CMPS 312 Project Phase 1 – UI Design and Implementation
(20% of the course grade)

The project phase 1 submission is due by 11:59pm Thursday 31th October 2024. Demos will be organized during office hours in the same week.
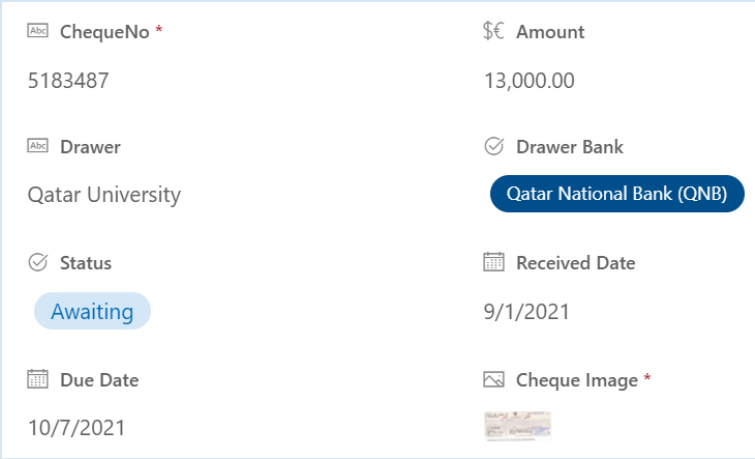
1. Requirements

You are requested to design and implement **YalaPay** B2B app for managing the payments of invoices. Its purpose is to streamline the financial transactions between a company and its customers by speeding the collection of payments from customers, which has a positive impact on the cash flow of the company and its profitability. YalaPay also manages the cashing cheques and provides valuable financial reports to track pending payments.

Note that storing data in a local database and reading/writing data from remote Cloud Firestore will be done in phase 2 of the project. For phase 1, data should be kept in memory and/or read from json files. The main YalaPay use cases are described Table 1.

Table 1. Use cases description

Use case	Brief description
U1 - Login	Allows the user to login. Login is prerequisite for all the YalaPay use cases described below. The list of users should be defined in `users.json` file. The minimum required user properties include: first name, last name, email, and password. No need to create a sign-up screen.
U2 - YalaPay Dashboard	The YalaPay Dashboard should display the summary of invoices and cheques by status as shown below. For phase 1 just design and develop the dashboard with some dummy values. In phase 2, you will use database queries to compute the summaries. **Invoices** All: **99.99 QR** Due Date in 30 days: **33.33 QR** Due Date in 60 days: **66.66 QR** **Cheques** Awaiting: **99.99 QR** Deposited: **22.22 QR** Cashed: **44.44 QR** Returned: **11.11 QR**
U3 - List/Search/Add/ Update and Delete a Customer	Provides the ability to List/Search/Add/Update and Delete a Customer. The minimum required customer properties include: customer Id, company name, address (street, city and country), contract details (first name, last name, mobile and email). When adding a new customer, the customer Id should be auto-assigned. The app should have an initial list of customers in `customers.json` file.

U4 - List/Search/Add/ Update and Delete an Invoice	Provides the ability to List/Search/Add/Update and Delete an invoice. The minimum required invoice properties include: Invoice number, customer Id, amount, invoice date and due date. When adding a new invoice, the invoice number should be auto-assigned. - The app should have an initial list of invoices in `invoices.json` file. - When listing or viewing an invoice the app should display the invoice details including the balance pending payment. The invoice balance is equal to the invoice amount minus the total payments excluding the returned cheques. - From the list of invoices, the user can also select an invoice and enter the payments associated with the invoice.

