CS/SE/CE 3354 Software Engineering: Deliverable #2 (Part IV)

UniWallet Banking

https://github.com/TejasRamanujam/3354-Team10

Note to Grader: *If viewing this in a PDF viewer, the "Chapters" on the sidebar should show you each enumerated item of this Deliverable to help navigate the document.*

2. Delegated Tasks by Member for Deliverable #2

1. Jorge Agueros:
    a. Worked on Architectural Design slide
    b. Collaborated on Cost Estimation Slide
2. Aldo Carmona:
    a. Comparison of our work with similar designs
    b. Collaborated on the presentation slides use case diagram, functional and nonfunctional requirements
3. Rohan K Cheruku:
    a. Updating the sequence process diagrams based on previous comments
    b. Emphasizing project objectives and research
    c. Conclusion: evaluating our work and the changes along the way
4. Zunaira Firdous:
    a. Updating the sequence process diagrams based on previous comments
    b. Collaborated on the Project Scheduling
5. Joshua Purushothaman:
    a. Model-View-Controller architecture
    b. Sprint Plan
    c. Updated Software Comparison paper
    d. Conclusion: evaluating our work and the changes along the way
6. Tejas Ramanujam:
    a. Updating the functional requirements based on previous comments
    b. Reworked software comparison paper
    c. Updated cost and pricing work
    d. Reformatted algorithmic estimation points
7. Varun Saravanan:

    a. Unit test code
    b. Project scheduling

---

3. BEGIN: Following this is “Everything required and already submitted in Course Team Project Deliverable #1”.

2. Delegated Tasks by Member

1.1. Deliverable # 1 [Each member’s tasks]

1. Jorge Agueros:
    a. I will work on the functional software requirements.
2. Aldo Carmona:
    a. I will be working on the use case diagram.
    b. GitHub: Create and write Project Scope.md
3. Rohan K Cheruku:
    a. I will collaborate on creating the sequence process diagrams.
4. Zunaira Firdous:
    a. I will collaborate on creating the sequence process diagrams.
5. Joshua Purushothaman:
    a. I will apply an appropriate architectural design pattern based on our project (MVC).
    b. GitHub: Create and write README.md
6. Tejas Ramanujam:
    a. I will be listing the non-functional software requirements.
    b. I will describe the software process model.
    c. GitHub: Initial repository creation
7. Varun Saravanan:
    a. I will be creating the class diagram.

3. List any assumptions and include details

- We assume that this app is to be used for non-corporate transactions under the US banking system, which means that all rules and regulations of the US banking system would apply.

- We also assume that live software maintenance would be handled by a “Bank IT” team. They would handle responsibilities such as app security, server maintenance, and the backing up and restoration of any databases that are used.

4. Address Feedback

- Given that feedback was “Good Job,” we have decided to continue on our current project path.

5. Software Process Models

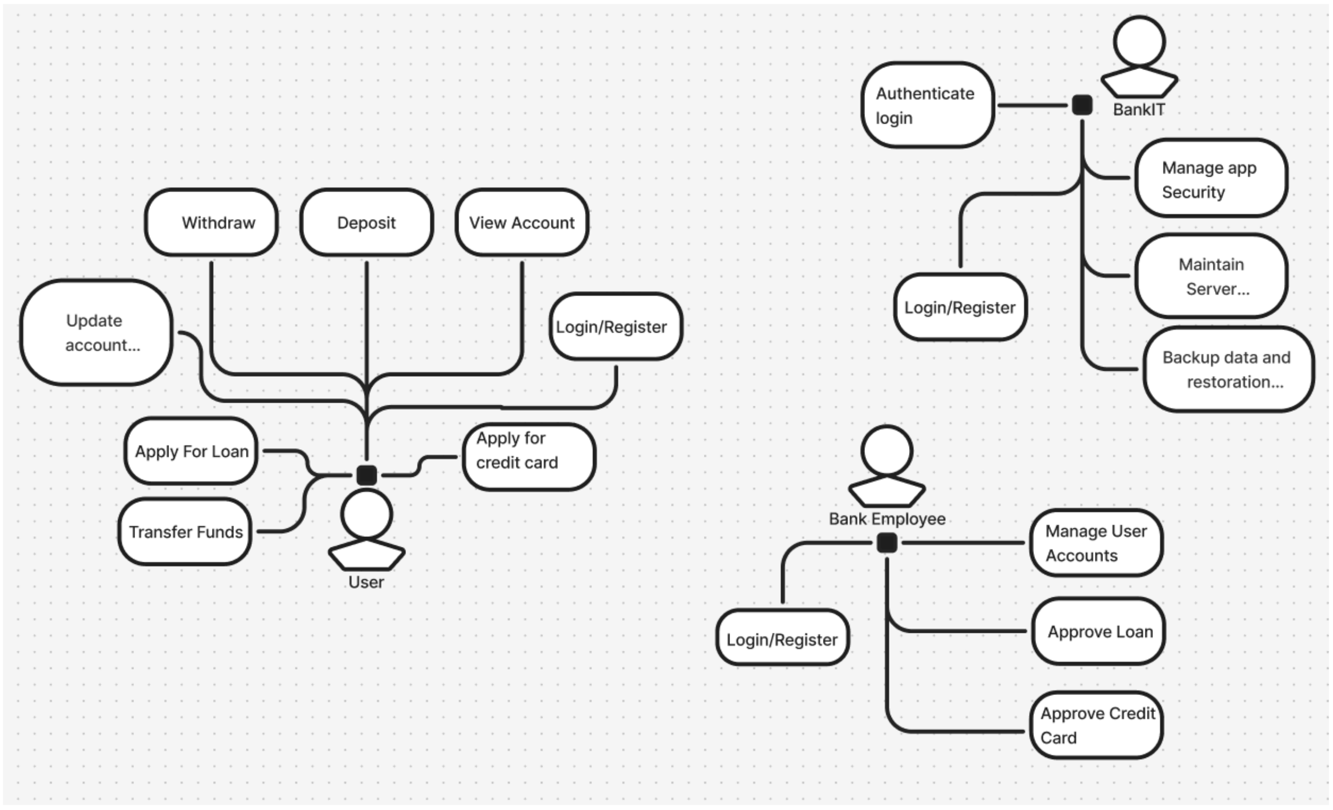
- We have decided to continue with the **waterfall model** as a basis for our project, as the structured and concise format of the model allows for the planning and processing of scheduled activities ahead of time.

