**Project: Payment Application**

**Project Introduction**

Payment systems are now available everywhere and everyone interacts with these systems every day.

There are different types of transactions you can make, SALE, REFUND, Pre-Authorization, and VOID.

* SALE: means to buy something and its price will be deducted from your bank account.
* REFUND: this means that you will return something and wants your money back to your bank account.
* Pre-Authorization: means holding an amount of money from your account, e.g Hotel reservation.
* VOID: this means canceling the transaction, e.g if the seller entered the wrong amount.

You are required to implement the SALE transaction only by simulating the card, terminal(ATM), and the server.

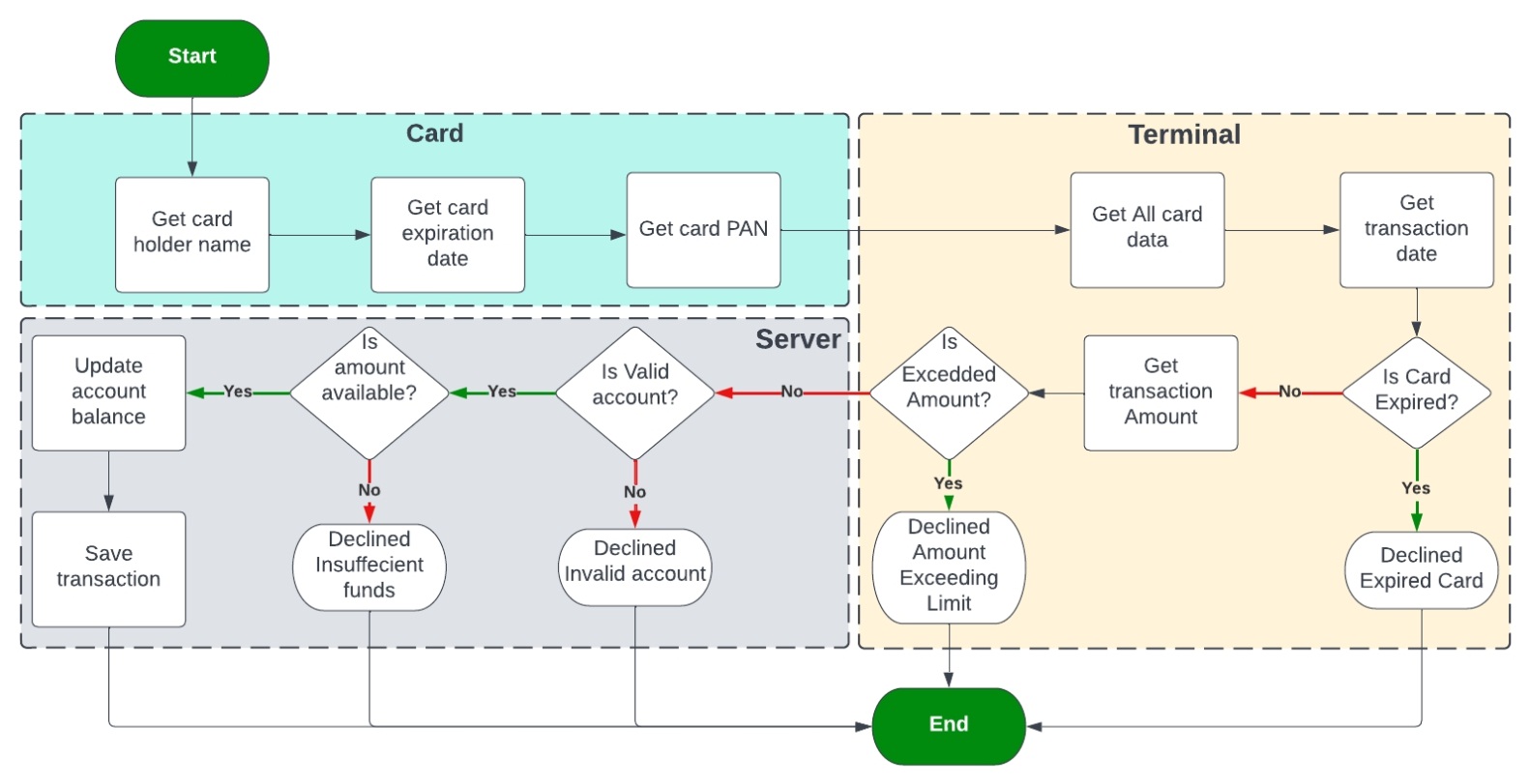
**Project Requirements**

1. Development environment preparation
2. Implement the card module
3. Implement the terminal module
4. Implement the server module
5. Implement the application
6. Testing the application