U5 - List/Search/Add/ Update and Delete payments for an invoice.	Provides the ability to List/Search/Add/Update and Delete payments for an invoice. An invoice can have one or many payments.
	The payment details include: the payment amount, payment date, mode of payment (cheque, bank transfer or credit card).
	For cheques more details should be entered as shown in Figure 1:
	  - Cheque No
	  - Amount
	  - Drawer
	  - Drawer Bank (the user should be able to select a bank from the list of banks in `banks.json` that includes banks in Qatar listed here).
	  - Status: should be set to *Awaiting*. Later the status could be changed to *Deposited*, *Cashed*, or *Returned*.
	  - Received Date (should be auto-set to today's date)
	  - Due Date
	  - Cheque Image (this attribute will store an image of the received cheque)
	
	Abc ChequeNo * $€ Amount
	5183487 13,000.00
	
	Abc Drawer ⊘ Drawer Bank
	Qatar University Qatar National Bank (QNB)
	
	⊘ Status 🗓 Received Date
	Awaiting 9/1/2021
	
	🗓 Due Date 🖼 Cheque Image *
	10/7/2021
	
	Figure 1. Example Cheque Details

U6 - Manage Cashing Cheques	One of core use cases of YalaPay if to manage the cheques caching process as shown in Figure 2. **6.1. Create Cheques Deposit:** - Select the cheques to be deposited. First the app should display the list of cheques **Awaiting** payment (see a sketch of the UI in the image below just to give you an idea).

☑	▽ Tireur	▽ Banque	▽ Numéro	▽ Date ▲	▽ Montant	▽ Statut	▽ Remise	⊞
☐	Client 1	CFG	897654	31/08/2020 (+13)	2 942,60	à déposer	0	✎ 🗑
☐	Brahim Lazrak	ATW	125891	24/08/2020 (+6)	2 000,00	à déposer	0	✎ 🗑
☐	M. Ali	ATW	133456	22/06/2020 (-57)	1 620,00	à déposer	0	✎ 🗑

Next to the Due Date show between parenthesis the remaining days to the due date (i.e., due date – today). Make the remaining days green positive and red if negative e.g., 30/10/2024 (+21), 01/09/2024 (-11). Red means the cheque is late to be deposited.
- Select a Bank Account to Deposit the cheques to.
Note the company may have 1 or many bank accounts stored in `accounts.json`. Just have a json document having the companies bank accounts no need to provide a UI to maintain them.
- Upon confirmation, the app should Create a **Cheques Deposit** having: the list of selected cheques, the *Deposit Date* (default is today's date) and the *Bank Account* to deposit the cheques to, *Status* set to *Deposited*.
Also update all cheques included in the Deposit and set their Deposit Date and their status to *Deposited*.

6.2. List Cheques Deposits:

List the Cheques Deposits created earlier and provide the ability to view, update and delete each entry (see a sketch of the UI in the image below just to give you an idea).

	Type Remise	Date Dépôt	Etat	Nb de Pièces	Total
🗑 📄 PDF	Remise Chèques	05/04/2016	En cours	4	3982,04
🗑 📄 PDF	Remise Chèques	30/03/2016	En cours	1	6490,48

When displaying the list of cheques associated with each Cheques Deposit, the app should provide the ability to show the cheque images and the *Total Amount* and the *Count* of the cheques to be deposited.

6.3. Update Cheques Deposit:

Once the Cheques Deposit has been processed by the bank (after few days) the app should allow Updating the status of the Cheque Deposit to either *Cashed or Cashed with Returns* and enter the **Cashed Date** (by default set it to today's date). This should also:
- Set the Cashed Date and the Cheques status to *Cashed* for all cheques included in the Deposit.
- For returned Cheques the user should be able to set the *Return Date* and the *Return Reason* (the user can select from a list of reasons from `return-reasons.json` file). Possible reasons of cheque return include (more info is available at this link - page 3):
  o No funds/insufficient funds
  o Drawer's signature differs
  o Alteration in date/words/figures
  o Drawer deceased/bankrupt

