

COMP 354

Design Document for myMoney

Team PA-PK

March 12, 2018

Table 1: Team

Name	ID Number
Anne-Laure Ehresmann	27858906
Marc-Antoine Dube	40029307
Kadeem Caines	26343600
Abdel Rahman Jawhar	27192142
Keith Dion	40036340
Hrachya Hakobyan	40041555
Andrew-Smith	40034936
Dongyu Chen	27241909
Yauheni Karaniuk	40005680
Renny Xu	40005262
Wei Wang	40041116

Contents

1	Introduction and Purpose	4
2	Architectural Design	4
2.1	Architectural Diagram	4
2.2	Subsystem Interface Specifications	5
	SignUpController Interfaces	5
	AccountDetailsController Interfaces	6
3	Detailed Design	7
3.1	Class Diagram	7
3.2	Glossary of Domain Concepts	13
3.3	Subsystem X	14
	Detailed Design Diagram	14
	Units Description	14
4	Dynamic Design Scenarios	14
4.1	Dynamic Models of System Interface	14
	Use Case 1: Create User Account	15
	Use Case 5: View Transactions for Specific Bank Account	16
	Use Case 6: View All Transactions from all Bank Accounts	17
5	Reference	18

List of Figures

1	Class Diagram	12
2	Use case 1 Sequence Diagram	15
3	Use case 5 Sequence Diagram	16
4	Use case 6 Sequence Diagram	17

List of Tables

1	Team	1
---	----------------	---

2	Glossary of Domain Concepts	13
---	---------------------------------------	----

1 Introduction and Purpose

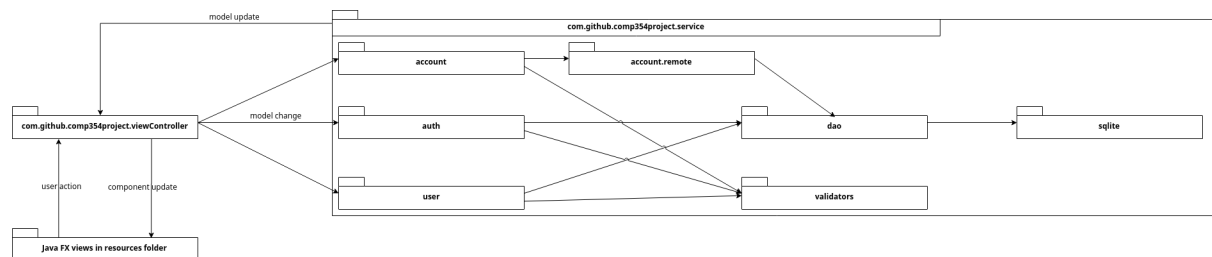
The primary goal of this project is to develop the myMoney application. The goal of this design document is to provide all the details of the Architectural Design (AD), Detailed design (DD) and the Dynamic Design Scenarios (DDS) for the application. The AD section focuses on the high-level project decomposition, the DD describes the overall system design including the UML design divided into subsections and the DDS will display how the subsystems interact with one another in order to produce system-level services.

2 Architectural Design

The myMoney application uses the Model-View-Controller (MVC) pattern to read, validate and modify the objects. The front-end, which is the view interacts with view controllers when events are triggered. These events trigger functions in the services package where the model is then modified.

In the service packages, there is a package named remote which is used to represent an external system via an API to connect to bank accounts from different companies.

2.1 Architectural Diagram



This design represents the MVC pattern discussed previously. The big advantage of this design is that everything is separated with interfaces which makes it easier to use different implementations, modify the features and create or mock tests.

The `com.github.comp354project.service` package is the main subsystem where most of the logic happens. The data validation and processing is done there. It connects to an SQLite database to persist the data. The `com.github.comp354project.viewController` calls this package to update the view and the model.

The `com.github.comp354project.service` package `account.remote` package is a subsystem to our services which is meant to mock an API call to systems outside of ours like banks or credit card companies. Because we don't have access to these APIs for real, obviously, the data is persisted in the same SQLite database as the rest of the system.