6. Software Requirements

- Functional Requirements:
    - The user should be able to add, view, edit, and delete their accounts.
    - The user should be able to deposit, withdraw, and view recent account activity.
    - The system should authenticate logins and manage app security.
    - The system should provide account statements when requested.
    - Bank Employees should have access to customer’s accounts and approve loans.
- Non-Functional Requirements:
    - Usability: User should be able to check balance within 3-4 clicks.
    - Performance: Loads / searches of transactions should be within 3s.
    - Space: App should take up less than 150 MB of storage on the target device.
    - Dependability: 99% projected uptime, along with consistent server status allowing for transactions to roll back.
    - Security: Data should be encrypted and passwords should be hashed appropriately.
    - Environmental: Runs on all current iOS/Android systems.
    - Operational: Admin access should be secure, along with daily database backups.
    - Development: Use of React Native or other app development languages.
    - Regulatory: Must comply with current US banking regulations.
    - Ethical: Data must be secured against misuse / exploitation.
    - Accounting: Daily reports on total financial change / account balance changes.

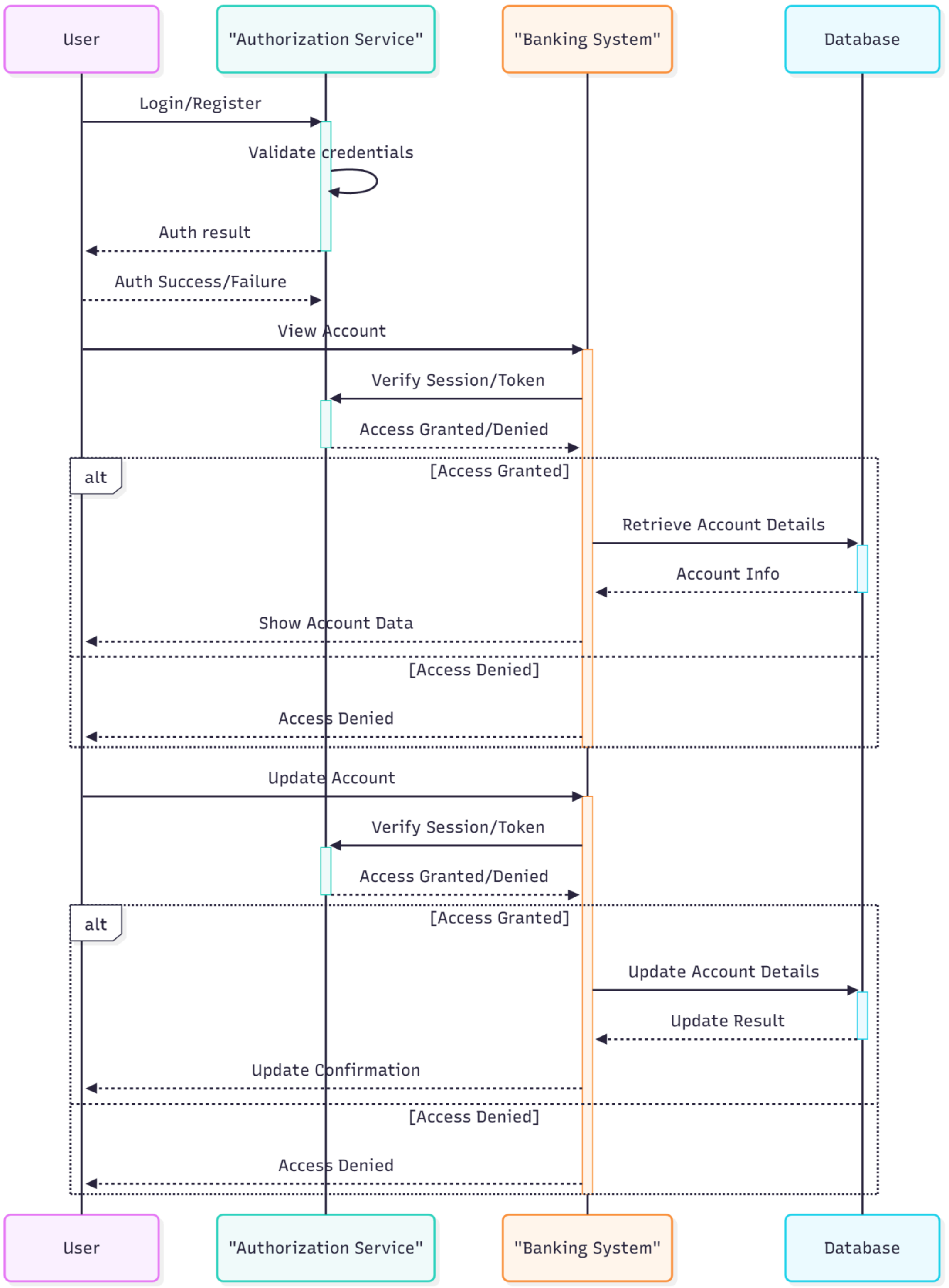
    - Safety: Comply with data protection laws, provide compliant format to authorities.

