusable UI components for consistent user experience
- **State Management**: Centralized state management using Redux
- **Responsive Design**: Mobile-first approach for all user interfaces
- **Accessibility**: WCAG 2.1 compliant for inclusive user experience

Error Pages Optimization

The frontend error pages have been enhanced with the following features:

1. **Animated 404 Page**:
    - Implemented using Framer Motion for smooth animations
    - Created a visually appealing design consistent with FalsoPay branding
    - Added helpful navigation options for users
    - Optimized for all device sizes
2. **Animated 401 Unauthorized Page**:
    - Implemented secure authentication redirection
    - Added clear explanation of authentication requirements
    - Created smooth transition animations
    - Integrated with the authentication system for seamless user experience

6.3 WebSocket Server

The WebSocket server handles real-time communication between clients and the server, enabling features like instant notifications and transaction status updates.

Optimization Measures:

1. **Connection Management**:
    - Implemented connection limits to prevent resource exhaustion
    - Added automatic cleanup of inactive connections
    - Created connection pooling for efficient resource usage
2. **Memory Management**:
    - Optimized message buffer sizes
    - Implemented periodic garbage collection
    - Added memory usage monitoring
3. **Heartbeat Mechanism**:
    - Created a ping/pong protocol for connection health checks
    - Implemented automatic disconnection of unresponsive clients
    - Added configurable heartbeat intervals
4. **Error Handling**:
    - Improved error detection and reporting
    - Added graceful error recovery mechanisms
    - Implemented detailed logging for troubleshooting

5. **Security Enhancements**:
    - Added WebSocket protocol validation
    - Implemented message authentication
    - Created rate limiting to prevent abuse

6.4 Nginx Configuration

A comprehensive Nginx configuration has been implemented to provide efficient request routing, load balancing, and security features.

Key Features:

1. **Request Routing**:
    - Configured routing for the frontend React app
    - Set up API request forwarding to the PHP backend
    - Implemented WebSocket proxy for real-time communication
2. **Performance Optimization**:
    - Enabled content compression
    - Configured browser caching
    - Implemented HTTP/2 for improved performance
3. **Security Features**:
    - Set up TLS/SSL with strong cipher suites
    - Added HTTP security headers
    - Implemented rate limiting and request filtering
4. **Load Balancing**:
    - Configured upstream server groups
    - Implemented health checks
    - Set up session persistence
5. **Logging and Monitoring**:
    - Configured structured access logs
    - Set up error logging
    - Added monitoring endpoints

6.5 Error Handling and Optimization

FalsoPay implements a comprehensive error handling strategy to ensure system stability and provide clear feedback to users.

Key Components:

1. **Centralized Error Logging**:
    - Structured error logs with contextual information
    - Error categorization and severity levels
    - Integration with monitoring systems
2. **User-Friendly Error Messages**:
    - Clear, actionable error messages
    - Localized error descriptions
    - Guidance for error resolution
3. **Graceful Degradation**:
    - Fallback mechanisms for service failures
    - Partial functionality during system degradation
    - Clear communication of system status
4. **Performance Monitoring**:
    - Real-time performance metrics
    - Alerting for performance degradation
    - Historical performance data analysis
5. **Continuous Optimization**:
    - Regular performance testing
    - Code profiling and optimization
    - Infrastructure scaling based on load patterns

8. References

Technical Resources

1. React Documentation - https://reactjs.org/docs/getting-started.html
2. PHP Documentation - https://www.php.net/docs.php
3. MySQL Documentation - https://dev.mysql.com/doc/
4. WebSocket Protocol - RFC 6455 - https://datatracker.ietf.org/doc/html/rfc6455
5. JWT Authentication - https://jwt.io/introduction
6. OWASP Security Guidelines - https://owasp.org/www-project-web-security-testing-guide/
7. Nginx Documentation - https://nginx.org/en/docs/
8. Docker Documentation - https://docs.docker.com/
9. Redis Documentation - https://redis.io/documentation
10. Framer Motion Documentation - https://www.framer.com/motion/

Academic Resources

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
2. Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley.
3. Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.
4. Newman, S. (2015). Building Microservices. O'Reilly Media.
5. Nygard, M. T. (2007). Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf.

Industry Standards

1. ISO 27001 - Information Security Management
2. PCI DSS - Payment Card Industry Data Security Standard
3. GDPR - General Data Protection Regulation
4. WCAG 2.1 - Web Content Accessibility Guidelines
5. PSR Standards - PHP Standards Recommendations

Additional Resources

1. Material Design - https://material.io/design
2. Financial Industry Best Practices - https://www.bis.org/publ/bcbs128.pdf
3. UML 2.5 Specification - https://www.omg.org/spec/UML/
4. Software Engineering Body of Knowledge (SWEBOK)
5. Project Management Body of Knowledge (PMBOK)