

## CPT373 / COSC2363 Assignment 2 Specification

<b>Deadline</b>	<b>Sunday, 14/08/2022 11:59pm AEST</b>
<b>Total marks</b>	<b>45 marks</b>
<b>% Allocated to this assignment</b>	<b>45%</b>
<b>Assignment mode</b>	<b>In a group of 2 OR individually</b>
<b>To be submitted via</b>	<b>Canvas</b>

### 2.1 IMPORTANT

- a. You **MUST** submit your code via Canvas. See section 2.9 *Submission Procedure* for more information.
- b. You will be marked on the use of GitHub and development process.
  - You must join the **rmit-wdt-sp2-2022** GitHub Organisation using a GitHub account that has been registered with your RMIT student email address.
  - Using a personal GitHub repository and not being a part of the **rmit-wdt-sp2-2022** GitHub Organisation **will lead to ZERO for the whole assignment.**
  - The GitHub repository for this assignment must be **private** and named as: **<student-id>-a2** for example **s3123456-a2**  
  
If working in a group include **both** student IDs in the repository name as: **<student-id>-<student-id>-a2** for example **s3123456-s3654321-a2**
  - Include the URL of your GitHub repository in the readme file. The URLs for the repositories shown in the examples above would be:  
  
<https://github.com/rmit-wdt-sp2-2022/s3123456-a2>  
<https://github.com/rmit-wdt-sp2-2022/s3123456-s3654321-a2>
- c. You **MUST** make a commit at the beginning of the assignment period. You must also commit on a regular basis and make use of branches. Branches must represent the implementation of specific features.
- d. All data must be saved to an **Azure Microsoft SQL Server** database. Email Matthew at [matthew.bolger2@rmit.edu.au](mailto:matthew.bolger2@rmit.edu.au) if you have not received an invite to join the GitHub organisation or have not received your SQL Server credentials. **Using any other database or files to save data will lead to ZERO for the whole assignment.**
- e. In an advanced elective like this we place importance on code quality and the development process.
- f. Do **NOT MAKE ASSUMPTIONS** regarding the specifications. When in doubt post a question on Canvas or email us.
- g. The marks will be weighted according to your contribution within the group.  
**Please submit the contribution form.**

## 2.2 PLAGIARISM

All assignments will be checked with plagiarism-detection software; any student found to have plagiarised would be subject to disciplinary action. Plagiarism includes:

- **CONTRACT CHEATING:** Paying someone to do your work.
- Posting assignment questions (in full or partial) on external forums, reddit, etc...
- Submitting work that is not your own or submitting text that is not your own.
- Copying work from / of previous / current semester students.
- Allowing others to copy your work via email, printouts, social media, etc...
- Sending or passing your work to your friends.
- Posting assignment questions on technical forums to get them solved.

A disciplinary action can lead to:

- A meeting with the disciplinary committee.
- A score of zero for the assignment.
- A permanent record of copying in your personal university records and / or
- Expulsion from the university, in some severe cases.

All plagiarism will be penalised. There are no exceptions and no excuses. You have been warned. For more details, please read RMIT's page on Academic Integrity at:

<https://www.rmit.edu.au/students/my-course/assessment-results/academic-integrity>