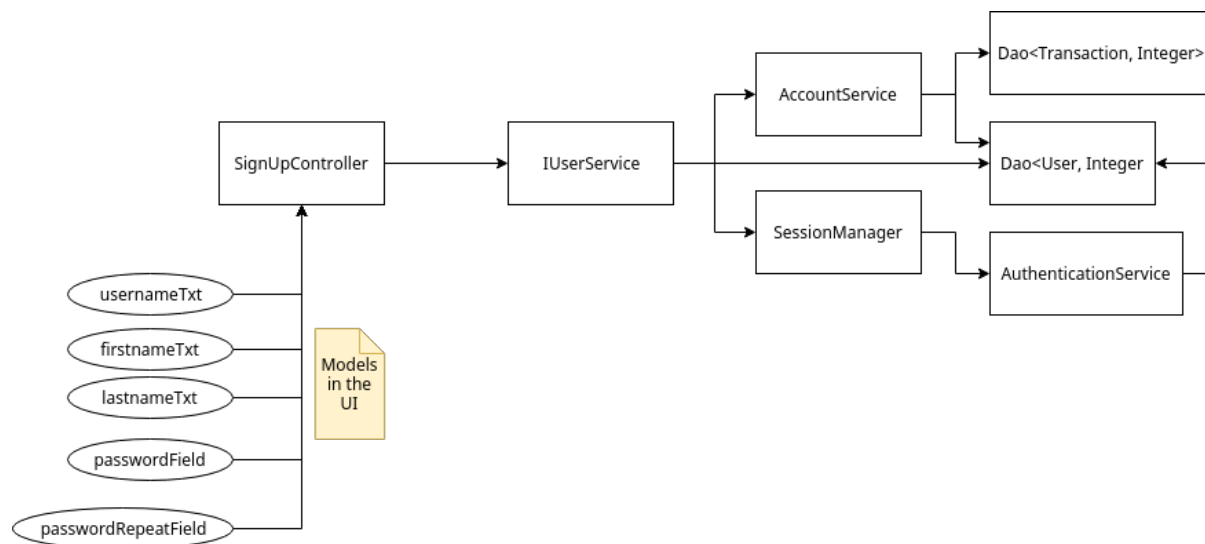
2.2 Subsystem Interface Specifications

Specification of the software interfaces between the subsystems, i.e. specific messages (or function calls) that are exchanged by the subsystems. These are also often called “Module Interface Specifications”. Description of the parameters to be passed into these function calls in order to have a service fulfilled, including valid and invalid ranges of values. Each subsystem interface must be presented in a separate subsection.

*Note: The above is a description of what to provide. Need to edit into our own

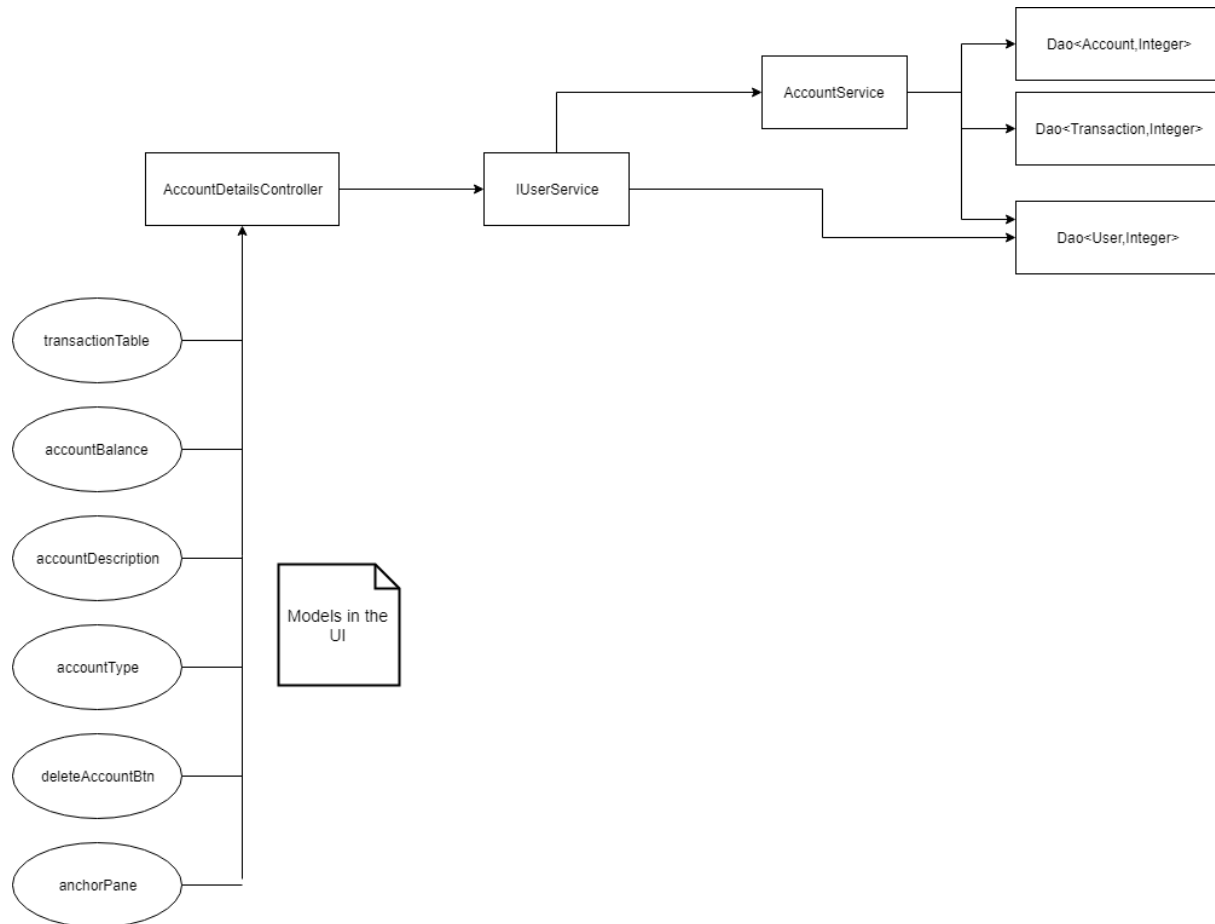
SignUpController Interfaces

Below are the different models and services used in the SignUpController view.



AccountDetailsController Interfaces

Below are the different models and services used in the AccountDetailsController view.