7. Use Case Diagram

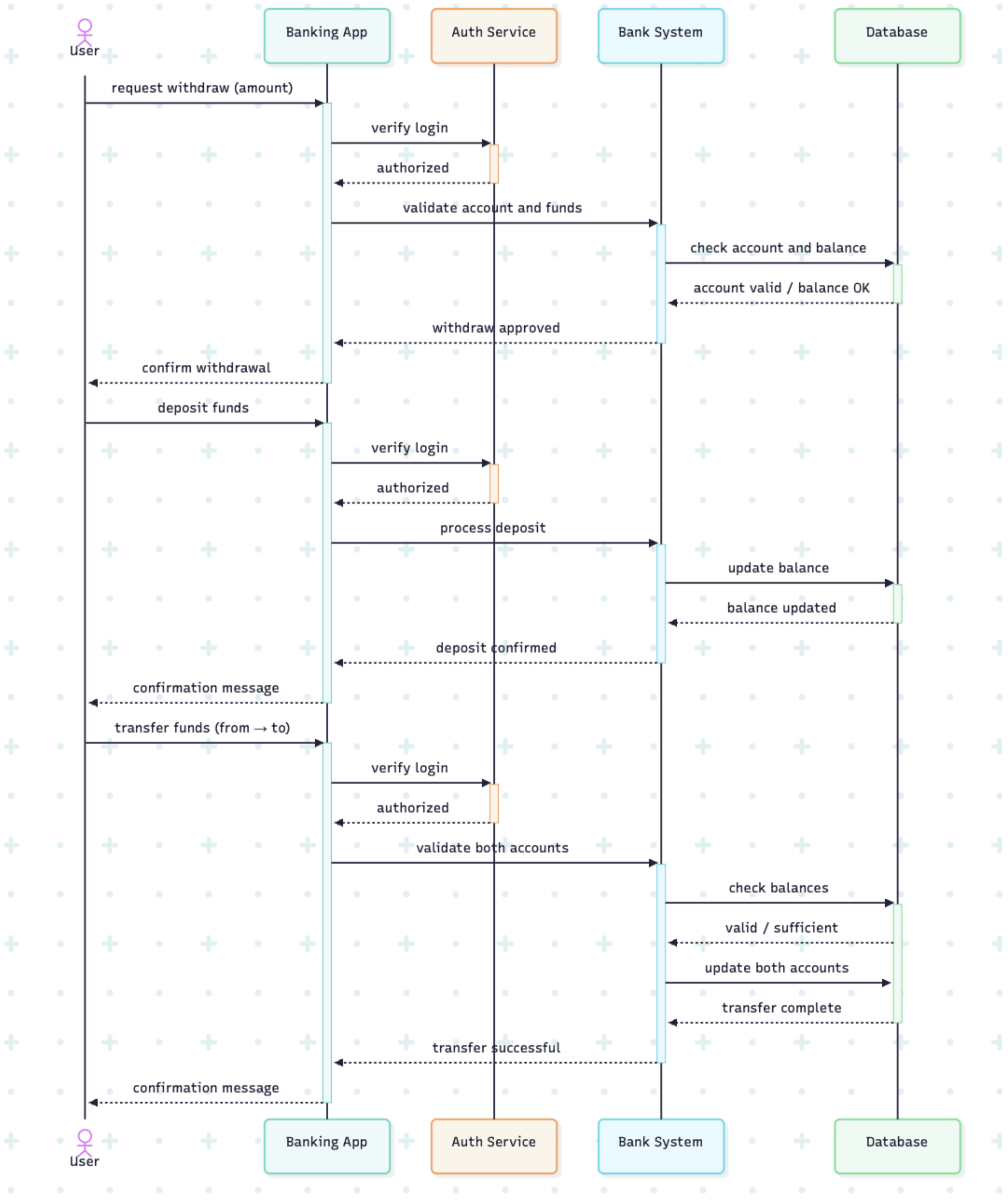


8. Sequence Diagram: The three primary use-cases for our app are:

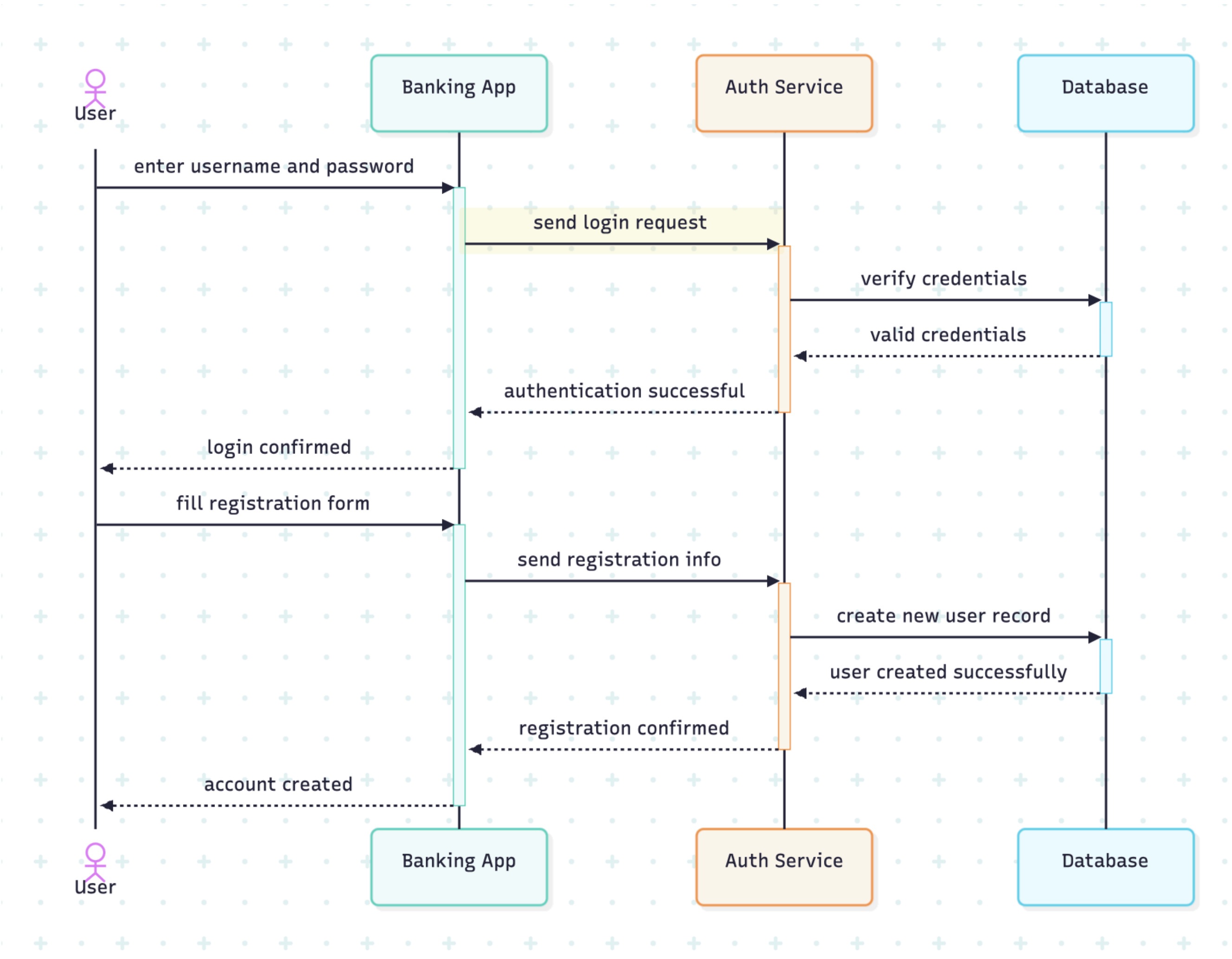
- Update Account



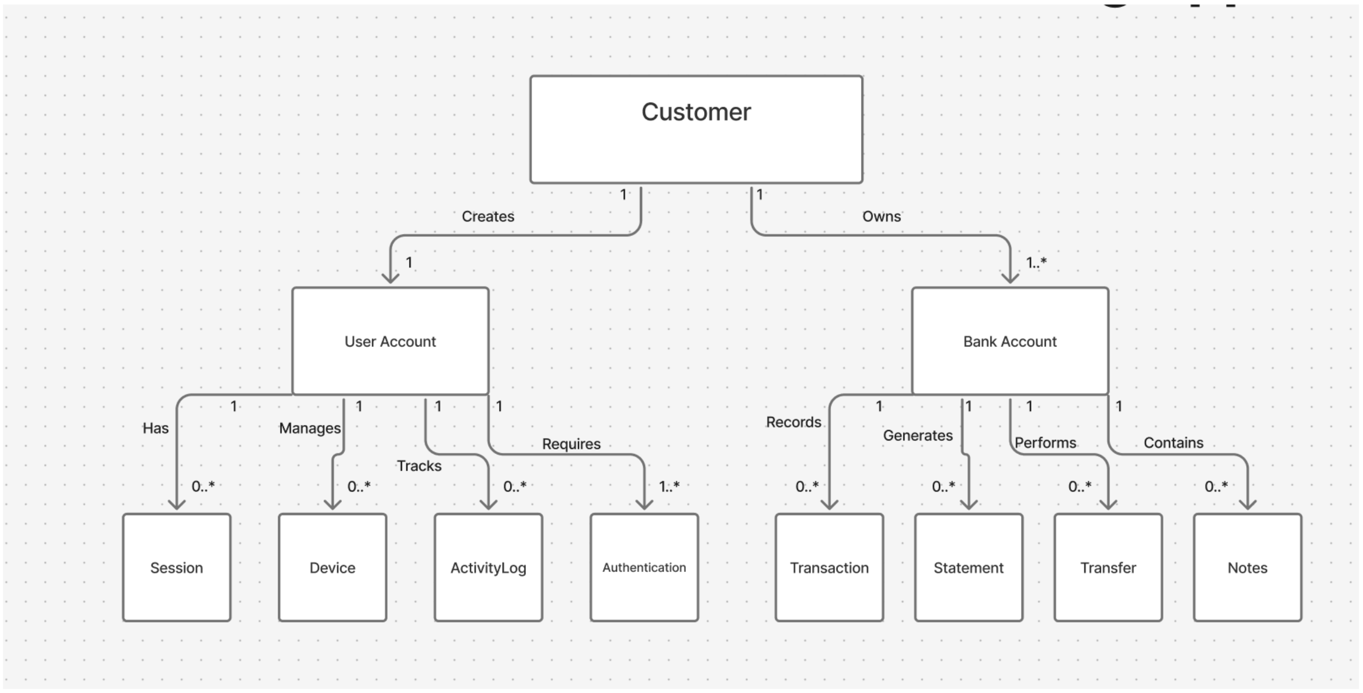
- Transactions

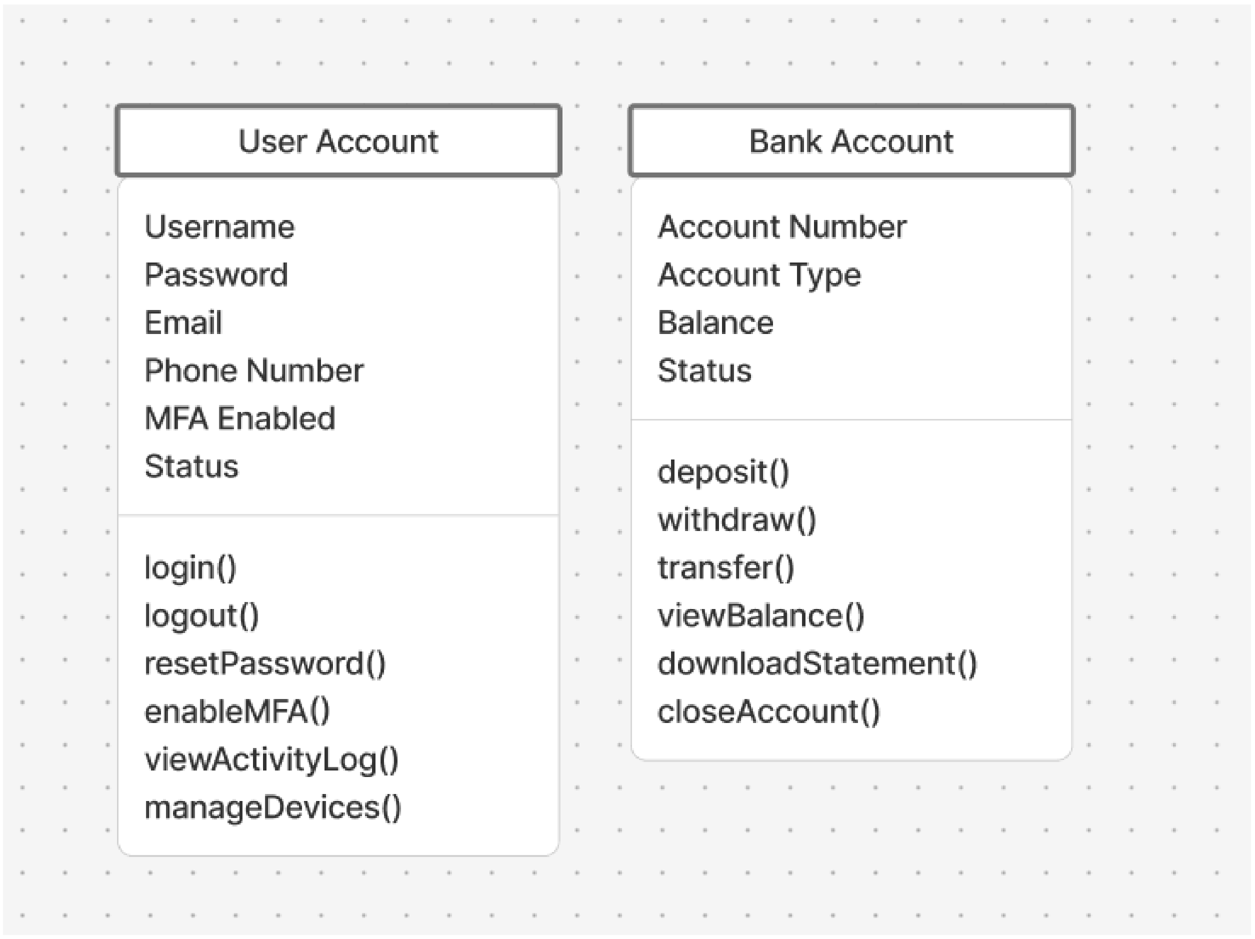


- Login/Register



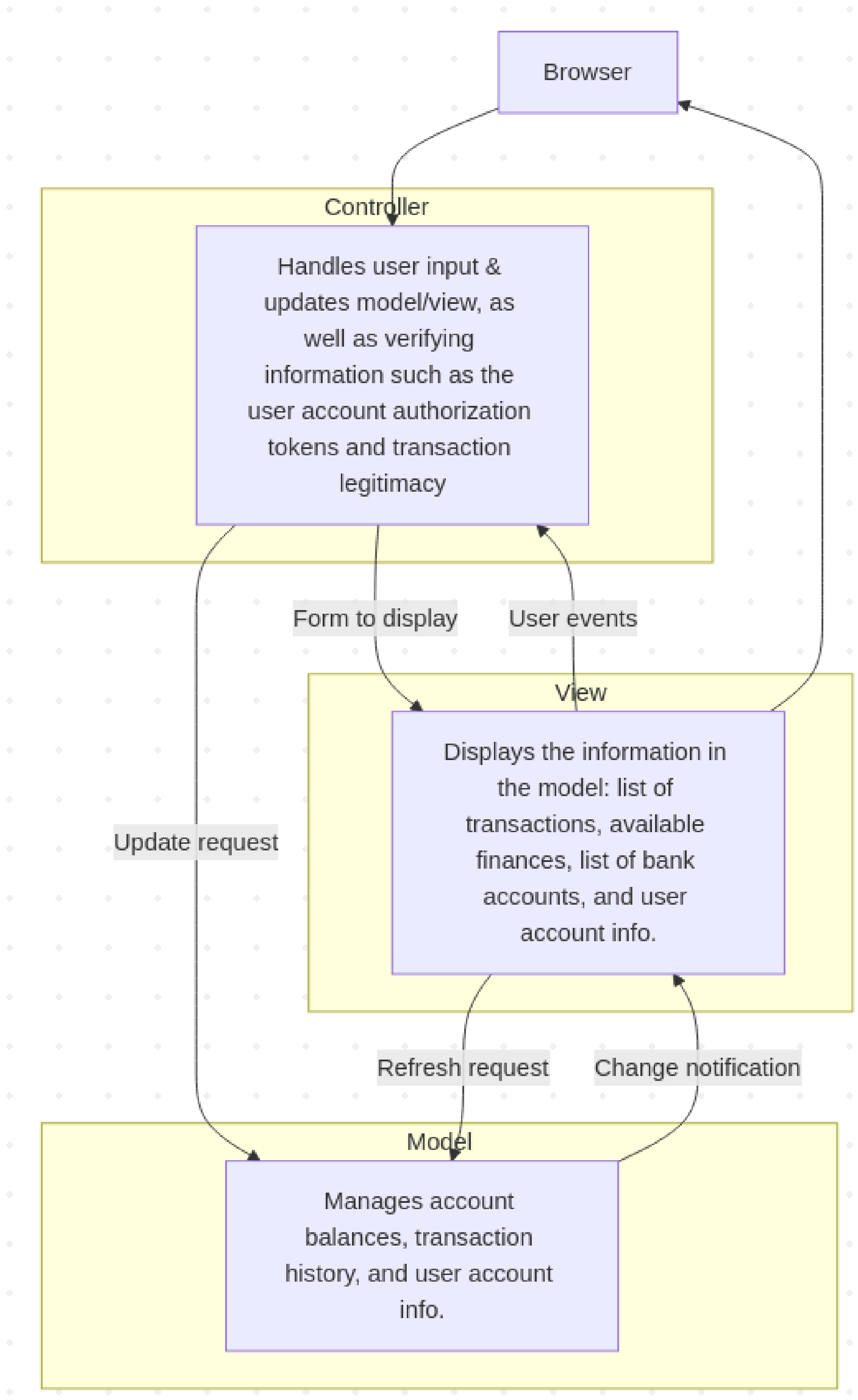
9. Class Diagram





10. Architectural Design

- We have applied the Model-View-Controller (MVC) pattern:


Browser
Controller
Handles user input & updates model/view, as well as verifying information such as the user account authorization tokens and transaction legitimacy
Form to display
User events
View
Displays the information in the model: list of transactions, available finances, list of bank accounts, and user account info.
Update request
Refresh request
Change notification
Model
Manages account balances, transaction history, and user account info.

3. END: Everything above this is “Everything required and already submitted in Course Team Project Deliverable #1”.

---

4. Project Scheduling, Cost, Effort, and Pricing Estimation, Project duration and staffing

4a. Project Scheduling

Project Timeline

- Start Date: January 15, 2026
- End Date: April 9, 2026
- Duration: 12 weeks

Estimation Method: Function Point Analysis

- Banking apps require compliance and security testing.