## 2.3 Scope

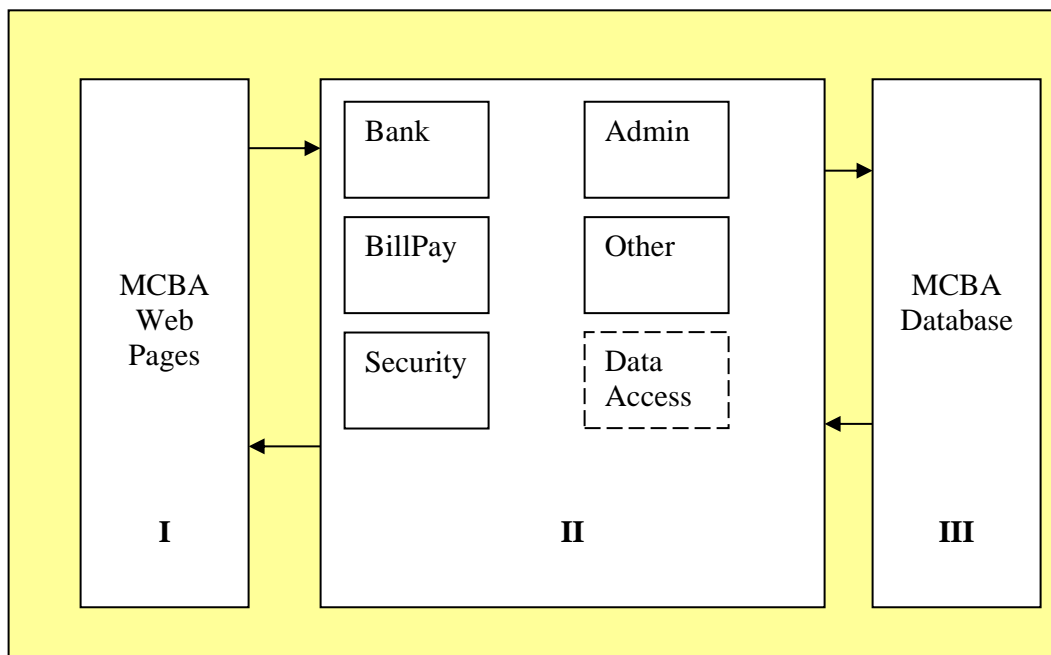
The aim of this assignment is to develop an ASP.NET Core MVC Website for Internet Banking with **.NET 6.0** written in **C#** using **Visual Studio 2022** with an **Azure Microsoft SQL Server** database. Database access is to be implemented with **EF Core** as the ORM with minimal SQL statements, ideally no SQL statements should be present.

## 2.4 Overview

MCBA (Most Common Bank of Australia) was impressed by the console application and wants you to design their Internet Banking website. It is a simulated banking application. When complete, the system will be able to:

- Check account balance and transaction history.
- Simulate transactions such as deposits and withdrawals.
- Transfer money between accounts.
- Schedule payments.
- Modify a personal profile.
- Perform administrative tasks.

The following diagram depicts the logical architecture:



The architecture is a three-tiered distributed design.

**Layer I:** User Interface (Presentation Layer): Will contain the VIEWS (web pages).

*Lorem Ipsum is **not** allowed.* Please add meaningful content and images to the pages.

**Layer II:** Business Logic / Object Layer: All the functionalities will be present in this layer. You can use CONTROLLERS or Business Objects for this purpose.