3 Detailed Design

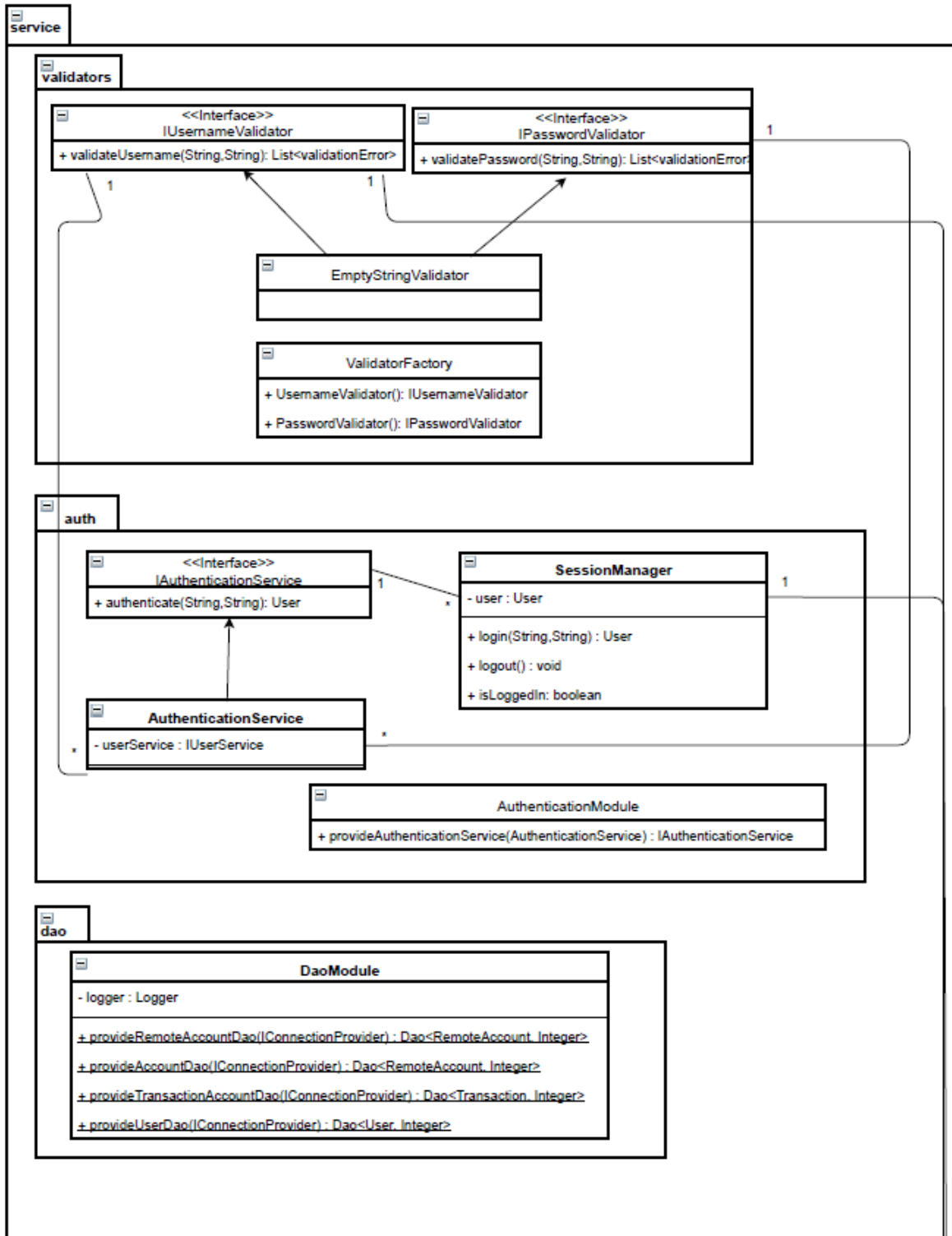
The myMoney system architecture is designed to be easily modified because of the low coupling between the modules. This was done with interfaces and auto injection of dependencies in classes. Each service package has a Module class designed to bind and provide an implementation to an interface. This way, classes are never instantiated directly into each other, but injected. This design pattern is useful because a change in implementation is as simple as creating a new class and change the module binding. The classes that use it and the tests should in no way be changed. Mocking classes for test purposes is also much easier.

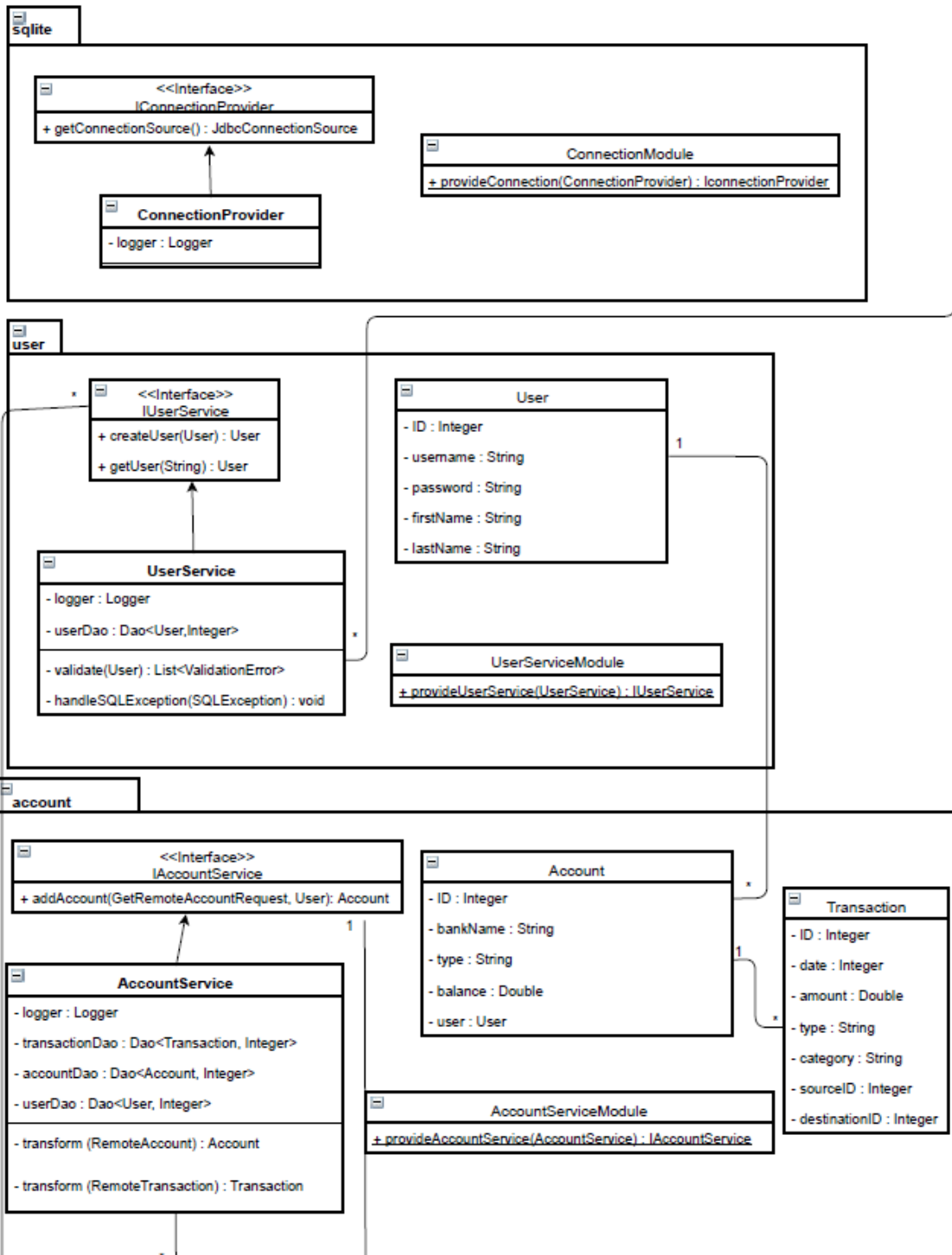
As a side note, we noticed that merge conflicts using git were much less likely to happen because we can each work on different parts of the system without modifying another module.

