CST2550

Software Engineering Management and Development

Coursework 1 – GROUP COURSEWORK.

Autumn/Winter term

2024/2025

Date of Submission: 16/04/2025

Student Name	Student ID Number	Group Contribution
ETIENNETTE MICHAËL	**M00961339**	**100%**
RAMASAMY KIMBERLY	**M00957365**	**100%**
SOONDRUM ANUSHREE	**M00906392**	**60%**
MOHAMED UTTANWALLA	**M00972900**	**60%**
MOHESH LAKSHVEER	**M00998155**	**50%**

Lab Tutor: Aisha Idoo.

Table of Contents

Table of Table

Table of Figures

Timeless Tapes LTD

A video renting system you have never seen before!

Introduction.

A "video rental system" is a system that allows customers to rent movies or videos from a store, typically by browsing a catalogue, selecting titles, and paying a fee to borrow them for a set period of time. Then the movies are returned to the store. Essentially, it's a way to access movies without buying them outright, similar to how libraries lend books. This report will contain a brief introduction of our company, **Timeless Tapes LTD**, what our video renting application will contain, how we managed the project and the task breakdown. Next you will find an analysis on the data structure we selected and the algorithms we used. Then, in the conclusion, you will find a summary of the work done, challenges, solutions and how we would tackle similar projects in the future.

Our company, **Timeless Tapes LTD**, have decided to created our very own video renting system application where each customer will be able to rent movies. Us, the admins, will be responsible for the renting system, where each renting period is tracked and if movies are returned accordingly. Below you will find the different components of our application.

Key aspects of a video rental system:

Inventory management.	The store needs to keep track of all available movies/videos in their collection.
Customer accounts.	Customers need to register and maintain their rental history with the store.
Search functionality.	Customers can search for movies by title, genre, or other criteria.
Delete functionality.	Admin can delete movies from database if no more on rent/sale.
Sort functionality.	
Rental process.	Customers select movies, pay the rental fee, and take the movies home.
Return process.	Customers return rented movies to the store and may be charged late fees if returned past the due date.

Table 1

Once the key features identified, we proceeded with the project management part of the coursework.

Design.

Components.

When starting the project, we needed to understand clearly what is a video renting system. We all brainstormed the different components that we will include in the database and the app's options. The basic structure of the application according to our pseudocode (see Appendix for full pseudo code layout).

Functionality	**Use**
User Authentication	Login, Register, Logout.
Customer screen	View history, Rent videos, Return videos, Calculate fines.

Video Catalogue	Browse catalogue, search movie, check availability, update ratings.
Transactions	Process payment, generate receipts, calculate fines.
Admin screen	Add videos, remove videos, update movie details, view all rentals, view customer history.
Database Handler	Open & close connection with database, read data (query), write data (query).
General Menu	Show menu, handle on-screen actions.

Table 2

Data Structure.

We needed to choose between, .NET Core Web API, WinForms Desktop or a console app for the base of the app.

.NET Core Web API allows a microservices-based structure which improves scalability. It also allows independent development and scaling of features like user authentication, customer management and videos catalogue services (Mashaly et al., 2022). It also supports cloud-based development which aligns well with today's enterprise solutions (Bakshi, 2017). In contrast, WinForms has more of a monolithic structure where all features are regrouped into one-single entity. It may suit smaller, simpler applications, but restricts scalability and makes maintenance complex when changes to one component impact the entire system (Bakshi, 2017). While Console apps are faster to build, they are limited in terms of structure. They lack built-in support for any sort of GUI which makes it unsuitable when making complex projects like our video renting system (Mashaly et al., 2022).