**Project: Payment Application**

**Project Instructions**

**Application Flowchart**



**The project can be distributed over 6 main tasks:**

**Development environment preparation**

1. Create modules folders
2. Create .c and .h file for each module
3. Add header file guard
4. Create the main.c file

**Implement the card module**

1. Fill in card.h file with functions' prototypes and typedefs
2. Implement getCardHolderName function
3. Implement getCardExpiryDate function
4. Implement getCardPAN function

**Implement the terminal module**

1. Fill in terminal.h file with functions' prototypes and typedefs
2. Implement getTransactionDate function
3. Implement isCardExpried function
4. Implement gatTransactionAmount function
5. Implement isBelowMaxAmount function
6. Implement setMaxAmount function

**Implement the server module**

1. Fill in server.h file with functions' prototypes and typedefs
2. Implement server-side accounts' database
3. Implement server-side transactions' database
4. Implement recieveTransactionData function
5. Implement isValidAccount function
6. Implement isAmountAvailable function
7. Implement saveTransaction function

**Implement the application**

1. Fill in application.h file with functions' prototypes
2. Implement appStart function

**Testing the application**

1. Transaction approved user story
2. Exceed the maximum amount user story
3. Insufficient fund user story
4. Expired card user story
5. Invalid card user story

For further details check the project [Rubric](https://review.udacity.com/#!/rubrics/4939/view)

## Project Specification

**Payment Application**

Development environment preparation

| **Criteria** | **Meets Specifications** |
| --- | --- |
| Create modules folders | 1. Create a new project 2. Create **"Application"** folder 3. Create **"Card"** folder 4. Create **"Terminal"** folder 5. Create **"Server"** folder   Note: To create a folder in Microsoft Visual Studio   1. In the solution explorer, right-click on the project name 2. Go to Add 3. Select Folder 4. Give a name to that folder   You should deliver a screenshot of the solution explorer that clarifies your folder structure. |
| Create .c and .h file for each module | 1. In the **"Application"** folder create **app.c and app.h** files 2. In the **"Card"** folder create **card.c and card.h** files 3. In the **"Terminal"** folder create **terminal.c and terminal.h** files 4. In the **"Server"** folder create **server.c and server.h** files   Note: To create a file into a folder in Microsoft Visual Studio   1. In the solution explorer, right-click on the folder you want 2. Go to Add 3. Select New Item 4. Select file type, .cpp or .h 5. If a .cpp is chosen, change the extension to .c 6. Give a name to that file"   You should deliver a screenshot of the solution explorer that clarifies files in each folder. |
| Add header file gaurd | 1. In the **app.h** file add the header file guard 2. In the **card.h** file add the header file guard 3. In the **terminal.h** file add the header file guard 4. In **server.h** file add the header file guard   You should deliver a screenshot for each .h file, the file name must appear in the screenshot, and the header file guard |

