Ignatius

Good afternoon everyone, we’re Team 2.
[CLICK]

We’ll begin with the problem statement and the guiding question behind our project, followed by our use-case diagram, the live demo, external APIs, good software engineering practices and traceability.

[CLICK]

First, the problem statement. In Singapore, there’s currently no centralized platform that redistributes usable goods to lower-income families through community clubs.
Because of that, many good-condition items go to waste, while families in need still struggle to access essentials.
Our goal with *CareConnect* is to bridge that gap using technology to make giving and receiving easier and faster.

[CLICK]

This led us to our guiding question:
How might we create an accessible, community-driven platform that efficiently matches surplus goods from donors to the real-time needs of lower-income families through local Community Clubs?

[CLICK]

Our Solution CareConnect answers that question as it helps to connect the community through Tech and compassion.

Now let’s look at our use-case diagram.

[CLICK]
We have three main user types: Clients, Beneficiaries, and Community Centre Managers.

Clients can make donations to community centres, track their donation status, edit their donations, and subscribe to centres to monitor goods shortages.

Beneficiaries who are clients with a monthly income of $800 or less have all client capabilities, plus the ability to submit requests for needed goods, edit those requests, and track their fulfillment status.

The System intelligently manages inventory, matches donations with requests, and monitors expiry dates to ensure goods remain usable.

Community Centre Managers oversee their respective centres by verifying new users, accepting donated items, confirming handovers to beneficiaries, and accessing detailed transaction records.

[CLICK]

We use four key APIs in our system — Google OAuth for quick sign-in, Supabase Storage to store image files, the Community Club GeoJSON API for local CC data, and OneStreetMap API to show them interactively on the map.

[CLICK]

Now i will pass on the time to Wen Hong and Xiao Yao who will take you through the live demo

LIVE DEMO

Now we will start our live demo

We have 2 ways to log in to our app – using email or google authentication. We will go with google log in first. Since we will not fill in our full profile details during authentication, we will be prompted to complete our profile at the home page.

Now we will proceed to register an account for log in with email. In the reg form, we will put down our name, contact, income, email and password. We make sure that the phone number and email follows the valid formatting, income cannot be a negative number and password must be of the valid length. All sections must be filled in correctly before registration is successful.

Now, we are logged in with the account that we have just created. This brings us to the client home page which shows a map that contains all cc locations. There is a button 'locate' which locates the user on the map so that they can see their distance relative to the CCs.

Clients can make a request only if they have a household income of lower than 800. To do this, we will need the cc managers to verify the users. So now we will log in to a cc manager's account. All cc accounts will be pre-registered in our system. Our system will push a notification to alert the manager of the new registration. The manager will then toggle to the action page to verify it.

After approval, clients will receive a notification and they can now use the full app functions.

Now, we will have a look at the subscription page which allows users to subscribe to the different ccs. Once subscribed to a cc, you can get broadcast messages from that cc. The broadcast messages will be on severe shortage of fulfilled requests in the cc. This is flagged when the fulfilled requests run below 50 percent.

Now we will run through the create request procedure. Clients can make a request when they select a specific cc from the map. In the request form we will fill in the relevant information. We will put in a request for one canned food. Once the request is successfully created, it will be displayed on the My Requests page as pending. During this stage, clients can choose to edit or delete their request.

Since the fulfilled request is now below 50 percent, the donor will receive the broadcast message as mentioned earlier. At the same time, he can see a red anchor point on the map indicating the shortage for the cc and he can click into the link to toggle to the inventory page. Under this page, users can filter based on cc names. Each cc will have a card that displays the total donations, requests, fulfilled requests and items in demand. Then he decides to make a donation. The make donation process is similar to that of making a request. We put the donation item as canned food and we are donating 2 items. Additionally, we will need to upload an image of the donation for manager verification later. Also, since the donation made is in the category of food, we will need to include the expiry date. Similarly in the pending stage, the donor can still edit or delete their donation.

The manager will then get a notification to verify the donation under the action page. After it is approved, the donor will get a notification of approval. This is so that they will be notified to go to the cc and hand over the donation. We set a timeframe of 2 days for this action as if they do not go down for the handover, the donation will be automatically deleted from our system.

After handover, the manager will need to add the item into the inventory if the physical item is verified. Under the approved section, we will click add. This updates our inventory system.

Our system will do auto matching based on the current requests and added donations in our database. This process will be triggered in the following situations:

1. When someone created a new request
2. When a pending request that has partial fulfilment of items is deleted
3. When a donation has been added
4. When a donated item expires
5. When someone rejects a matched request
6. When no action is taken after a request has been matched

We will now go through the last two situations in detail. Since we have donated a matched item to our request, the request status is now matched. The requester will receive a notification about the match and they will need to head down to the cc for collection within 2 days. If this action is not done within the timeframe, their request will be deleted and they will be notified that their request has expired. After the requester collected the items, the manager will take action to mark the request as completed. Besides that, after the match, the requester can choose to reject it which is our last situation that triggers the automated matching process. We will show this by making another request of the same item using another account.