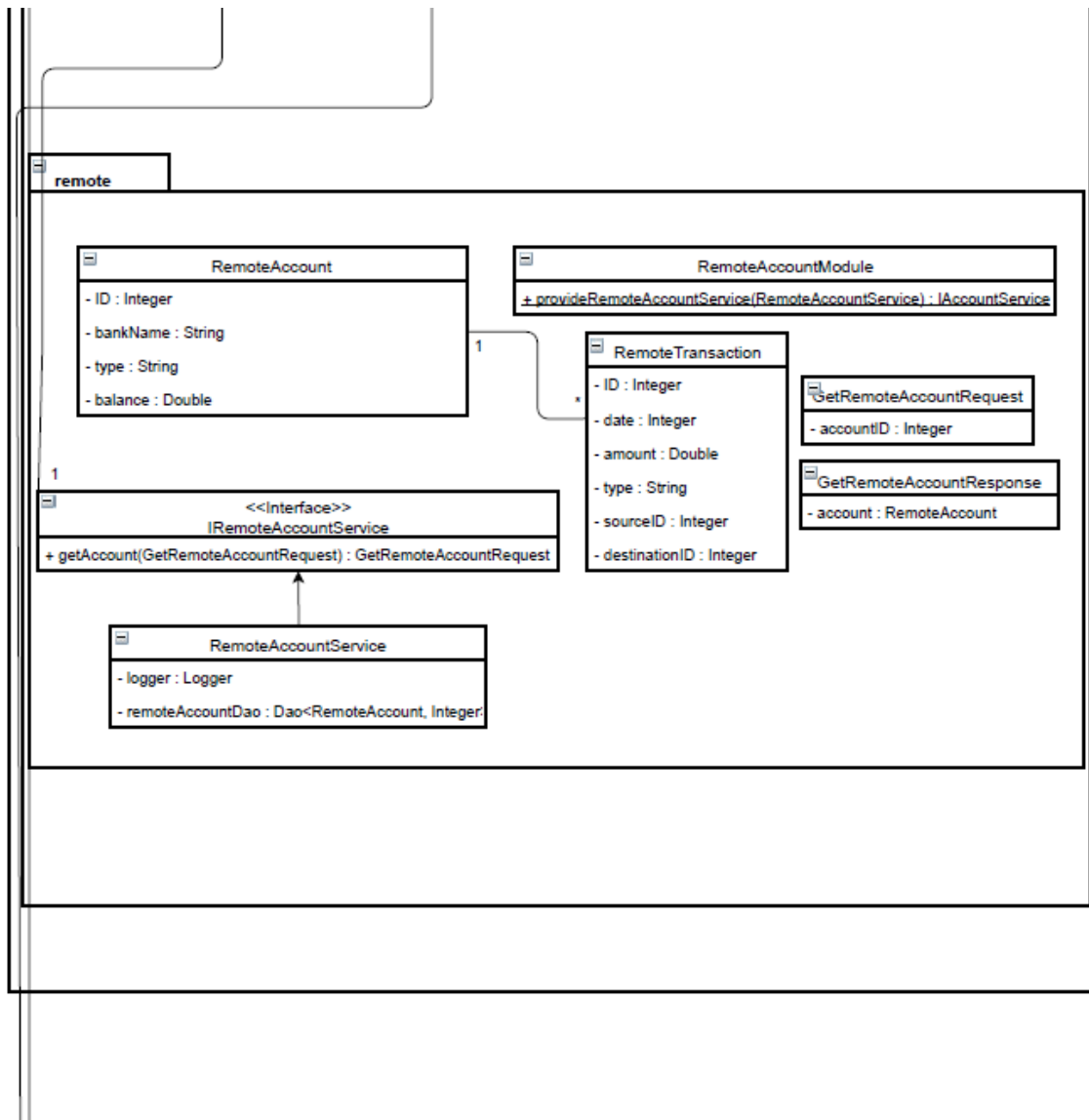
The tool used for this purpose is Dagger version 2.