Implement the card module

| **Criteria** | **Meets Specifications** |
| --- | --- |
| Fill in card.h file with functions' prototypes and typedefs | 1. Use the following typedef as-is: 2. typedef struct ST\_cardData\_t 3. { 4. uint8\_t cardHolderName[25]; 5. uint8\_t primaryAccountNumber[20]; 6. uint8\_t cardExpirationDate[6]; 7. }ST\_cardData\_t; 8. typedef enum EN\_cardError\_t 9. { 10. CARD\_OK, WRONG\_NAME, WRONG\_EXP\_DATE, WRONG\_PAN 11. }EN\_cardError\_t; 12. Use the following prototypes as is: 13. EN\_cardError\_t getCardHolderName(ST\_cardData\_t \*cardData); 14. EN\_cardError\_t getCardExpiryDate(ST\_cardData\_t \*cardData); 15. EN\_cardError\_t getCardPAN(ST\_cardData\_t \*cardData);   You should deliver a screenshot of your card.h file |
| Implement getCardHolderName function | 1. This function will **ask** for the **cardholder's name and store it into card data**. 2. Cardholder name is **24 alphabetic characters string max and 20 min**. 3. If the cardholder name is NULL, **less than 20 characters** or **more than 24** will return a WRONG\_NAME error, else return CARD\_OK. 4. Test your function:    * Create a test function void getCardHolderNameTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, use the following as a guide:    * Tester Name: your name    * Function Name: getCardHolderName    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement getCardExpiryDate function | 1. This function will **ask** for the **card expiry date and store it in card data**. 2. Card expiry date is **5 characters string** in the format **"MM/YY", e.g "05/25"**. 3. If the card expiry date is NULL, **less or more than 5 characters**, or has the **wrong format** will return the WRONG\_EXP\_DATE error, else return CARD\_OK. 4. Test your function:    * Create a test function void getCardExpiryDateTest (void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, use the following as a guide:    * Tester Name: your name    * Function Name: getCardExpiryDate    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement getCardPAN function | 1. This function will **ask** for the card's **Primary Account Number and store it in card data**. 2. PAN is **20 numeric characters string**, **19 character max, and 16 character min**. 3. If the PAN is NULL, **less than 16 or more than 19 characters**, will return the WRONG\_PAN error, else return CARD\_OK. 4. Test your function:    * Create a test function void getCardPANTest (void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, use the following as a guide:    * Tester Name: your name    * Function Name: getCardPAN    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. You should deliver the test function as well as a screenshot of the results on the console. |
| Explain your work | 1. Record a video where you discuss each function you implemented in this module. 2. Explain and execute all test functions you made. 3. The video is **4 minutes maximum**. 4. You may record it in Arabic or English. 5. Muted videos will not be acceptable. 6. **All of the above are mandatory to pass this criterion**. |