- Duration accounts for requirements, design, backend, frontend, testing, and deployment phases.

Sprint Plan

Sprint 1 (Weeks 1 - 2):

- Focus: Requirements and Architecture
- Deliverables: Requirements document, architecture diagrams

Sprint 2 (Weeks 3 - 4):

- Focus: Backend Development
- Deliverables: Authentication, account module, database schema

Sprint 3 (Weeks 5 - 6):

- Focus: Backend Development
- Deliverables: Transactions, APIs, validation logic

Sprint 4 (Weeks 7 - 8):

- Focus: Frontend Development
- Deliverables: Navigation, main screens, user flows

Sprint 5 (Weeks 9 - 10):

- Focus: Frontend Integration and Polishing
- Deliverables: API integration, UI refinement, error handling

Sprint 6 (Weeks 11 - 12):

- Focus: Testing and Deployment
- Deliverables: System testing, bug fixes, deployment

Schedule Details

Working from Monday - Friday only:

12 weeks × 5 days = 60 working days and 8 hours per day per developer

Breakdown:

- 6 hours development
- 1 hour meetings/standups
- 1 hour documentation/review

Total Hours:

- 15 developers × 60 days × 8 hours = 7,200 total person-hour
- Effective per developer: 480 hours

4b. Cost, Effort, and Pricing Estimation

Method Selected: Function Point (FP) Analysis

Step 1: Count functions

- Customer transaction inputs: 42