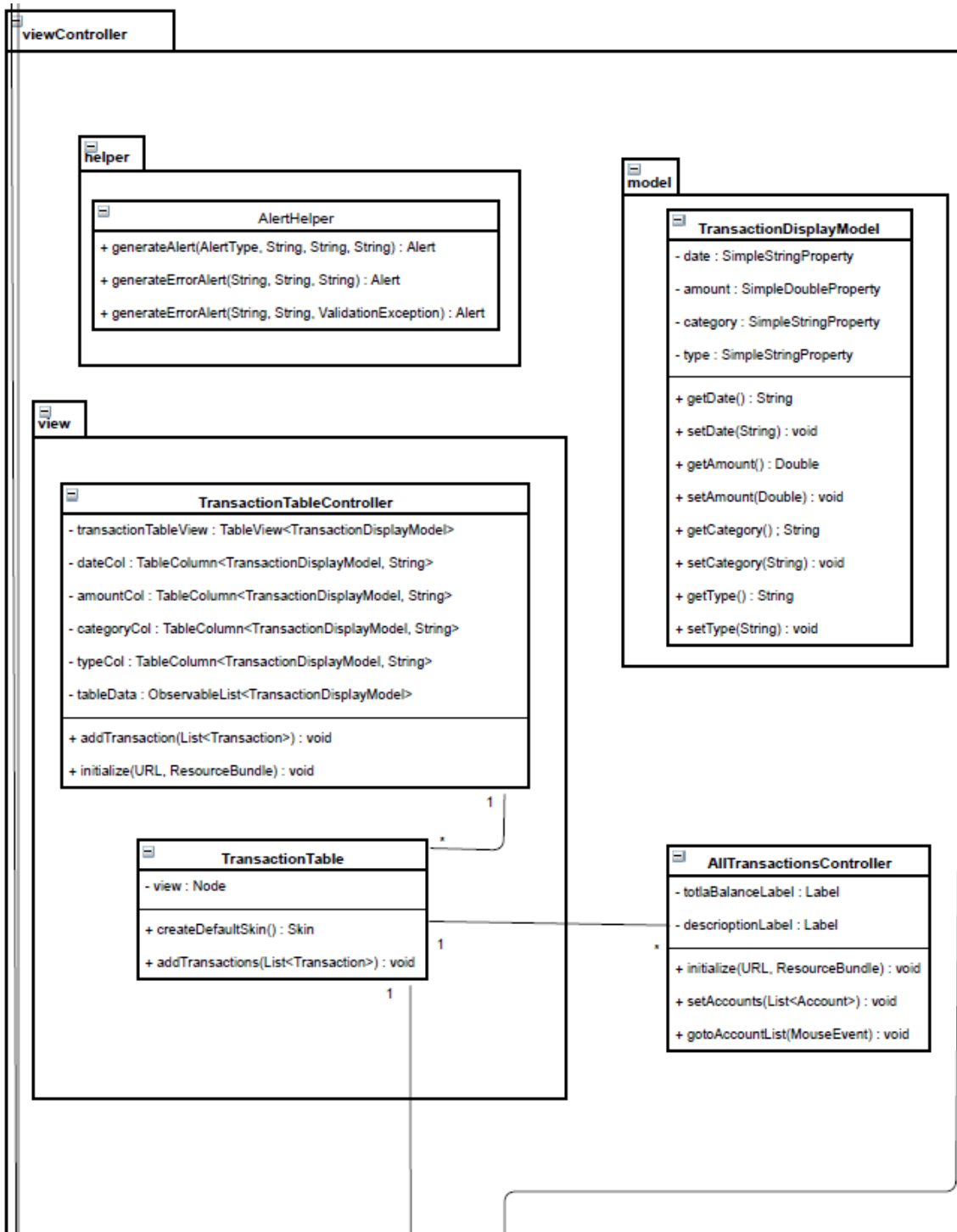
3.1 Class Diagram

In this section we provide the class diagram of our system, useful for the system developers and testers. This is an in depth look at all of the classes within our system see figure 1 below. If a term is unclear, view section 3.2 for the glossary.