Implement the terminal module

| **Criteria** | **Meets Specifications** |
| --- | --- |
| Fill in terminal.h file with functions' prototypes and typedefs | 1. Use the following typedef as is: 2. typedef struct ST\_terminalData\_t 3. { 4. float transAmount; 5. float maxTransAmount; 6. uint8\_t transactionDate[11]; 7. }ST\_terminalData\_t; 8. typedef enum EN\_terminalError\_t 9. { 10. TERMINAL\_OK, WRONG\_DATE, EXPIRED\_CARD, INVALID\_CARD, INVALID\_AMOUNT, EXCEED\_MAX\_AMOUNT, INVALID\_MAX\_AMOUNT 11. }EN\_terminalError\_t ; 12. Use the following prototypes as is: 13. EN\_terminalError\_t getTransactionDate(ST\_terminalData\_t \*termData); 14. EN\_terminalError\_t isCardExpired(ST\_cardData\_t \*cardData, ST\_terminalData\_t \*termData); 15. EN\_terminalError\_t getTransactionAmount(ST\_terminalData\_t \*termData); 16. EN\_terminalError\_t isBelowMaxAmount(ST\_terminalData\_t \*termData); 17. EN\_terminalError\_t setMaxAmount(ST\_terminalData\_t \*termData, float maxAmount); 18. EN\_terminalError\_t isValidCardPAN(ST\_cardData\_t \*cardData); // Optional 19. You should deliver a screenshot for your terminal.h file |
| Implement getTransactionDate function | 1. This function will **ask** for the **transaction date and store it in terminal data**. 2. Transaction date is **10 characters string** in the format **DD/MM/YYYY, e.g 25/06/2022**. 3. If the transaction date is NULL or is **less than 10 characters or wrong format** will return the WRONG\_DATE error, else return TERMINAL\_OK. 4. Test your function:    * Create a test function void getTransactionDateTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: getTransactionDate    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. **Optional**: The function will read the current date from your computer and store it into terminal data with the mentioned size and format. 6. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement isCardExpried function | 1. This function **compares the card expiry date with the transaction date**. 2. If the card **expiration date is before** the transaction date will return EXPIRED\_CARD, else return TERMINAL\_OK. 3. Test your function:    * Create a test function void isCardExpriedTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: isCardExpried    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 4. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement getTransactionAmount function | 1. This function **asks** for the **transaction amount and saves it into terminal data**. 2. If the transaction amount is **less than or equal to 0** will return INVALID\_AMOUNT, else return TERMINAL\_OK. 3. Test your function:    * Create a test function void getTransactionAmountTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: getTransactionAmount    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 4. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement isBelowMaxAmount function | 1. This function **compares the transaction amount with the terminal max allowed amount**. 2. If the transaction amount is **larger than the terminal max allowed amount** will return EXCEED\_MAX\_AMOUNT, else return TERMINAL\_OK. 3. Test your function:    * Create a test function void isBelowMaxAmountTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: isBelowMaxAmount    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 4. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement setMaxAmount function | 1. This function **takes the maximum allowed amount and stores it into terminal data**. 2. Transaction max amount is a **float number**. 3. If transaction max amount **less than or equal to 0** will return the INVALID\_MAX\_AMOUNT error, else return TERMINAL\_OK. 4. Test your function:    * Create a test function void setMaxAmountTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: setMaxAmount    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement isValidCardPAN function **(Optional)** | 1. This function will **check if the PAN is a Luhn number or not**. 2. For more about **Luhn number**, and how to generate and check please refer to this [Site](https://www.dcode.fr/luhn-algorithm). 3. If the number is not Luhn number, will return INVALID\_CARD, else will return TERMINAL\_OK. 4. Test your function:    * Create a test function void isValidCardPANTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: isValidCardPAN    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. If you are going to implement this function, please deliver the test function as well as a screenshot of the results on the console. |
| Explain your work | 1. Record a video where you discuss each function you implemented in this module. 2. Explain and execute all test functions you made. 3. The video is **4 minutes maximum**. 4. You may record it in Arabic or English. 5. Muted videos will not be acceptable. 6. **All of the above are mandatory to pass this criterion**. |