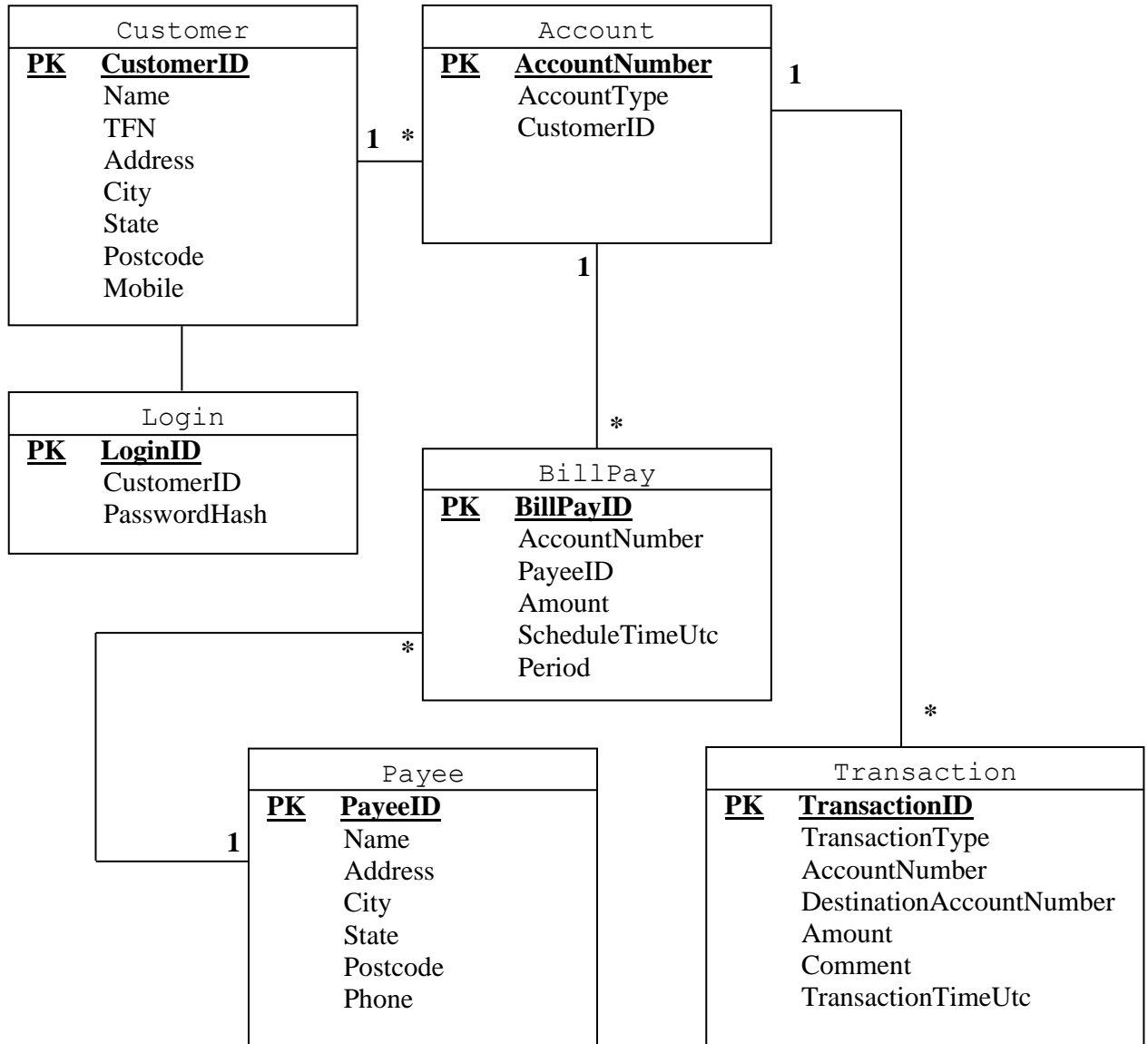
Business Objects are separate files where pieces of usable functionality are written. Instead of writing all application logic in controllers, you can call methods defined in business objects.

**Layer III:** Database / Data: This comprises of your Azure Microsoft SQL Server database.

For database operations you **MUST** only use **Entity Framework Core** (EF Core). If your query cannot be expressed with EF Core syntax you may use embedded SQL, however you must justify the use of any SQL in the readme file. Do not resort to using SQL unless necessary.

## 2.5 Database

Below is a more detailed version of the database provided to you for Assignment 1. This design is only a suggestion, you can modify the database structure, for example changing or adding constraints, default values, tables, fields, etc... however an unnormalized database with duplicated data / not well-defined entities / or too few tables will attract HEAVY penalty.