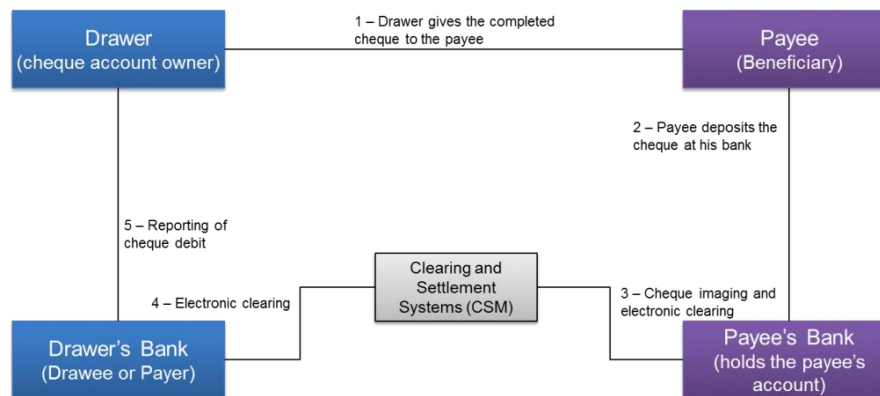|	
	o Account closed o Cheque information and electronic data mismatch
U7 - Invoices Report	Get the invoices for a period and by status. The user can enter the **from date** and **to date** and select invoice status: *All* (this is the default value), *Pending*, *Partially Paid*, or *Paid*. Upon submission, the app should display the invoice details including the balance pending payment. The invoice balance is equal to the invoice amount minus the total payments excluding the returned cheques. Also, the **total amount** and **count** of invoices should be displayed. In case the status is set to All then the **totals** and **counts** should be provided for *Pending*, *Partially Paid, Paid* and *Grand Total*.
U8 - Cheques Report	Get the cheques for a period and by status. The user can enter the **from date** and **to date** and select cheque status: *All* (this is the default value), *Awaiting*, *Deposited*, *Cashed*, or *Returned*. Upon submission, the app should display the cheque details. Also, the **total amount** and **count** of cheques should be displayed. In case the status is set to All then the **totals** and **counts** should be provided for *Awaiting*, *Deposited*, *Cashed, Returned* and *Grand Total*.

Figure 2. Cheque Cashing Process (source: https://www.paiementor.com/the-four-corner-model-for-cheque-payment/)

2. Deliverables

Seek further clarification about the requirements/deliverables during the initial progress meeting with the instructor. Note that further important clarifications maybe added/modified to the project requirements.

1) Application design documentation that includes UI design and the Repositories class diagram.

Start by designing the UI wireframe (sketch). Decide the UI components and the layout either on paper or use a design tool such as https://www.figma.com/ui-design-tool/

   During the weekly project meetings with the instructor, you are required to present and discuss your design and get feedback.

2) Implement **responsive** UI for each use case following design best practices. The UI should be fully working using json data files. Remember that 'there is elegance in simplicity'!

3) Design and implement the app navigation. It should be fully working, and the user can navigate from one screen to another in intuitive and user-friendly way.

4) Implement the entities and repositories using Dart. They should be fully working. Create some test json data to ease testing. First test them using a main function that displays the results to the console before using them in the UI.

5) Document the testing of UI and repositories using screen shots illustrating the testing results.

6) Every team member should submit a description of their project contribution. Every team member should demo their work and answer questions during the demo.

Push your implementation and documentation to your group GitHub repository as you make progress.

3. Grading rubric

Criteria	%	Functio nality*	Quality of the implementation
1) Application Design - Repositories Class Diagram.	5		
2) Design and implement responsive UI	60		
3) Design and implement the UI Navigation	10		
4) Implement the entities and repositories	20		
5) Testing documentation using screen shots illustrating the testing of UI and Repositories.	5		
6) Discussion of the project contribution of each team member [-10pts if not done]			
Total	100		
Copying and/or plagiarism or not being able to explain or answer questions about the implementation	- 100		

* Possible grading for functionality - Working (get 70% of the assigned grade), Not working (lose 40% of assigned grade and Not done (get 0). The remaining grade is assigned to the quality of the implementation.

In case your implementation is not working then 40% of the grade will be lost and the remaining 60% will be determined based on of the code quality and how close your solution to the working implementation.

Solution quality also includes meaningful naming of identifiers (according to Flutter naming conventions), no redundant code, simple and efficient design, clean implementation without unnecessary files/code, use of comments where necessary, proper code formatting and indentation.

Marks will be reduced for code duplication, poor/inefficient coding practices, poor naming of identifiers, unclean/untidy submission, and unnecessary complex/poor user interface design.