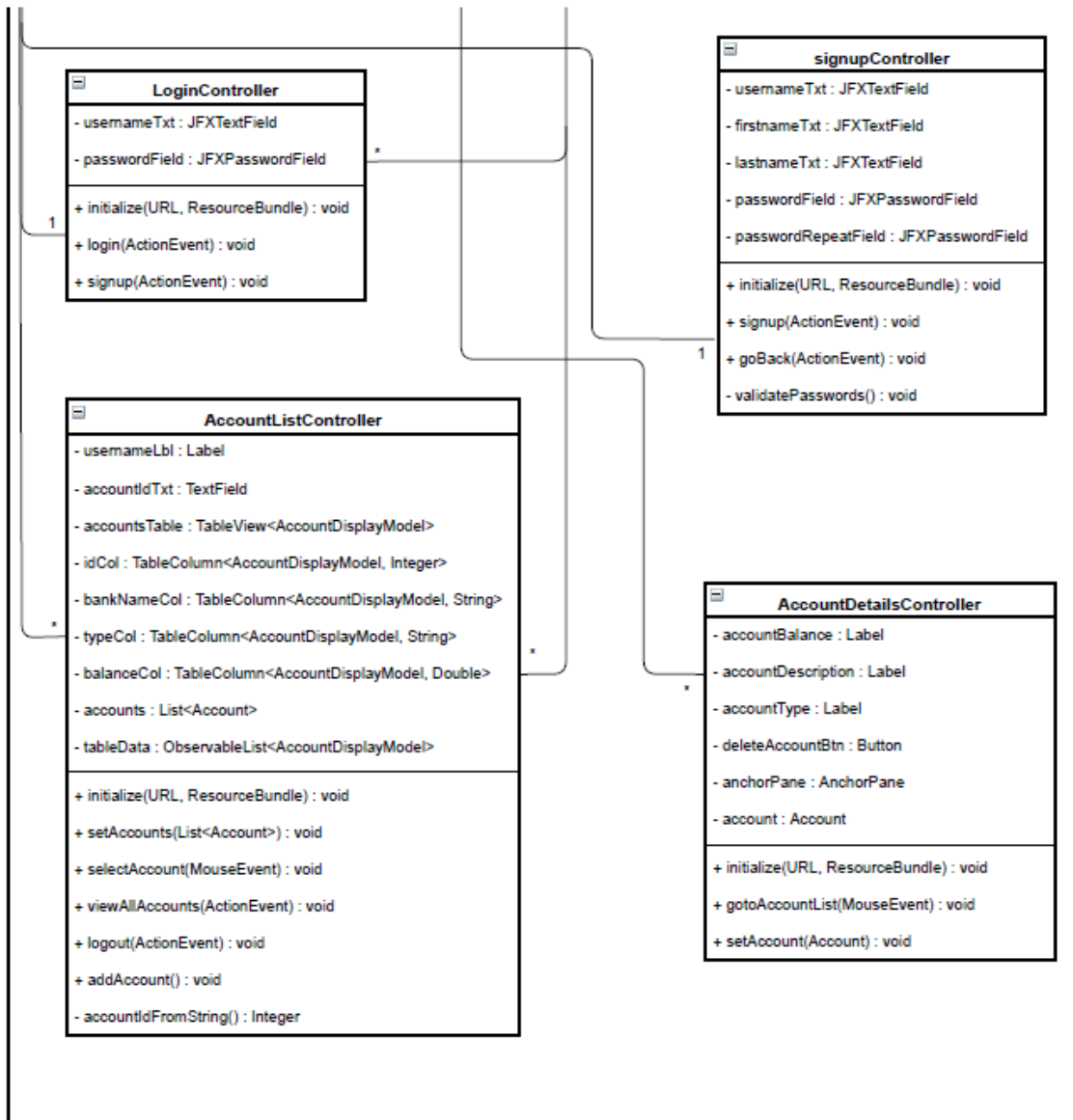


Figure 1: Class Diagram

3.2 Glossary of Domain Concepts

Table 2: Glossary of Domain Concepts

Expression	Definition
User	The person that is using the application and the main provider of requests to the system.
User Account	A data object containing user information. It also contains the various bank accounts that a user may have linked to the system.
Bank Account	A data object containing transactions linked with a specific bank account in a bank institution. One user account may have more than one bank accounts.
Transaction	Any kind of money exchange associated with a bank account.
Transfer	A type of transaction that occurs between two parties.
Deposit	A type of transaction where the owner puts money in his own bank account.
Withdrawal	A type of transaction where the owner of the bank account removes money from his balance.
Database	A local or online container which holds data in an organised, efficient manner.
Server	a computer that is accessible on a network, on which a database and/or system may be hosted. The bank institutions' databases will be hosted on here.
Object-Oriented Programming	A programming paradigm which separates entities into objects, and uses the concept of inheritance of properties, polymorphism of objects, encapsulation of objects. We use this paradigm for its maintainability and structural benefits.
MVC - Model-View-Controller Architecture	An architectural pattern which strictly separates components into the model (manages the data and logic), the view (output of the model), and the controller (handling input and passing it to the model or view).
Interface	A component of a system by which other entities (be it humans or other systems) may engage in an exchange of data with the system in question.
API - Application Programming Interface	A protocol or set of functions which serve as a method of communication to a software system. It is a type of interface, and the one by which our system will communicate with the banking institutions' databases.
DAO - Data access object	An object that provides an abstract interface to some type of database or other persistence mechanism.

3.3 Subsystem X

Detailed Design Diagram

UML class diagram depicting the internal structure of the subsystem, accompanied by a paragraph of text describing the rationale of this design.

*Note: The above is a description of what to provide. Need to edit into our own

Units Description

List each class in this subsystem and write a short description of its purpose, as well as notes or reminders useful for the programmers who will implement them. List all attributes and functions of the class.

*Note: The above is a description of what to provide. Need to edit into our own

4 Dynamic Design Scenarios

Describe some (at least two) important execution scenarios of the system using UML sequence diagrams. These scenarios must demonstrate how the various subsystems and units are interacting to achieve a system-level service. Units and subsystems depicted here must be compatible with the descriptions provided in section 2 and 3.

*Note: The above is a description of what to provide. Need to edit into our own

4.1 Dynamic Models of System Interface

We have chosen 3 major functionalities of the system (also known as use cases) in order to portray the interactions between the classes of the system. By using a sequence diagram, this will display the dynamics visually by showcasing the sequences of method calls when a particular use case begins functioning.