**IMPORTANT NOTE:** All format constraints must be enforced with server-side validation.

*Brief description of these tables follows →*

## Customer Table

Field Name	Data Type	Length	Description	Format	Allow Null
CustomerID	int		A unique ID for each customer	Must be 4 digits	NOT NULL
Name	nvarchar	50			NOT NULL
TFN	nvarchar	11	Tax File Number, this is just for identification purposes, <i>no tax implications</i>	Must be of the format: XXX XXX XXX	
Address	nvarchar	50			
City	nvarchar	40			
State	nvarchar	3		Must be a 2 or 3 lettered Australian state	
Postcode	nvarchar	4		Must be 4 digits	
Mobile	nvarchar	12		Must be of the format: 04XX XXX XXX	

## Login Table

Field Name	Data Type	Length	Description	Format	Allow Null
LoginID	nchar	8		Must be 8 digits	NOT NULL
CustomerID	int		FK to Customer		NOT NULL
PasswordHash	nchar	64		Must be stored in salted and hashed format	NOT NULL

## Account Table

Field Name	Data Type	Length	Description	Format	Allow Null
AccountNumber	int		A unique ID for each account, also the customer's account number	Must be 4 digits	NOT NULL
AccountType	char		The type of account such as checking or savings account	'C' or 'S'	NOT NULL
CustomerID	int		FK to Customer		NOT NULL

## Transaction Table

Field Name	Data Type	Length	Description	Format	Allow Null
TransactionID	int		Auto-generated unique ID for each transaction		NOT NULL
TransactionType	char		Type of transaction	Refer to Transaction Types below	NOT NULL
AccountNumber	int		Source account, FK to Account		NOT NULL
DestinationAccountNumber	int		Target account, used for transfers, FK to Account		
Amount	money		Amount to credit or debit for the transaction	Must be a positive value	NOT NULL
Comment	nvarchar	30	Comment for the transaction		
TransactionTimeUtc	datetime2		Date and time when the transaction occurred		NOT NULL

## BillPay Table

Field Name	Data Type	Length	Description	Format	Allow Null
BillPayID	int		Auto-generated unique ID		NOT NULL
AccountNumber	int		Account number to withdraw funds from, FK to Account		NOT NULL
PayeeID	int		FK to Payee		NOT NULL
Amount	money		Amount of funds to be withdrawn from the account	Must be a positive value	NOT NULL
ScheduleTimeUtc	datetime2		Next scheduled date and time for transaction to occur		NOT NULL
Period	char		How often scheduled payment will occur	One-off 'O' OR Monthly 'M'	NOT NULL

## Payee Table

Field Name	Data Type	Length	Description	Format	Allow Null
PayeeID	int		Auto-generated unique ID		NOT NULL
Name	nvarchar	50			NOT NULL
Address	nvarchar	50	Payee's address		NOT NULL
City	nvarchar	40			NOT NULL
State	nvarchar	3		Must be a 2 or 3 lettered Australian state	NOT NULL
Postcode	nvarchar	4		Must be 4 digits	NOT NULL
Phone	nvarchar	14		Must be of the format: (0x) xxxx xxxx	NOT NULL



## Transaction Types

D	=	Credit	Deposit
W	=	Debit	Withdraw
T	=	Debit	Transfer for source account
T	=	Credit	Transfer for target account
S	=	Debit	Service Charge
B	=	Debit	BillPay

**Customer** table is used to store all the customer information. Distinguishes between bank customers and payees by creating a separate Payee table. Only the bank's own customers will be held in the Customer table.

**Login** table is used to store customer's username and salted + hashed password allowing customers to login to the website.

**Account** table is used to store all the information about the different accounts that a customer may have.

**Transactions** table is used to track all credit and debit items. Each credit adds the amount to the balance and each debit subtracts the amount from the balance.

**BillPay** table is used to schedule automatic payments to a third party such as a telephone bill, electricity bill, or a home mortgage company. You can schedule these payments to be made at a specific date and time as a one-off or monthly payment.

## 2.6 Business Rules

Refer to page 4 within the Assignment 1 specification for the business rules.