Implement the server module

| **Criteria** | **Meets Specifications** |
| --- | --- |
| Fill in server.h file with functions' prototypes and typedefs | 1. Use the following typedef as-is: 2. typedef enum EN\_transState\_t 3. { 4. APPROVED, DECLINED\_INSUFFECIENT\_FUND, DECLINED\_STOLEN\_CARD, FRAUD\_CARD, INTERNAL\_SERVER\_ERROR 5. }EN\_transStat\_t; 6. typedef struct ST\_transaction\_t 7. { 8. ST\_cardData\_t cardHolderData; 9. ST\_terminalData\_t terminalData; 10. EN\_transState\_t transState; 11. uint32\_t transactionSequenceNumber; 12. }ST\_transaction; 13. typedef enum EN\_serverError\_t 14. { 15. SERVER\_OK, SAVING\_FAILED, TRANSACTION\_NOT\_FOUND, ACCOUNT\_NOT\_FOUND, LOW\_BALANCE, BLOCKED\_ACCOUNT 16. }EN\_serverError\_t ; 17. typedef enum EN\_accountState\_t 18. { 19. RUNNING, 20. BLOCKED 21. }EN\_accountState\_t; 22. typedef struct ST\_accountsDB\_t 23. { 24. float balance; 25. EN\_accountState\_t state; 26. uint8\_t primaryAccountNumber[20]; 27. }ST\_accountsDB\_t; 28. Use the following prototypes as is: 29. EN\_transState\_t recieveTransactionData(ST\_transaction\_t \*transData); 30. EN\_serverError\_t isValidAccount(ST\_cardData\_t \*cardData, ST\_accountsDB\_t \*accountRefrence); 31. EN\_serverError\_t isBlockedAccount(ST\_accountsDB\_t \*accountRefrence); 32. EN\_serverError\_t isAmountAvailable(ST\_terminalData\_t \*termData, ST\_accountsDB\_t \*accountRefrence); 33. EN\_serverError\_t saveTransaction(ST\_transaction\_t \*transData); 34. void listSavedTransactions(void); 35. You should deliver a screenshot for your server.h file. |
| Implement server-side accounts' database | 1. Create a global array of ST\_accountsDB\_t for the valid accounts database. ST\_accountsDB\_t accountsDB[255]; 2. Fill in the array initially with any **valid data**. 3. This array has a **maximum of 255** element/account data. 4. You can fill **up to 10 different accounts for the sake of testing**. 5. Example of a **running** account: {2000.0, RUNNING, "8989374615436851"}. 6. Example of a **blocked** account, **its card is stolen**: {100000.0, BLOCKED, "5807007076043875"}. 7. You should deliver a screenshot of your accounts database array with a minimum of at least 5 different accounts for the different test cases, check all needed test cases in the **"Testing the application"** section. |
| Implement server-side transactions' database | 1. Create a global array of ST\_transaction\_t. 2. Fill in the array **initially with Zeros**. 3. This array has a **maximum of 255 element/transaction data**. 4. You should deliver a screenshot of your transaction database array |
| Implement recieveTransactionData function | 1. This function will **take all transaction data and validate its data**, it contains all server logic. 2. It checks the account details and amount availability. 3. If the account **does not exist** return FRAUD\_CARD, if the **amount is not available** will return DECLINED\_INSUFFECIENT\_FUND, if the account **is blocked** will return DECLINED\_STOLEN\_CARD, if a **transaction can't be saved** will return INTERNAL\_SERVER\_ERROR , else returns APPROVED. 4. It will **update the database with the new balance**. 5. Test your function:    * Create a test function void recieveTransactionDataTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: recieveTransactionData    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 6. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement isValidAccount function | 1. This function will **take card data** and validate **if the account related to this card exists or not**. 2. It checks if the **PAN exists or not** in the server's database (**searches for the card PAN in the DB**). 3. If the PAN doesn't exist will return ACCOUNT\_NOT\_FOUND and the account reference will be NULL, else will return SERVER\_OK and return a **reference to this account in the DB**. 4. Test your function:    * Create a test function void isValidAccountTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: isValidAccount    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement isBlockedAccount function | 1. This function **takes a reference to the account into the database and verifies if it is blocked or not**. 2. If the account **is running** it will return SERVER\_OK, else if the account **is blocked** it will return BLOCKED\_ACCOUNT. 3. Test your function:    * Create a test function void isBlockedAccountTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: isBlockedAccount    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 4. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement isAmountAvailable function | 1. This function will **take terminal data and a reference to the account** in the database and **check if the account has a sufficient amount to withdraw or not**. 2. It checks if the transaction's amount is available or not. 3. If the transaction amount **is greater than the balance** in the database will return LOW\_BALANCE, else will return SERVER\_OK. 4. Test your function:    * Create a test function void isAmountAvailableTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: isAmountAvailable    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 5. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement saveTransaction function | 1. This function will **store all transaction data** in the transactions database. 2. It gives a **sequence number to a transaction**, **this number is incremented once a transaction is processed** into the server, you must check the last sequence number in the server to give the new transaction a **new sequence number**. 3. It saves any type of transaction, APPROVED, DECLINED\_INSUFFECIENT\_FUND, DECLINED\_STOLEN\_CARD, FRAUD\_CARD, INTERNAL\_SERVER\_ERROR. 4. It will list all saved transactions using the listSavedTransactions function. 5. Assuming that the connection between the terminal and server is always connected, then it will return SERVER\_OK. 6. Test your function:    * Create a test function void saveTransactionTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.    * Print all results of your test cases on the console window, and use the following as a guide:    * Tester Name: your name    * Function Name: saveTransaction    * Test Case 1:    * Input Data:    * Expected Result:    * Actual Result:    * Test Case 2:    * Input Data:    * Expected Result:    * Actual Result:    * .    * .    * .    * Test Case n:    * Input Data:    * Expected Result:    * Actual Result: 7. You should deliver the test function as well as a screenshot of the results on the console. |
| Implement listSavedTransactions function | 1. This function **prints** all transactions found in the transactions DB. 2. Please follow the following format for only one transaction data: 3. ######################### 4. Transaction Sequence Number: 5. Transaction Date: 6. Transaction Amount: 7. Transaction State: 8. Terminal Max Amount: 9. Cardholder Name: 10. PAN: 11. Card Expiration Date: 12. ######################### 13. Test your function:     * Create a test function void listSavedTransactionsTest(void); to test all possible scenarios, happy-case, and worst-case scenarios.     * Print all results of your test cases on the console window, and use the following as a guide:     * Tester Name: your name     * Function Name: listSavedTransactions     * Test Case 1:     * Input Data:     * Expected Result:     * Actual Result:     * Test Case 2:     * Input Data:     * Expected Result:     * Actual Result:     * .     * .     * .     * Test Case n:     * Input Data:     * Expected Result:     * Actual Result: 14. You should deliver the test function as well as a screenshot of the results on the console. |
| Explain your work | 1. Record a video where you discuss each function you implemented in this module. 2. Explain and execute all test functions you made. 3. The video is **4 minutes maximum**. 4. You may record it in Arabic or English. 5. Muted videos will not be acceptable. 6. **All of the above are mandatory to pass this criterion**. |