- Account & statement outputs: 20
- Balance & transaction inquiries: 20
- Banking data files: 14
- External financial interfaces: 8

Step 2: Determine complexity:

Function Category	**Count**	**Simple**	**Average**	**Complex**	**Count × Complexity(simple)**
1. Customer transaction inputs	42	3	4	6	126
2. Account & statement outputs	20	4	5	7	80
3. Balance & transaction inquiries	20	3	4	6	60
4. Banking data files	14	7	10	15	98
5. External financial interfaces	8	5	7	10	40

Step 3. Compute gross function point (GFP).
GFP = 126 + 80 + 60 + 98 + 40 = 404

Step 4. Determine processing complexity (PC).

PC1 - Does the system require reliable backup and recovery? Score: 5
Banking application: must maintain transaction integrity in failure recovery.

PC2 - Are data communications required? Score: 5
With online and mobile access, API calls, and integrations to third-party services, communication can't be avoided.

PC3 - Are there distributed processing functions? Score: 4
Likely some distribution, such as mobile clients, back-end services, and database clusters.

PC4 - Is performance critical? Score: 5
The response time for a transaction and real-time balances is critical for user experience.

PC5 - Will the system run in an existing, heavily utilized operational environment? Score: 3
May be deployed on shared/cloud infra with limited resources.

PC6 - Does the system require online data entry? Score: 5
Users enter transfers, payments, etc.-it is essential that this be done online.

PC7 - Does online data entry require multi-screen / multi-step transactions? Score: 4
Bill pay, multi-factor auth, and span multiple steps/screens.

PC8 - Are the master files updated online? Score: 5
Account data and transactions are updated live.

PC9 - Are the inputs, outputs, files, or inquiries complex? Score: 4
Reports and queries (transaction histories, statements) are substantial.

PC10 - Is the internal processing complex? Score: 2
Business logic exists, but the complexity is simple.

PC11 - Is the code designed to be reusable? Score: 1
It will be one deployment with small changes, meaning reuse is not a factor.

PC12 - Are conversion and installation included in the design? Score: 0
Neither is included in the design.

PC13 - Is the system designed for multiple installations in different organizations? Score: 2
Could be adapted later, but not a primary goal.

PC14 - Is the application designed to facilitate change & ease of use? Score: 1
Ease of use matters, but it is not essential to system architecture.

PC: 5 + 5 + 4 + 5 + 3 + 5 + 4 + 5 + 4 + 2 + 1 + 0 + 2 + 1 = 46

Step 5. Compute processing complexity adjustment (PCA).