Criteria	**.NET Core Web API**	**WinForms Desktop**	**Console App**
Scalable Architecture	✓	✖	✖
Cross-Platform Support	✓	✖	✓
Front-End Integration (e.g. React, Angular)	✓	✖	✖
Cloud Deployment Compatibility	✓	✖	✖
High-Concurrency Handling	✓	✖	✖
GUI Support	✓	✓	✖
Ease of Maintenance (Modular Updates)	✓	✖	✖
DevOps & Automation Support	✓	✖	✖
Rapid Prototyping	✖	✓	✓
Use in Modern Applications	✓	✖	✖

Table 3

Key: ✖ = does not support, ✓ = supports

.NET Core Web API stands out as the best option for a modern, scalable and maintainable video rental system. It allows a service-oriented structure, support cross-platform deployment, as well as proved a seamless front-end integration. It also adheres to best practices in DevOps and system design.

Database design.

Next step for the pseudo code was the creation of the database. During our team meetings, we identified the components of the tables that will be included in the database (see image below).

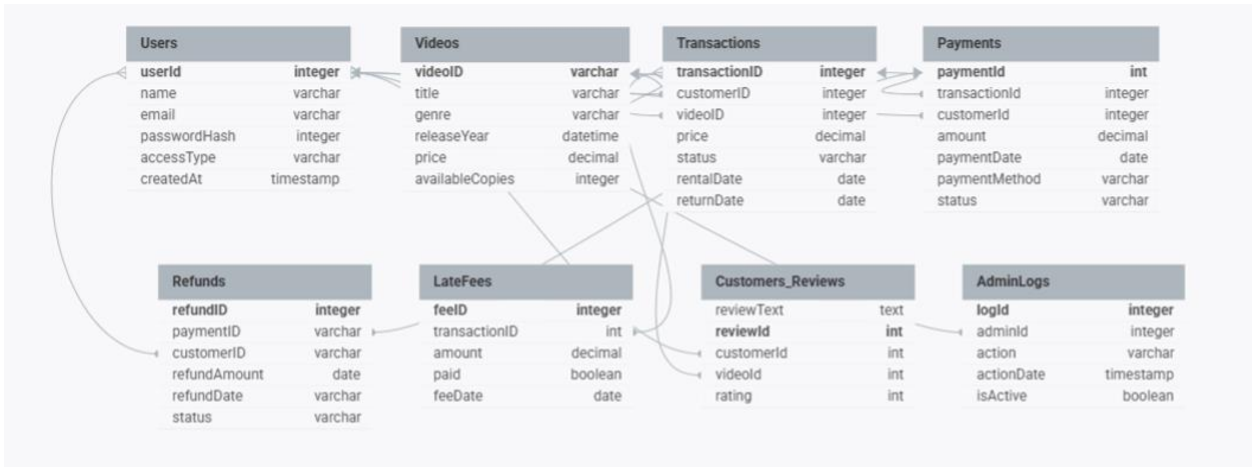


Figure 1

Here you have the tables that are present in the database. We needed to store information like “userId”, “accessType” so that each user will have a unique identifier and specific access type depending on whether they are client or admin. If client, the user will be able to see the catalogue, rent movies, see movies being rent by them and any late fee that applies if they return the movie past the deadline. If admin, the admin will be able to edit the database by adding and removing movies from the catalogue and database. Other information like “price” and “status” are important since they show the user if they can rent the movie (status) and the rental price, along with the “rentalDate” and “returnDate”.

Choice of algorithm.

Once the database design was done, we proceeded with the choice of the correct algorithm for our data structure. We could use either Hash tables or Linked Lists. To be able to choose the one that fits best the complexity of our video renting system app, we needed to see which one would provide the best result overall.

Criteria	**Hash Table**	**Linked List**
Search Efficiency	✓ O(1) average-case access due to direct indexing via hashing. Highly efficient for frequent lookups.	✖ O(n) search time as each node must be traversed sequentially (Costea et al., 2020).
Insertion/Deletion Speed	✓ Constant time (O(1)) in ideal conditions; suitable for dynamic datasets like user or video records (Jünger et al., 2020).	✓ Also supports O(1) insertion/deletion at head, but locating nodes takes O(n) time.
Memory Usage	✖ Higher memory consumption due to preallocated space and possible collisions (Jünger et al., 2020).	✓ More memory-efficient for smaller or sequential data operations (Costea et al., 2020).
Ordering of Data	✖ Does not maintain element order.	✓ Maintains sequence, useful for ordered traversal or playlist-style features.
Use Case Suitability	✓	✖

	Best for frequent, fast retrieval of user/video IDs, rental status, or login credentials.	More suitable for educational demos or scenarios where order matters more than speed.
Parallelization Capability	✓ Can be GPU-optimized for high-speed processing in parallel systems (Jünger et al., 2020).	✖ Less suitable for parallelism due to sequential node structure.