Use Case 1: Create User Account

The following scenario depicts the actions that occur when a user clicks the sign up button in order to create their account. Firstly, the SignUpController (which represents the controller part of the MVC architecture) is called to handle the sign up event. The SignUpController then calls the validatePassword() method in order to check if the password chosen fits the required format. Next, the signupcontroller sends a message to the User class in order to start its builder method. The builder method is filled out with the info needed to sign up. The user class then sends a message to the IUserService class in order to finally create a user within the database using the information entered. Finally, the IUserService class calls the MyMoneyApplication's displayLogin() method in order to show the user the success of creating an account (The View)

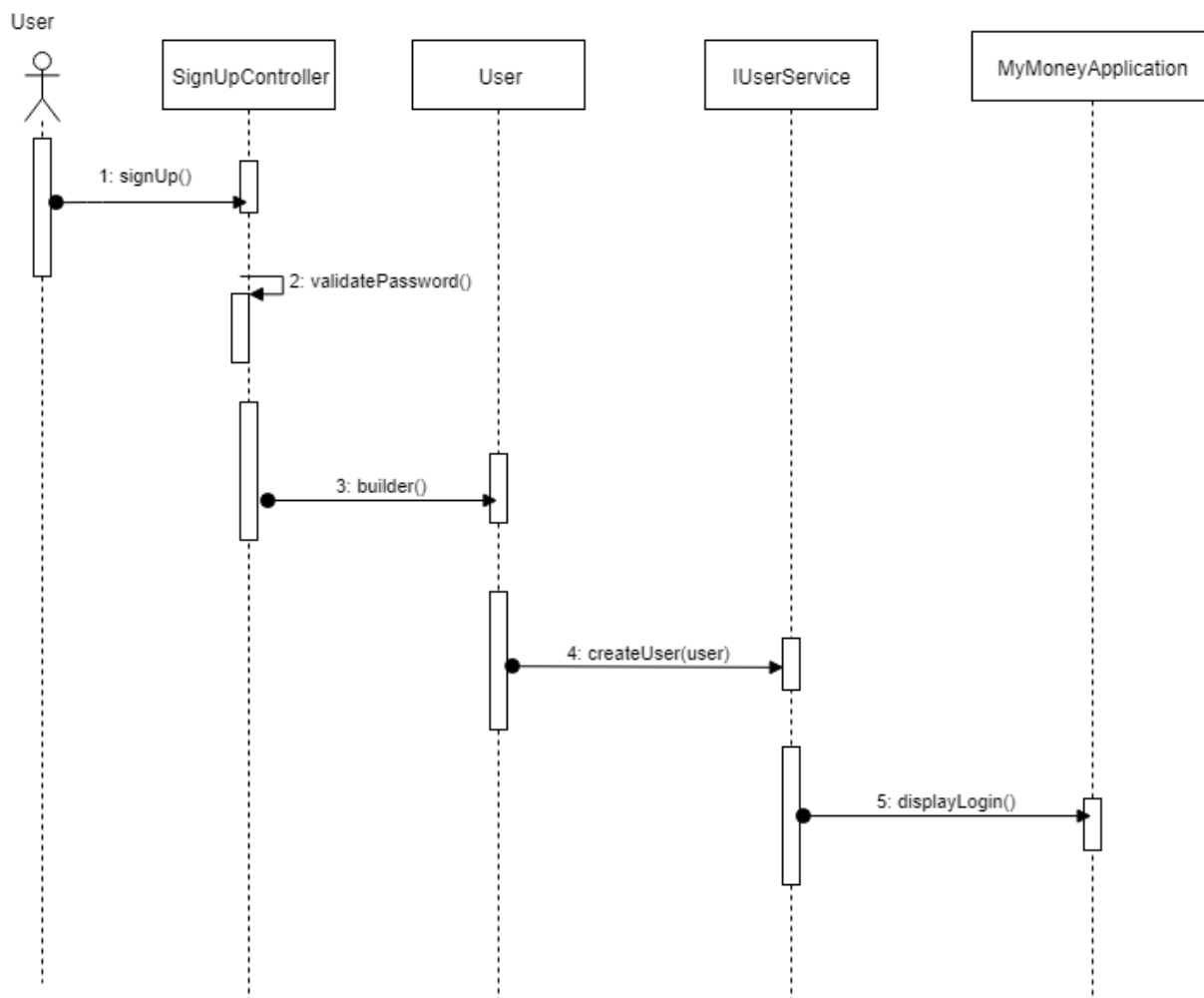


Figure 2: Use case 1 Sequence Diagram

Use Case 5: View Transactions for Specific Bank Account

The following scenario describes the actions occur when user click the button;view transactions; for a specific bank account.