PCA = 0.65 + 0.01 * (PC total)
PCA = 0.65 + 0.01 * 46 = 1.11

Step 6. Compute function point (FP).

FP = GFP × PCA
FP = 404 × 1.11 = 448 FP

Estimated Effort:

Assuming productivity is 10 FP/person-month

- Productivity = 10 FP/person-month
- Effort = FP / Productivity
- Effort = 448 / 10 = 44.8 person-months

Project Duration:

- Team size = 15 developers
- Duration = Effort/Team size
- Duration = 44.8 / 15 = 2.99 months = 12 weeks

4c. Hardware Costs

For our application's hardware, we determined that we'd need a server to host it on, a database cloud service, storage for customer data, as well as networking costs for the server itself.

Category	**Details**	**Cost (6 Months)**
Application Server	AWS EC2 t3.medium (web backend + API server)	$180
Oracle Database Cloud Service	Oracle Autonomous Transaction Processing (1 OCPU, 1 TB storage)	$450
Storage	Additional S3 buckets / logs (100 GB)	$14
Networking	Data transfer for API + DB traffic	$60
Total Hardware Cost		**$704**

4d. Software Costs

Since our application is a collaboratively developed GUI / mobile app, we determined that we'd need software for:

- Development (IDEs)
- Hosting (DBMS, server-side deployment tools)

- Collaboration (GitHub/VCS, professional chat software, project management and sprint tracking software)

Software / Tool	**Purpose**	**Cost**
Apple Developer Program	Required for publishing the iOS app to the App Store	$99/year
React	Web front-end	$0 (open-source)
GitHub Team Plan	Repository + CI for 15 users	$180
Postman Team Plan	API Testing	$57
Jira Software	Project Management	$630
Slack Pro	Team communication	$405
Oracle SQL Developer	DB Management GUI	$0 (free)
Total Software Cost		**$1,371**

4e. Cost of Personnel

Developer Cost:

- Pay rate: $35/hour
- Work duration: 60 days × 8 hours/day = 480 hours per developer
- Cost per developer: 480 × $35 = $16,800
- Team of 15 developers: 15 × $16,800 = $252,000

Training Cost

- 4 trainees × 50 hours × $20/hour = $4,000

5. Test Plan

5a. Test Plan Description

Unit tested: Bank Withdrawal Method

What it does:
- Checks if withdrawal amount is valid
- Verifies account has enough money
- Updates account balance
- Records the transaction

Testing approach:
- Automated testing with JUnit
- Test normal withdrawals
- Test multiple withdrawals
- Test edge cases (zero, negative, exact balance)

5b. Example unit tests

Java Code:

```java
package bank;

public class BankAccount {
   private double balance;

   public BankAccount(double initialBalance) {
       this.balance = initialBalance;
   }

   public boolean withdraw(double amount) {
       if (amount <= 0 || amount > balance) {
           return false;
       }
       balance -= amount;
       return true;
```

```java
    }

    public double getBalance() {
        return balance;
    }
}
```

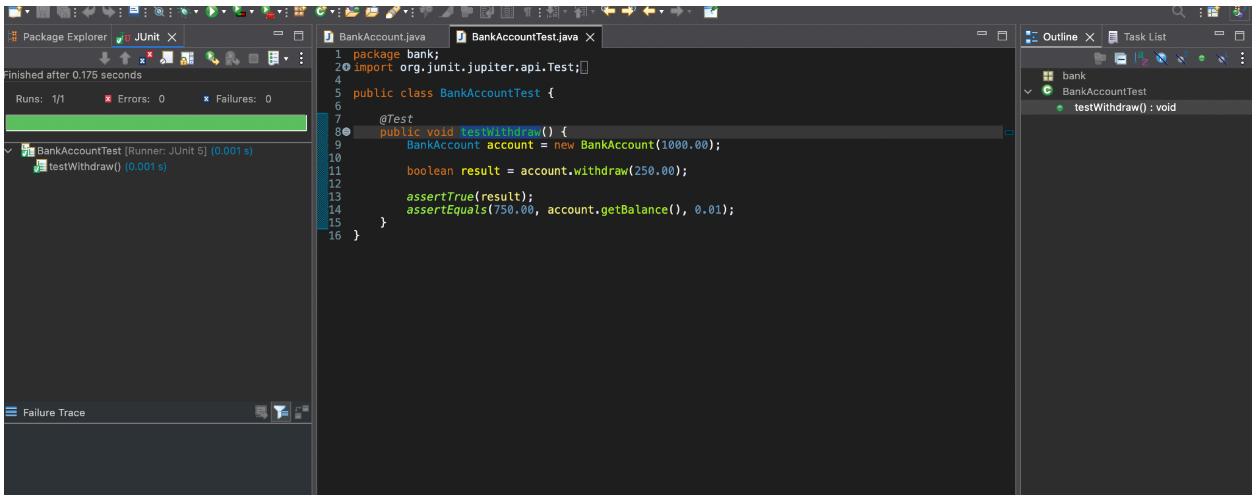
JUnit Code:

```java
package bank;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class BankAccountTest {
    @Test
    public void testWithdraw() {
        BankAccount account = new BankAccount(1000.00);

        boolean result = account.withdraw(250.00);

        assertTrue(result);
        assertEquals(750.00, account.getBalance(), 0.01);
    }
}
```



```
Finished after 0.175 seconds
Runs: 1/1    Errors: 0    Failures: 0
BankAccountTest [Runner: JUnit 5] (0.001 s)
    testWithdraw() (0.001 s)
```

```java
package bank;
import org.junit.jupiter.api.Test;

public class BankAccountTest {

    @Test
    public void testWithdraw() {
        BankAccount account = new BankAccount(1000.00);

        boolean result = account.withdraw(250.00);

        assertTrue(result);
        assertEquals(750.00, account.getBalance(), 0.01);
    }
}
```

```
bank
BankAccountTest
    testWithdraw() : void
```

5c. Results of test case

For this test case, we are doing a normal withdrawal. We are withdrawing $250 from our initial balance of $1000. Since there is enough money in the bank account to withdraw $250, the transaction is completed. The remaining balance of $750 is left. The test case passes since the withdrawal is valid.