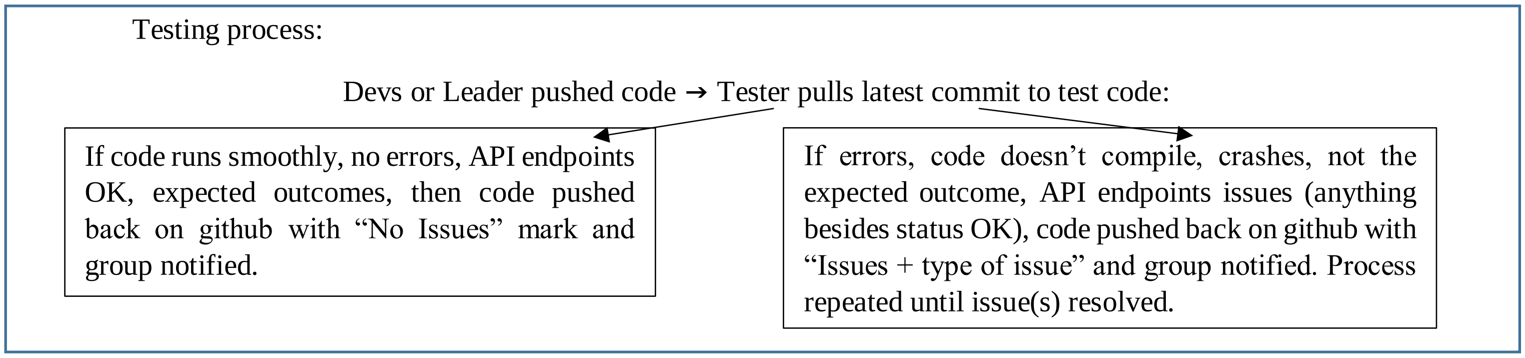
Table 4

Key: ✖ = does not support, ✓ = supports

Our app demands fast access to user data, regular checking of video availability and processing transactions real-time. Plus, we needed to make sure that time complexity was taken into account. In light of the above comparisons, we decided to go for Hash Tables implementation. This algorithm will prove consistent performance for our application since it requires scalable access across many users or products.

<u>Testing.</u>

Testing was carried out in an orderly manner. Whenever the developers or the leader pushed their codes on github, the tester was notified right away on the group-chat or during meetings. From there, the codes were pulled for testing. The tester would test API end-points (see appendix for full table) since our application uses API endpoints to send and retrieve data.



Testing process:

Devs or Leader pushed code → Tester pulls latest commit to test code:

If code runs smoothly, no errors, API endpoints OK, expected outcomes, then code pushed back on github with “No Issues” mark and group notified.

If errors, code doesn’t compile, crashes, not the expected outcome, API endpoints issues (anything besides status OK), code pushed back on github with “Issues + type of issue” and group notified. Process repeated until issue(s) resolved.

Figure 2

<u>Project Management.</u>

Throughout the course of our project, we needed to make sure that task flow was smooth, requirements respected and everything was done within a reasonable period of time. Hence, during the group formation, each member was given a specific role. The leader, SCRUM Master, Michaël, delegated tasks and ensured everyone stayed on track. Kimberly was the secretary and was tasked to handle the meetings, recording the minutes and making the report. Anushree and Mohamed were the developers who, alongside Michaël, coded the application. Lakshveer was the tester and handled every required tests.