Before that we will first make another request with the first requester's account. Since we made 2 donations of canned food just now, this request will also be auto matched to the donation. Then we make a request of the same item with the second requester's account. This request will queue behind the first request. Now our first requester decides to reject his request. Then, the matching algorithm runs and our second requester will receive a notification that his request has matched.

Lastly, we will go through the dashboard page for cc managers. Although the page interface looks the same as that of the inventory page of the client, the manager will be able to click into specific ccs to view the detailed inventory information including the donations, requests and fulfillment percentage of specific items.

This is all for our app demo. Thank you for your kind attention.

AARON

Thank you for that wonderful demo, Wen Hong and Xiao Yau.
Now we know how our product works.
But to create a successful product, it shouldn't just work. It should be designed with good software engineering practices.

[NEXT]

During our system design, we clearly partitioned our system in 6 clear different layers:
User Layer
Presentation Layer
API Gateway
Application
External API
And finally, the Data Access Layer.
This technique also prevents the construction of a class with multiple roles, thus also demonstrating SRP principle, which we will discuss later.

[NEXT]
Now onto the Design Patterns we used…

[NEXT]
Firstly, we used the Strategy pattern (which separates the *what* from the *how*).

This is demonstrated clearly from the following examples:
Authentication Strategy
The Authentication Content (the *what*) uses a standard Authentication Strategy contract to perform a task, while the concrete strategies like GoogleOAuth or PasswordStrategy handle the specific *how* of the execution.

Notification Strategy follows the similar pattern.

[NEXT]
Next, we use the Observer Pattern by creating a mechanism where the ISubject Interface maintains a list of dependents. When its state changes, it automatically notifies all registered SubscriptionObserver instances (which implement the IObserver interface) without knowing their specific class details.

[NEXT]

Third, we also implement the Factory Pattern:
The App class uses the Database Factory class to create a new database object without specifying the exact class, thus decoupling itself from the creation logic. The Factory then creates and returns a concrete product (SQLite Database or PostgreSQL Database), which is referenced by the common Database Interface.

[NEXT]
A central Facade class is implemented to handle all steps, from User Authentication and Role Validation to executing the request. It internally coordinates the Database Factory for queries and updates, and manages the triggering of Background Jobs like the matching algorithm.
This design keeps the Controllers clean by delegating complex, multi-component workflow entirely to the Facade.

[NEXT]
Now onto how we demonstrated the use of SOLID principles:

[NEXT]
To enforce Single Responsibility Principle, we have used the ECB pattern by clearly separating appropriate classes into Entity (the data), Boundary (how the system interacts with the user) and Control Classes (how the application performs logic).
SRP is also exhibited in the Layered Architecture mentioned previously.

[NEXT]
Next, we will talk about the Open-Closed Principle.

Firstly referencing the example from the strategy pattern:
Notification Strategy
To add a new notification option, we just create a new strategy class that implements the NotificationStrategy interface. There is no need to alter any other code.

[NEXT]

Similarly, for Broadcast Observer, part of the Observer Pattern.
The observer interface allows new observer types to be added without changing the existing code.

[NEXT]
Lastly, referencing the Factory Pattern.
The database factory can create different databases — SQLite or PostgreSQL

And a new database type can be added simply as a subclass. Again, there is no need to edit existing logic.

Now over to Nikhilesh to continue with the SOLID principles and other aspects of System Design.

NIKHILESH

[NEXT]
Now I'll continue with the Liskov Substitution Principle, which states that child classes should be replaceable for their parent class without causing any issues.
This is reflected in our Notification Strategy.
Our context only depends on the parent notification strategy interface, not the individual concrete classes.
So whether we use Email, SMS, or Push Notification, we can swap them in and out *without changing existing code*, keeping our system flexible and future-proof.

[NEXT]
Next, the Interface Segregation Principle.

ISP states that no class should be forced to implement methods it doesn't need.
We followed this by designing focused interfaces, especially in our Observer Pattern.
For example, Isubject and IObserver are kept separate, so each class only implements what is relevant to its role.

[NEXT]
Finally, the Dependency Inversion Principle, which states that high-level modules should depend on abstractions, not on concrete implementations.
An example of this is the Authentication Context, which does not rely directly on Google OAuth or Password Login but on the IAuthenticationStrategy interface.
This reduces coupling and makes upgrades effortless.

[NEXT]
Apart from SOLID, we also ensured high code quality throughout our system.