## 2.7 Tasks and Marks Allocation

Pass [PA]

Part 1: Internet Banking - Customer Website [23 marks]

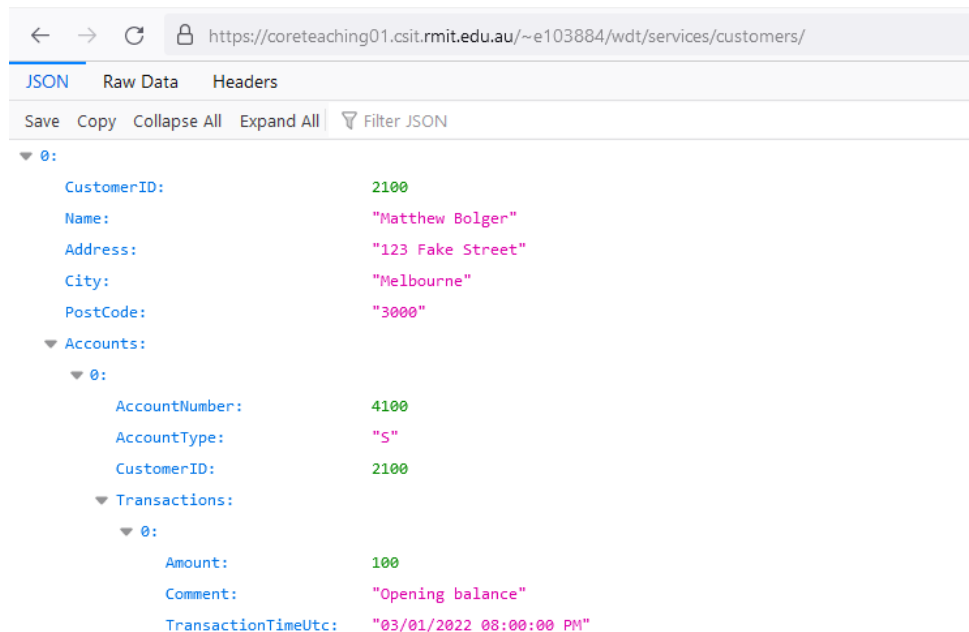
a) [3 marks] Using EF Core **Code First** approach create suitable Models and DbContext to produce the database tables. The migrations needed to generate the database schema must also be included.

b) [2 marks] Implement the pre-loading of data by making a **REST** call using **HTTP GET** to the provided customers web-service using the **JSON** returned to insert data into the database as appropriate. Generics should be used for the **JSON** deserialisation. The **REST** call should be implemented using the **System.Net.Http.HttpClient** class provided by **.NET**.

**NOTE:** Contacting the web-service should **only** occur if the system has no customer data, meaning if there are any customers present in the database then the web-service **should not** be contacted.

**Customers web-service:**

<https://coreteaching01.csit.rmit.edu.au/~e103884/wdt/services/customers/>



```
{
  "0": {
    "CustomerID": 2100,
    "Name": "Matthew Bolger",
    "Address": "123 Fake Street",
    "City": "Melbourne",
    "PostCode": "3000",
    "Accounts": {
      "0": {
        "AccountNumber": 4100,
        "AccountType": "S",
        "CustomerID": 2100,
        "Transactions": {
          "0": {
            "Amount": 100,
            "Comment": "Opening balance",
            "TransactionTimeUtc": "03/01/2022 08:00:00 PM"
          }
        }
      }
    }
  }
}
```

When inserting customers and creating their accounts all the transactions from the web-service are to be transaction type D (Deposit). Additionally, the account's balance is to be set to the sum of all the account's associated transactions.

Refer to the **Dealing\_with\_Passwords.pdf** file for information on how to handle passwords and using the PasswordHash field.

c) [6 marks] Create a login page (start page), which should have some introductory information about the bank and a form to login. Once logged in provide a logout link. **You may use the customised logic for login and logout as discussed during Week 6.**

Once logged in the user should be shown see the balance of their accounts. Each page must have a navigation bar, which should provide links to:

Deposit || Withdraw || Transfer || My Statements || My Profile

The UI layout and design is up to you. For example, you could have a separate page for Deposit, Withdraw and Transfer or reuse the same page for all these actions, etc...