Daily stand-up meetings were done during the whole month of March. Each member provided an update on the tasks they were doing and what they were planning to do next according to the task allocation schedule. Two separate meeting links were created according to our university timetable to find the best time to meet. The Monday-Wednesday link and the Thursday-Friday link. The times of the meetings varied depending on the availability of the members but reminders were sent for everyone to be present. All meetings were done online using Google Meet.

The first meetings were mostly for learning more about the coursework and our respective roles. We brainstormed and did in-depth research on video renting systems to draw inspiration for the application to be developed. The team got familiar with their roles and what was expected from them. Before proceeding, we made sure that everyone agreed to their assigned roles. After understanding what the coursework was about, we drafted a first pseudocode, a database design - which was revised later for approval - researched and chose the type of data structure and algorithms we would use to build our video renting application. Once everything was finalised, the developers started working on the codes, with the help of the leader and tester where necessary.

The tasks were broken down into six main categories. Each category had sub-tasks which needed to be completed and reviewed before working on the next category. For the tasks allocation, the leader would keep track of the current tasks being done and advise them on what to do next. All team members communicated via a group-chat were members provided updates. The developers and the leader regularly made commits in the repos. Then, the tester would pull the code from github, test it, commit again with flagged issue or nothing if the code ran and the expected outcomes were obtained. If any issues were encountered, we tackled them during the meetings by doing brainstorming combined with research. Each member had a small personal timetable with their respective tasks to stay on track and be able to meet all the respective deadlines. During each meeting, minutes were recorded until the minimum required of 30 was met.

Concerning the report, research of material was done on Google Scholar or IEEE Xplore. Everyone contributed to obtained the necessary material, then the secretary did the report according to the guidelines. The full meeting minutes sheet was added to the repos (see Appendix to see how minutes were recorded).

Conclusion.

Summary of work done.

1. Group formation. → Getting assigned project. → Group name.
2. First meeting. → Roles assigned. → Understanding who does what.
3. Brainstorming about the project → Identifying what we will include. → First draft of ideas and research.
4. Choosing data structure and algorithm → Pseudocode draft. → Database design. → Started coding.
5. Obtaining data source. → Data cleaning → Adding values to database. → Database connected set.
6. Completing codes. → Testing of codes → Tackling errors if any. (Category 6 was repeated until all code writing was done).
7. Making video demonstration. → Adding all deliverables to the repos. → Submitting project.

Figure 3

Limitations and critical reflection.

The coursework did not come without drawbacks. Oftentimes, communication wasn't seamless which resulted in tasks being delayed or dragged over multiple days. With little knowledge and expertise concerning C#, building a project this complex was quite challenging. Sometimes, resources were outdated or too complicated for us to understand in the short amount of time. Therefore, we decided to create a simpler application, while ensuring that guidelines were met. With code errors, frustration started to arise and the developers had to take breaks to cope. However, these breaks lasted longer than anticipated which slowed down the pace of work to be done. Not all group members used the same types of devices to work. This problem mainly concerned the developers and the leader since one of the developers used Mac and had trouble pushing the code to github. Towards the end of the project, the deadline was unexpectedly preponed which increased stress level amongst the members. This event was clearly unforeseen and broke the group dynamic. We had to review our initial schedule to be able to finish in time.

Nonetheless, we did not let these drawbacks affect us. Despite the challenges, we learned to overcome them as a team. Plus, these problems are not uncommon and tend to surface when working in groups. Communication remains the toughest one to solve but we figured that with better time management and by assessing our priorities, we can communicate better with each team members, and instead of communicating with one member specifically, we can use the group-chat. Someone who's more free can pitch in to help while the other is busy. When stuck on codes, it is important to take break and come back with a cleared mind to be able to work. However, we should not let the break last for too long because we tend to forget about the tasks we have to do. The cross-compatibility will always remain an issue but we need to learn to ask for help instead of trying to fix everything on our own. The preponed deadline forms part of reality and it is something that happens regularly when doing big projects. We truly think that we could have dealt with the situation better by using it as motivation to finish and adjust our time effectively instead of panicking.