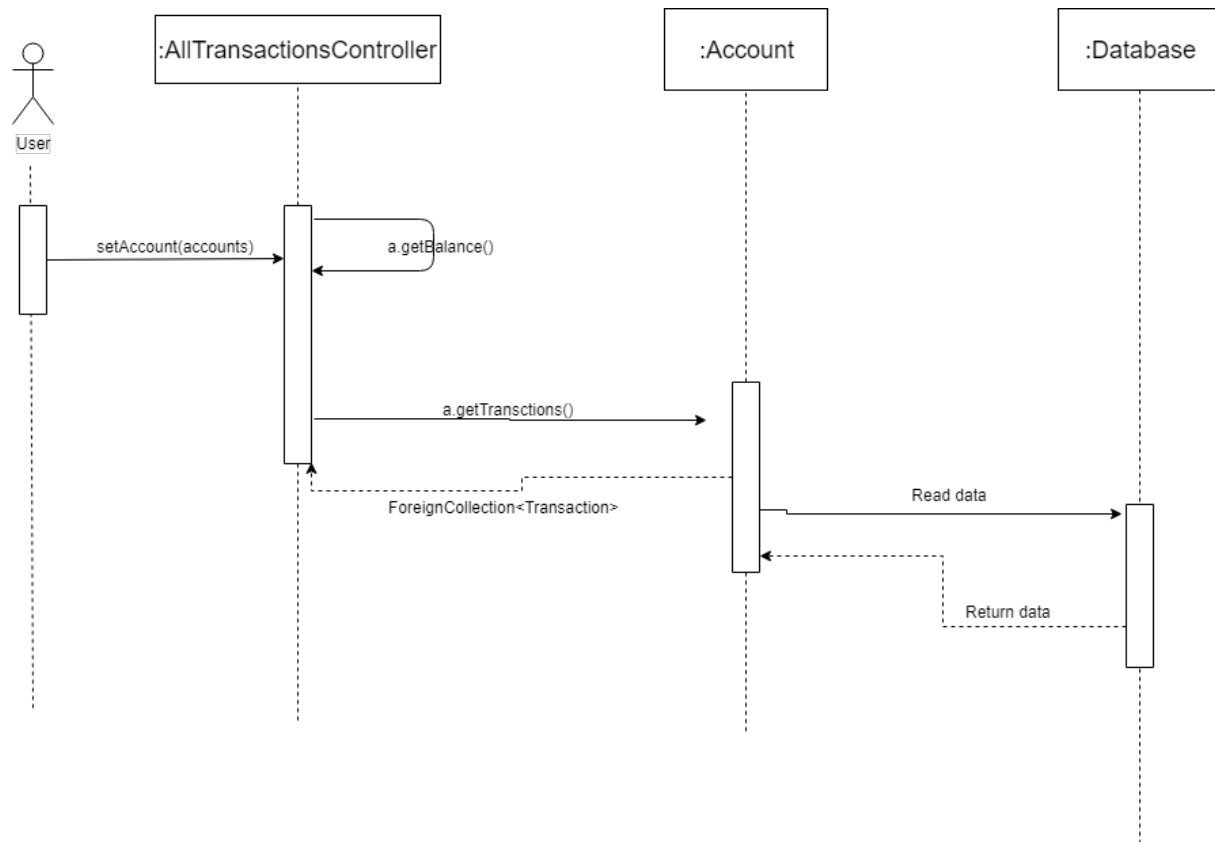


Figure 3: Use case 5 Sequence Diagram

Use Case 6: View All Transactions from all Bank Accounts

The following scenario describes the actions occur when user click the button view all transactions, for viewing all transactions from all bank accounts.

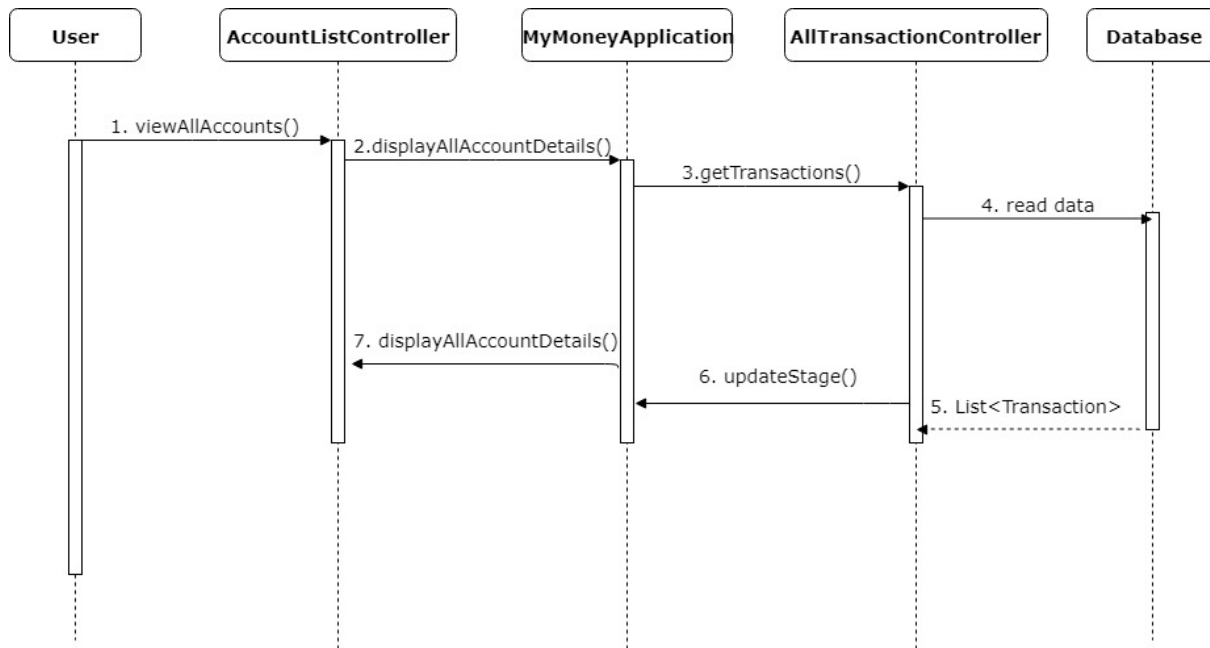


Figure 4: Use case 6 Sequence Diagram

5 Reference

- User information: As our user and use-cases was based on feedback provided by our developers, our references lie mainly within our own team.
- Craig Larman - Applying UML and Patterns
- Greg Butler's course COMP 354 content
- [MIT Curricular Information System Software Requirements Document](#)
- [Carnegie Mellon Business Goals](#)
- [Use-Case: Oracle](#)
- [Google Dagger Github](#)