6. Comparison of our work with similar designs

The software that our group decided to create was a banking app. We compared our app design to several different banking software applications available on the market such as those from companies like Bank of America, Chase, and Wells Fargo. These various banking apps are designed to different customer and business needs, and form a background for us to develop our app from.

While our group utilized the Model View Controller architecture to maintain a clear separation of concerns, other companies often require more complex systems to handle scale. For example, Capital One and Monzo utilize microservices as their architecture of choice, where the application is broken down into thousands of independent services, such as separate services for accounts, transfers, and security [1]. JPMorgan is another example that supports this approach by employing services focusing on scalability across regions, making sure that a failure in one feature does not crash the entire banking platform. This is different from our MVC approach, which creates a stronger dependency between the user and the app itself - one that could potentially result in larger failures if shutdowns occur. The MVC architecture, however, allows us to model our GUI and its coupling with the business logic, which allows for well-defined interactions between the backend and frontend.

Although our app contains many of the basic features that other apps do, it is not as robust. Many of these other apps contain features such as transferring money, banking AI helpers, buying and selling stocks in a brokerage account, and providing documents for filing taxes [2]. This allows for their software to bring various additional features to their customers, while still remaining true to the purpose of their banking needs. However, companies like Bank of America, Chase, and Wells Fargo reside in America, so they must comply with the laws and regulations the government has in place, which our group's banking app would also need to comply with if deployed in the United States [3].

DDI Development created a case study on how they developed online banking software [4]. The case study shared the most essential requirements needed for banking software: Security, Speed, Usability, Profitability, and Scalability. Our banking app aimed to hit all but one of these requirements, which is profitability. This is still an important requirement, as the cost of an app is not just the upfront development cost; it also includes the cost of maintaining the software for years to come.

One of the main things that separates their software implementation from our group's is that their app used multiple technologies, such as Oracle DB, Swift (iOS), and React (Web App), which would likely use different software architectural designs [4]. While we do also design our app around using Oracle DB, our implementation only used one framework for the GUI: React Native. This allows us to unify the GUI design over any mobile operating system (Android, iOS). To be able to implement more features in banking software, an approach that uses multiple architectural designs with clear separation would aid in creating software that is feature-rich and can reliably conform to the requirements set at the beginning of the software development process.

Another comparison that can be made is that the use case they identified closely matches the use case diagram for our banking software. It is assumed that a bank employee manages user accounts, manages loans, and manages credit cards. The user would have the same use cases as in our banking app. These use cases rely on secure data transmission as a key aspect, which while not touched upon in our presentation, can be tackled using techniques like threat modeling and audit systems [5].

Our current implementation of banking software was developed using software development processes that we can assume that most major companies that have similar software have done before. We as a group chose these processes as we determined that these would overall fit the design that we wanted to implement. Although, our current implementation, while rudimentary, has a good foundation for creating fully-fledged software that could potentially launch to the public in the future.

7. Conclusion

Throughout the development of our banking application, a few key adjustments were made as the project progressed, leading us to deviate slightly from our original plan. One of the most significant changes was our shift from a simple client-server architecture to an MVC architectural pattern. This update allowed us to better organize responsibilities within the system, improve scalability, and support clearer separation of concerns. We also modified our project timeline, reducing the development period from twenty-four weeks to twelve weeks to align with our available budget, which included considerations such as employee salaries and software costs for tools like Xcode, GitHub, and Swift. In refining our documentation, we ensured that our requirements became fully atomic, each representing a single, testable behavior, and we standardized the language of our requirements by using “shall” for desirable behaviors and “must” for mandatory ones. Finally, we enhanced our sequence diagrams by adding activation bars and lifelines that accurately reflect the duration of each component’s engagement in the process.

These changes were justified by the need for efficiency, clarity, and long-term maintainability. Transitioning to MVC provided a more robust structural foundation, especially as the complexity of our features increased. Shortening the project timeframe was necessary to remain within budget, and this constraint encouraged us to prioritize essential features and streamline our workflow. Improving requirement phrasing and structure strengthened our ability to validate and test the system, reducing ambiguity and preventing miscommunication. Updating our sequence diagrams with proper activation bars and lifelines improved the accuracy of our modeling and made system interactions easier to interpret. Together, these decisions ensured that our final design was not only feasible within our resources but also more organized, testable, and aligned with industry best practices.

8. References

References

[1] O. Beattie, “Building a modern bank backend,” *Monzo*, Sept. 19, 2016. https://monzo.com/blog/2016/09/19/building-a-modern-bank-backend. (Accessed: Nov. 22, 2025.)

[2] R. Pandya, “Banking app development: Lessons from Chase, Wells Fargo & Bank of America,” *Webelight Solutions*, Jul. 18, 2025. https://www.webelight.com/blog/banking-app-development-lessons-from-chase-wells-fargo-bank-of-america. (Accessed: Nov. 18, 2025.)

[3] Bank of America, “U.S. consumer privacy notice,” *Bank of America*. https://web.bankofamerica.com/en/privacy/consumer-privacy-notice. (Accessed: Nov. 22, 2025.)

[4] DDI Development, “Case study: How we developed an online banking software,” *DDI Development*, 2023. https://ddi-dev.com/blog/case/how-we-developed-an-online-banking-software/. (Accessed: Nov. 16, 2025).

[5] C. Shi and D. Bartkevicius, “How we secure Monzo’s banking platform,” *Monzo*, Mar. 30, 2022. `https://monzo.com/blog/2022/03/31/how-we-secure-monzos-banking-platform. (Accessed: Nov. 19, 2025.)