[NEXT]
First, Code Reusability.
We avoid writing repeated logic by centralizing key functions.
For example, our allocation function is triggered from different scenarios, such as new requests, deletions, or matches, but the logic lives in one reusable module.
This ensures consistent behavior and makes maintenance much simpler.

[NEXT]
We also applied the same approach to our broadcast notification check, which is encapsulated in a single function.
This prevents repeated code, reduces errors, and makes updates faster since we only change logic in one place.

[NEXT]
Beyond those, we reuse UI components across multiple frontend pages, for example, CC cards, inventory list components, and map markers, which reduces front-end duplication and keeps the UI consistent.

[NEXT]
We also enforced strong Error Handling using try–except blocks with rollback mechanisms.
This prevents system crashes and ensures data integrity even when something fails, such as a database update.

[NEXT]
To support future developers, we maintained thorough Documentation: a setup guide, a clear file-structure map, API endpoint notes, and meaningful inline code comments.

[NEXT]

This reduces onboarding time and prevents knowledge silos — anyone joining the project can get productive faster.

[NEXT]
Finally, Security, a key priority in our system.
We used Redis-backed sessions, which store session data in an encrypted and centralized in-memory database preventing session hijacking and ensures that only authenticated users can access protected pages.

[NEXT]
On the second security layer, we protected user credentials using Argon2 password hashing.
Argon2 is currently considered one of the most secure hashing algorithms, because it is memory-hard, slow to crack, and highly resistant to brute-force and GPU-based attacks.
This ensures that user data remains safe and private.

[NEXT]
With that, I will now hand over to Rudolf, who will walk you through our Traceability section.

RUDOLF

[NEXT]
Before we dive into the details of a process and its design, it is important to understand the concept of traceability and its importance for successful software development. Traceability refers to the ability to link different elements of a system, from **high-level requirements** to **detailed designs**, **implementation and testing**. This ensures that **every use case** has a corresponding **design, code and test case**, providing a clear path from conception to delivery.

[NEXT]
I will be using the Login Use Case as an example for today. The precondition for this use case is that the user already has an account registered on the platform. The main flow includes **entering credentials, system authentication, and redirection to the appropriate dashboard if successful.**

[NEXT]
This class diagram highlights the **key components** involved in the Login process.

The AuthController, is essentially the central component in charge of handling authentication logic. It queries the **User** class by email to validate the provided password against the stored hash. Finally, the **AuthController** either grants access or returns the appropriate error based on the outcome (e.g., **401 Unauthorised or 403 Forbidden**).

[NEXT]
The flow of the Login process can also be visualised via a sequence diagram.
To briefly describe the flow that the sequence diagram is essentially showing,

1. Firstly, the User sends credentials (email, password)
2. Then, AuthController validates the email and password format. If either is invalid, it returns an error message.
3. The system then checks if the user exists by querying the **User class.** If the user is not found, it returns a **404 Not Found**
4. However, if the user exists → it will verify the **password hash.** If the password is **incorrect, it returns a 401 Unauthorised message**
5. If the password is correct, it will run an account status check, which issues a **session token and returns 200 OK if the account is CONFIRMED**
    - It will block the login with a **403 Forbidden if the account is Pending or rejected**

[NEXT]
During the process of designing the code structure, we made sure to adhere to positive design principles. One of them, for example, would be implementing the facade pattern, which can be seen in the code for the Login Use Case.

[CLICK FOR ARROWS TO APPEAR]

auth_routes.py defines routes and acts as a **boundary** layer that handles HTTP requests. It **calls functions** in the **AuthController**, which performs the actual logic for handling authentication (like login with password, Google OAuth, user registration, etc.).

[CLICK FOR ARROWS TO APPEAR]

AuthController is the **facade** that **abstracts the complexity** of authentication. It delegates tasks like **validating credentials**, **authenticating passwords**, and **managing user sessions** through its methods like login_with_password() and start_google_login().

We can see that it provides a **simplified interface** for different login methods, which ties back to the **Facade Pattern concept.**

[NEXT SLIDE**]**

Here, we are looking at **Cyclomatic Complexity**, a metric that measures the complexity of a program based on its decision points. For the **Login** process, the **Cyclomatic Complexity** is **4,** indicating that **4 independent paths need to be tested**

To ensure full coverage, we have identified the **basis paths** in the flowchart. These paths represent the different possible routes the system takes during the login process, covering all decisions and conditions. By testing these paths, we can confirm that **Login** works correctly in all scenarios, including valid login, invalid credentials, and account status checks.

[NEXT SLIDE]

This table outlines the **test conditions**, **inputs**, **expected outputs**, and **actual outputs**. We've tested all the **basis paths** to ensure comprehensive coverage of the login flow, including both **valid scenarios** and **boundary cases**.

All the test cases have **passed**, showing that the system behaves correctly for all tested scenarios. The **actual outputs** match the **expected outputs**, confirming the integrity of the **Login process**.