Implement the application

| **Criteria** | **Meets Specifications** |
| --- | --- |
| Fill in application.h file with functions' prototypes | 1. Use the following prototypes as-is: 2. void appStart(void); 3. You should deliver a screenshot for your application.h file. |
| Implement appStart function | 1. Please refer to the **flow chart** attached under the **instructions video** in order to implement this application. 2. You should deliver **all project folders and files**. |
| Explain your work | 1. Record a video where you discuss each function you implemented in this module. 2. The video is **2 minutes maximum**. 3. You may record it in Arabic or English. 4. Muted videos will not be acceptable. 5. **All of the above are mandatory to pass this criterion**. |

Testing the application

| **Criteria** | **Meets Specifications** |
| --- | --- |
| Transaction approved user story | 1. As a bank customer have an account and has a **valid** and **not expired card**, I want to withdraw an **amount** of money **less than the maximum allowed** and **less than or equal to the amount in my balance**, so that I am expecting that the transaction is **approved** and my **account balance is reduced by the withdrawn amount**. 2. You should deliver a screenshot of the test result on the console. |
| Exceed the maximum amount user story | 1. As a bank customer have an account, that has a **valid** and **not expired card**, I want to withdraw an amount of money that **exceeds the maximum allowed amount** so that I am expecting the transaction **declined**. 2. You should deliver a screenshot of the test result on the console. |
| Insufficient fund user story | 1. As a bank customer have an account and has a **valid and not expired card**, I want to withdraw an amount of money **less than the maximum allowed and larger than the amount in my balance** so that I am expecting that the transaction **declined**. 2. You should deliver a screenshot of the test result on the console. |
| Expired card user story | 1. As a bank customer have an account and a **valid but expired card**, I want to withdraw an amount of money so that I expect that the transaction **declined**. 2. You should deliver a screenshot of the test result on the console. |
| Stolen card user story | 1. As a bank customer have an account and has a **valid and not expired but stolen card**, I want to block anyone from using my card so that I am expecting that any transaction made by this card is **declined**. 2. You should deliver a screenshot of the test result on the console. |
| Explain your work | 1. Record a video where you discuss each test case. 2. The video is **4 minutes maximum**. 3. You may record it in Arabic or English. 4. Muted videos will not be acceptable. 5. **All of the above are mandatory to pass this criterion**. |

## Suggestions to Make Your Project Stand Out!

1. In getCardPAN function:
   * Provide the PAN as a Luhn number.
2. Implement all optional functions.
3. For the server-side accounts DB:
   * Instead of a global array create a text file "Accounts DB.txt" that stores all account data and read this file into your application.
4. For server-side transactions DB:
   * Instead of a global array create a text file "Transactions DB.txt" where you will save all transactions and read if you need.
5. Test your application against the Fraud card:
   * As a bank administrator, I want to issue my own cards, so I am expecting that any transaction made by any fraud card (failed in Luhun check) is declined.