When performing a deposit, withdraw or transfer the user should be presented with a **confirmation page** displaying the details of the transaction with an option to either **confirm** or **cancel** the transaction.

Ensure **login** and **logout** are fully functional.

d) [3 marks] My Statements page allows the user to see the **current balance for a specified account** and list its transaction history. The transactions should be ordered by transaction time with the most recent transaction coming first and must be displayed in a paged manner, showing only 4 transactions at a time and allows the user to move to the next and previous pages as appropriate. The transaction time should display both the **date and time** information with the time portion including the hours and minutes.

e) [2 marks] My Profile page displays the user's customer information Name, TFN, Address, City, State, Postcode and Mobile in a read-only (non-editing) view. The user should also be able to proceed to modify this information.

Additionally, the user should be able to change their password. Any changes made must be written back to the database.

f) [3 marks] All format constraints (outlined in the table descriptions) must be enforced with server-side validation. The format constraints can *optionally* also be enforced with client-side validation, however server-side validation is mandatory.

The format constraints should be implemented using appropriate data-annotations within the Models when possible.

**g) [4 marks]** Effective use of version control and Trello.

Version control has the following guidelines:

- Effective use of a GitHub repository that demonstrates your development practises.
- Regular commits have been made throughout the development period.
- Commit messages should be meaningful, the messages can be short provided they are concise.
- Branches are used to represent each feature you are implementing; you may break a feature into additional sub-components.
- Branches should be given meaningful names.
- A minimum of 6 branches should be used in addition to the main branch.
- All branches should be merged into the main branch when completed.
- All branch commits must be checked and verified by your group partner prior to merging back into the main branch. If working in a group create and use GitHub Pull Requests to perform the merging of a branch into the main branch.

Trello has the following guidelines:

- Document your development and thought process over time as you build features of the software.
- The board should show the journey from the start of the project through to the end.
- Include milestones and features on the board. You may also include design, proof-of-concept, testing and bug-fixing tasks on the board if desired.
- You are allowed to use online templates.
- The boards should show sufficient details to the marker.
- Embed a minimum of 8 screenshots in your GitHub repository in a directory called Trello. The screenshots should be taken over time, for example adding a new screenshot every few days reflecting the project's progress. The file name should include the date in **year-month-day** format, for example **2022-06-30.png**. If multiple screenshots are taken on the same day, then include a timestamp using 24-hour format, for example **2022-06-30-11-30.png** and **2022-06-30-16-30.png**.

Credit [CR]

Part 2: BillPay, Payee and Scheduled Payments [5 marks]

**h) [5 marks]** Add another link to the navigation bar called BillPay. This page shows the user their currently scheduled bills to be paid and includes a link to create a new entry. Additionally, there should be an option to modify a scheduled bill, including the ability to cancel a payment.

Once a bill is scheduled the payment should be automatically executed at the date and time scheduled by the user. The processing of scheduled payments must be persistent, meaning if the web server is stopped or restarted then the pending payments are not forgotten and are still processed. Any payments due to be paid should be processed immediately upon the web server starting. The bill payment should fail if the account has an insufficient balance. If failure occurs the error should be conveyed to the user on the BillPay page with options to deal with the error, for example cancel, reschedule, or try again, etc... how error handling is implemented, including storage and presentation is up to your own discretion.

Distinction [DI]

Part 3: Internet Banking - Admin Web API and Admin Portal Website [9 marks]

i) [4 marks] Create an **ASP.NET Core Web API** exposing endpoints to perform all back-end and database operations required for the Admin Portal Website. The methods of this API should be well-documented by writing comments to explain the various endpoints of the API. The Web API is to be implemented using the **Repository** design pattern.

To simplify the Web API there is **no requirement** to include authentication or authorisation when calling the API.

**NOTE:** The Admin Web API and Admin Portal Website can be contained within the same project. Although the **recommendation** is to use separate projects for each site.

j) [5 marks] Create an Admin Portal Website as described below. This site **must be independent** from the Customer Website and thus **should not** be included within the Customer Website project.