To conclude, this C# project was a tough one but we have gained incredible insight on how to do a project using a more professional approach, tackle issues and manage our time to the best of our abilities.

Appendix

References

Mashaly, B., Selim, S., Yousef, A. H., & Fouad, K. M. (2022). Privacy by Design: A Microservices-Based Software Architecture Approach. *2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, Cairo, Egypt, pp. 357–364. doi: 10.1109/MIUCC55081.2022.9781685 [Retrieved from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9781685&isnumber=9781642]

Bakshi, K. (2017). Microservices-based software architecture and approaches. *2017 IEEE Aerospace Conference*, Big Sky, MT, USA, pp. 1–8. doi: 10.1109/AERO.2017.7943959 [Retrieved from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7943959&isnumber=7943554]

Jünger, D., Menon, H., & Besta, M. (2020). *WarpCore: A Library for fast Hash Tables on GPUs*. 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC), Pune, India, pp. 11-20. doi: 10.1109/HiPC50609.2020.00015. [Retrieved from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9406635]

Costea, F.-M., Chirila, C.-B., & Creţu, V.-I. (2020). *Auto-Generative Learning Objects for Learning Linked Lists Concepts*. 2020 International Symposium on Electronics and Telecommunications (ISETC), Timisoara, Romania, pp. 1-4. doi: 10.1109/ISETC50328.2020.9301136. [Retrieved from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9301136]

Data source obtained from Kaggle [https://www.kaggle.com/datasets/shivamb/netflix-shows]

Pseudo code.

```
Class User
  Attributes:
    UserID: Integer  // Unique identifier for the user
    Name: String     // User's full name
    Password: String  // User's login password
  Methods:
    Function Login(UserID, Password) Returns AccessType
      // Purpose: Authenticate a user and determine their access level.
      // Search: Searches the user database for a matching UserID.
      // Logic:
      //   1. Search user database for User with matching UserID.
      //   2. If User found:
      //     3. If User.Password matches Password:
      //       4. Return User's AccessType (Admin, Customer, Guest).
```

```
        //      5. Else:
        //          6. Return "Guest" (or an error code).
        //   7. Else:
        //      8. Return "Guest" (or an error code).
    End Function
    Function Register(Name, Password, AccessType) Returns Integer
        // Purpose: Create a new user account.
        // Logic:
        //   1. Generate a unique UserID.
        //   2. Create a new User object with Name, Password, AccessType, and UserID.
        //   3. Store the new User in the user database.
        //   4. If successful:
        //      5. Return the new UserID.
        //   6. Else:
        //      7. Return an error code.
    End Function
    Function Logout()
        // Purpose: Terminate the user's session.
        // Logic:
        //   1. Clear the user's session data (e.g., in memory or database).
    End Function
End Class
```

Class Admin

```
  Methods:
    Function AddVideo()
        // Purpose: Add a new video to the system.
        // Logic:
        //   1. Get video details (Title, Genre, Price, etc.).
        //   2. Create a new Video object with the details.
        //   3. Store the new Video in the video database.
    End Function
    Function RemoveVideo()
        // Purpose: Remove a video from the system.
        // Search: Searches the video database for a matching VideoID.
        // Logic:
        //   1. Get the VideoID to remove.
        //   2. Search video database for Video with matching VideoID.
        //   3. If Video found:
        //      4. Remove the Video from the video database.
        //   5. Else:
        //      6. Return an error message (Video not found).
    End Function
    Function CustomersTransactions() Returns List of Transactions
        // Purpose: Retrieve a list of all transactions.
        // Sort: Sorts the list of transactions based on a criterion (e.g., date).
        // Logic:
        //   1. Retrieve all transactions from the transaction database.
        //   2. Sort the list of transactions (e.g., by date).
        //   3. Return the sorted list of transactions.
    End Function
End Class
```

Class Customer

```
  Methods:
    Function RentVideo(VideoID) Returns String
        // Purpose: Allow a customer to rent a video.
        // Search: Searches the video database for a matching VideoID.
        // Logic:
        //   1. Search video database for Video with matching VideoID.
        //   2. If Video found and available:
        //      3. Create a new Transaction object.
        //      4. Store the Transaction in the transaction database.
        //      5. Mark the Video as unavailable.
        //      6. Return a success message.
        //   7. Else:
        //      8. Return an error message (Video not found or unavailable).
    End Function
    Function ReturnVideo(VideoID) Returns String
        // Purpose: Allow a customer to return a rented video.
        // Search: Searches the transaction database for a matching VideoID and CustomerID.
        // Logic:
        //   1. Search transaction database for Transaction with matching VideoID and CustomerID.
        //   2. If Transaction found:
        //      3. Mark the Video as available.
        //      4. Update the Transaction status.
        //      5. Return a success message.
        //   6. Else:
        //      7. Return an error message (Transaction not found).
    End Function
    Function MyTransactions() Returns List of Transactions
        // Purpose: Retrieve a list of transactions for the customer.
        // Search: Searches the transaction database for transactions with the customer's UserID.
        // Sort: Sorts the list of transactions based on a criterion (e.g., date).
        // Logic:
        //   1. Search transaction database for Transactions with matching CustomerID.
        //   2. Sort the list of transactions (e.g., by date).
        //   3. Return the sorted list of transactions.
    End Function
End Class
```

Class Video

```
  Attributes:
        VideoID: Integer       // Unique identifier for the video
        Title: String          // Title of the video
        Genre: String          // Genre of the video
        Price: Double          // Rental price of the video
        Availability: Boolean  // Indicates if the video is available for rent
End Class
```

Class Transactions

```
    Attributes:
        TransactionID: Integer  // Unique identifier for the transaction
        CustomerID: Integer    // UserID of the customer who rented the video
        VideoID: Integer       // VideoID of the rented video
        Price: Double          // Rental price of the video
        DaysRented: Integer    // Number of days the video is rented

    Methods:
        Function ProcessPayment(CustomerID, Amount) Returns String
            // Purpose: Process a payment for a transaction.
            // Logic:
            //   1. Validate the payment details.
            //   2. Deduct the Amount from the Customer's balance.
            //   3. Record the payment in the database.
            //   4. If successful:
            //      5. Return a success message.
            //   6. Else:
            //      7. Return an error message.
        End Function
        Function RefundPayment(TransactionID) Returns String
            // Purpose: Process a refund for a transaction.
            // Logic:
            //   1. Locate the Transaction in the database.
            //   2. Verify if the refund is eligible.
            //   3. Process the refund by updating records.
            //   4. If successful:
            //      5. Return a success message.
            //   6. Else:
            //      7. Return an error message.
        End Function
    End Class
```

Class DatabaseHandler

```
  Methods:
    Function Connect()
        // Purpose: Establish a connection to the database.
        // Logic:
        //   1. Connect to the database.
        //   2. Return the connection object.
    End Function
    Function ExecuteQuery(Query, Parameters) Returns Data
        // Purpose: Execute a query and return the result.
        // Logic:
        //   1. Execute the given Query with Parameters.
        //   2. Fetch the results.
        //   3. Return the results.
    End Function
    Function ExecuteNonQuery(Query, Parameters) Returns Boolean
        // Purpose: Execute a query that does not return data (e.g., Insert, Update, Delete).
        // Logic:
        //   1. Execute the given Query with Parameters.
        //   2. Return True if successful, otherwise False.
    End Function
End Class
```

Test Code Table.