Admin username and password **do not** need to be stored into the database at this stage. After a successful login (**username = admin, password = admin**), admin should be able to:

1. View transaction history for an account. Admin should be able to generate a table displaying the transaction history for an account within a specified start date and end date period, if no start date and / or end date is entered then no filter should be applied for that field.
2. Modify a customer's profile details Name, TFN, Address, City, State, Postcode and Mobile.
3. Lock and unlock a customer's login. Once locked the customer should not be able to login to the Customer Website until unlocked by an admin.
4. Block and unblock scheduled payments. Blocked bills should be displayed as "Blocked" within the BillPay list on the Customer Website. The customer cannot unblock the payment, although the customer can delete the bill if desired. When a scheduled payment is blocked it should not run until unblocked by an admin.

All the above features **MUST** use the Admin Web API for accessing the database. The pages should contain **minimal** logic / code beyond calling the API and processing the response.

**NOTE:** The following section of the assignment requires self-effort. You will need to take initiative and research to implement the following requirements.

High Distinction [HD]

Part 4: Backend and Unit Testing [8 marks]

**k) [3 marks]** Backend logic and implementation should be decoupled and separated from the controllers. This includes business rule validation / enforcement, performing complex database operations such as deposit, withdraw, transfer, scheduled payments, etc...

The controllers will utilise the backend endpoints resulting in light-weight controllers.

**NOTE:** Validation performed with data-annotations on models / view-models is allowed.

**l) [5 marks]** Unit test all backend endpoints / classes, structs, records / methods, controllers, and actions. Methods and types should have various use-cases / edge-cases covered if they take in parameters or have context / state involved. The BillPay scheduled payments, both the one-off and monthly, should also be unit tested.

The unit tests must support being executed via the **dotnet test** command.

Unit testing should adhere to the AAA (Arrange, Act, Assert) pattern convention:

- The **Arrange** section of a unit test method initialises objects and sets the value of the data that is passed to the method under test.
- The **Act** section invokes the method under test with the arranged parameters.
- The **Assert** section verifies that the action of the method under test behaves as expected.

**NOTE:** Please read this online resource:

<https://docs.microsoft.com/en-au/visualstudio/test/unit-test-basics?view=vs-2022>

**NOTE:** Multiple cycles of arrange-act-assert can be present within a single unit test.

The unit test naming convention is entirely up to your own discretion.

Code should not be duplicated between tests when possible. For example, if multiple tests are implemented as **Facts** that have the same duplicated code only with different test data, then these **Facts** should be refactored into a **Theory** with input data supplied via **InlineData**.

The entire project should be covered by unit testing. Significant coverage is required to achieve full marks in this section.

## 2.8 Coding Standards

- Read the C# coding standard from the following website:  
<https://docs.microsoft.com/en-au/dotnet/csharp/fundamentals/coding-style/coding-conventions>
- Remember that there are too many to be followed, implementing 6 to 10 of the standards will be a job well done.
- Do not force yourself to implement every object-oriented feature that you have learnt, use the features wisely and if needed.

## 2.9 Submission Procedure

Each submission must include a readme file containing your full name, student ID, the URL to your GitHub repository, document your application, and any other relevant information. If you are working in a group, then please mention the details of your partner. The readme file should ideally be named **Readme.md** and be contained within the root of your repository.

Upload the solution / project, readme, and contribution form (if working in a group) as a single zip file to Canvas.

**NOTE:** You can include the contribution form within the repository if desired.

## 2.10 Late Submissions and Extension-Related Information

A penalty of 10% per day of the total marks for the assignment will apply for each day a submission is late, including both weekdays and the weekend. After 5 days, you will receive zero marks for the assignment. Email the course instructor [matthew.bolger2@rmit.edu.au](mailto:matthew.bolger2@rmit.edu.au) for extension related queries.