EndPoint	Method	Input Being Tested	Output	Test Outcome
api/adminaction/add-video	POST	admin posting a video	200 OK: Video added	Passed
		not admin posting a video	403 Forbidden: Not logged in or not admin	Passed
/api/adminaction/remove-video	DELETE	Removes an existing video from the system.	200 OK: Video removed.	Passed
		Not admin removing a video	403 Forbidden: Not logged in or not admin.	Passed
/api/csvupload/upload	POST	CSV file containing multiple video records is being uploaded	200 OK:CSV uploaded and processed successfully.,count=1	Passed
		No file is not provided	400 Bad Request:Please upload a valid CSV file.	Passed
		wrong file is provided	400 Bad Request:Please upload a valid CSV file.	Passed
		CSV is empty	400 Bad Request:CSV file is empty or could not be parsed.	Passed
		CSV Columns heading do not match	400 Bad Request:CSV file is empty or could not be parsed.	Passed
/api/transactions/rent	POST	Rent a video for a customer	200 OK: Video rented successfully	Passed
		Renting video for non-existing customer	404 Not Found: Customer not found.	Passed
		Renting non existing video	404 Not Found:Video unavailable or already rented.	Passed
/api/transactions/return	POST	Return a video rented by a customer	200 OK: Video returned successfully	Passed

		non-existing customer	404 Not Found: "Customer not found	Passed
		The video wasn't rented	404 Not Found: Transaction not found.	Passed
		video does not exist	404 Not Found:Video not found or unavailable	Passed
/api/transactions/customer/1	GET	Get all transactions of customer 1	{ "TransactionId": 1, "TitleId": "1234", "CustomerId": 1, "RentalDate": "2024-04-10T00:00:00", "ReturnDate": "2024-05-15T00:00:00" }	Passed
/api/transactions/1/receipt	GET	Get the receipt of a specific transaction.	200 OK	Passed
/api/transactions/1/fines	GET	Retrieves the total fines for a customer 1	{ "customerId": 1, "totalFine": 5.99 }	Passed
/api/user/register	POST	Register a new customer	200 OK: "User registered successfully	Passed
		Using an already registerd email	400: "Email is already in use."	Passed
		Not putting name	400:"Name is required."	Passed
		Name Length too long	400:"Name cannot exceed 100 characters."	Passed
		Not putting email	400:"Email is required."	Passed
		Wrong email format	400:"Invalid email address format."	Passed
		Email too long	400:"Email cannot exceed 100 characters."	Passed
/api/user/login	POST	Logging in	200 OK:"User logged in successfully."	Passed
		wrong email and password combiantion	401: "Invalid email or password."	Passed
		not putting a password	400:"Password is required."	Passed
		Not putting email	400:"Email is required."	Passed

		Wrong email format	400:"Invalid email address format."	Passed
/api/user/logout	POST	Logging out	200 OK:"Logged out successfully."	Passed

Table 5

Meeting Minutes.
(Example sheet.)

MINUTES OF MEETING

Subject	SCRUM MEETING					
Date & Time	13/03/2025 09:00 to 10:00			**Location**	Online – Google Meet	
Meeting No.	10			**Minutes By**	Kimberly Ramasamy	(KR)

MEETING ATTENDEES

NO	NAME	INITIALS	ROLES	PRESENT
1	Michaël Etiennette	ME	Team Leader	✓
2	Kimberly Ramasamy	KR	Secretary	✓
3	Anushree Soondrum	AS	Developer	✓
4	Mohamed Uttanwalla	MU	Developer	✓
5	Lakshveer Mohesh	LM	Tester	✓

#	Discussion Points	Responsible
1	Welcome	**ME**
2	Apologies	×
3	Agenda Review	**ME**
4	Updates on current tasks • looking for the perfect API to feed data in the database tables – Mika & Kim • Main focus on logic for classes – Mohamed & Anushree • Testing with the branch we have on Github, use controllers to send Post request thus ensuring that each table is properly connected - Mohesh	**ALL**
	Task Allocation • Data source edits and further cleaning to add to table - Mika • User class interface creation – Anushree • User auth and transaction codes – Mohamed • Looking for smaller datasets – Kim • Testing of classes - Mohesh	**ME**
5	Q & A	
6	Any Other Business (AOB)	
6	Closing	**ME**

Table 6