SOFTENG 306 PROJECT 1

Quality Report

Team one

Developer 1: Nick Huang
Developer 2: John Chen
Developer 3: Serena Lau

Table of Contents

1.0 Background

1.1 App Context

The venture into the digital age comes with increasing saturation in the market for digital devices, with vendors fighting for the wallets of consumers with every product release. This is perhaps fueled further by the recent coronavirus pandemic which has only highlighted the deep reliance of the modern economy on technologies at scale. As customers, it can therefore become difficult to navigate all these products offered and select the best fit for oneself, often having to traverse through a myriad of sites with each displaying product information differently, resulting in unnecessary cognitive load. Further, existing platforms in purchasing devices often lack emotional impact and visual appeal in the overall user experience in shopping for devices.

Nolo is our ecommerce Android application which seeks to address this space by becoming the “one stop shop” for laptops, mobile devices, and tech accessories. Nolo allows users to browse, search, and purchase their next favourite tech product. We wish for Nolo to become *the* go-to app for technology shopping which is why we elected for a high-end, modern, magazine-like look.

1.2 Report Structure

As dependence on software systems grows, the desire for low coupled and highly cohesive systems, absent of code smells, increases alongside. In *Agile Software Development, Principles, Patterns, and Practices*, Robert C Martin articulates the first five object-oriented design principles to help achieve such a design. These principles, better known as the SOLID principles, have been readily employed in industry today [1], allowing systems to be easily understood, maintained and extended.

In this report, we outline both the functional and non-functional principles which govern the creation of our application. We spend considerable time reflecting on our system design, compliance with SOLID principles in Section 2.0. Here, we also take a look into our workflow as a team and development methodologies employed. Section 3.0 outlines all functional attributes of our application, with specific notes on how our application complies with the project brief. We discuss UX and general design choices briefly in Section 4.0. We then conclude with acknowledgements and references. In sections where applicable, we make note of any inconsistencies between implementation and what was initially provided in our design document. Justifications are provided for such inconsistencies.

The developers working on this project are Nick Huang, Serena Lau, and John Chen.

2.0 Code Quality

2.1 System Design

2.1.1 Design Document System Design

In constructing our system design, we consult the official Android documentation for Android app architecture [2]. Here, we see that the recommended architecture is a UI-Domain-Data layered architecture (Figure 1). It is therefore logical to use this as a basis for our design. Further, sublayers can be expanded to that shown in bullet points below, with the following characterisation.

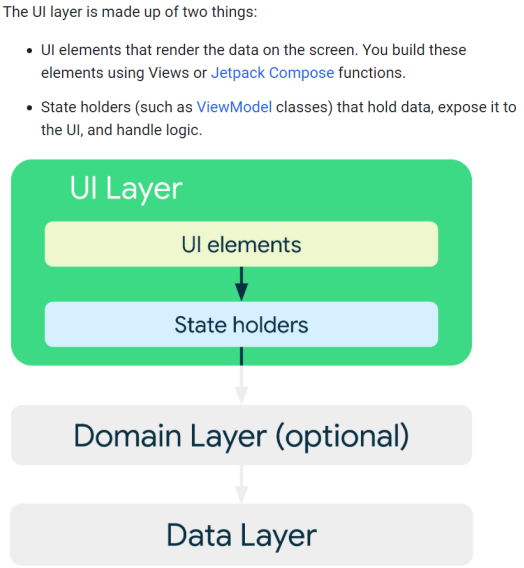


Figure 1. UI-Domain-Data layered architecture for Android Development [2]

Activities (aka UI elements)

- Renders the UI state from the ViewModel and handles UI user events, invoking the appropriate ViewModel methods.

ViewModels (aka State holders)

- UI "state holders" which performs UI behaviour logic and supplies UI state to Activities.

Interactors (aka Domains)

- Used to handle complex business logic (e.g., manipulating data from multiple repositories to hand back to the ViewModel) or allowing for the reuse of simple business logic.
- Although interactors are employed on a ***need-for*** basis as recommended in the official documentation, we choose to harness interactors for all use cases to facilitate the principle of interface segregation which will be expanded upon in Section 2.2.4.

Repository (aka Data)

- Deals with the database and exposes the necessary calls for interacting with the database.

We must consider other classes as well for our application, notably Adaptors and Entities. Moreso, interfaces should also be included.

Entities

- Represents the entities in the database, allowing the application to map between Firebase collection documents to workable objects.

Adaptors

- Handles the mapping of complex entities to dynamically generated ListViews/RecyclerViews etc.

Interfaces

- Improves system conformity to SOLID principles as discussed in Section 4.2

These were the packages that we considered for our design document. We see that it was heavily inspired by the recommended layered architecture in the official documentation with the addition of Adaptors working alongside Activities to support dynamically generated UI elements and Entities to define the models being stored and passed around throughout the sections and in the database. This can be seen in the diagram below.

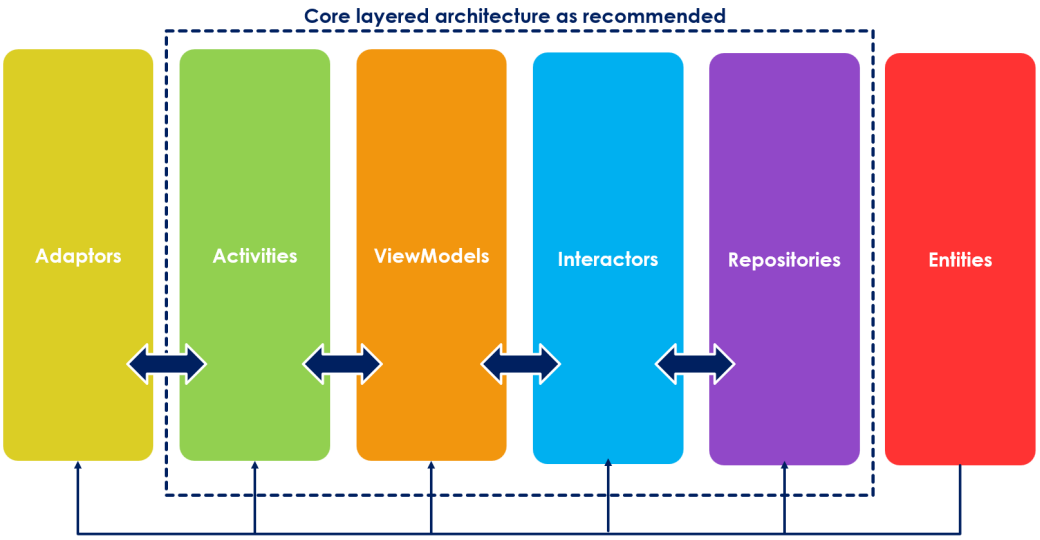


Figure 2. Design Document System Design Architecture

The system design itself is shown below as well.

[(view full-sized online here)]

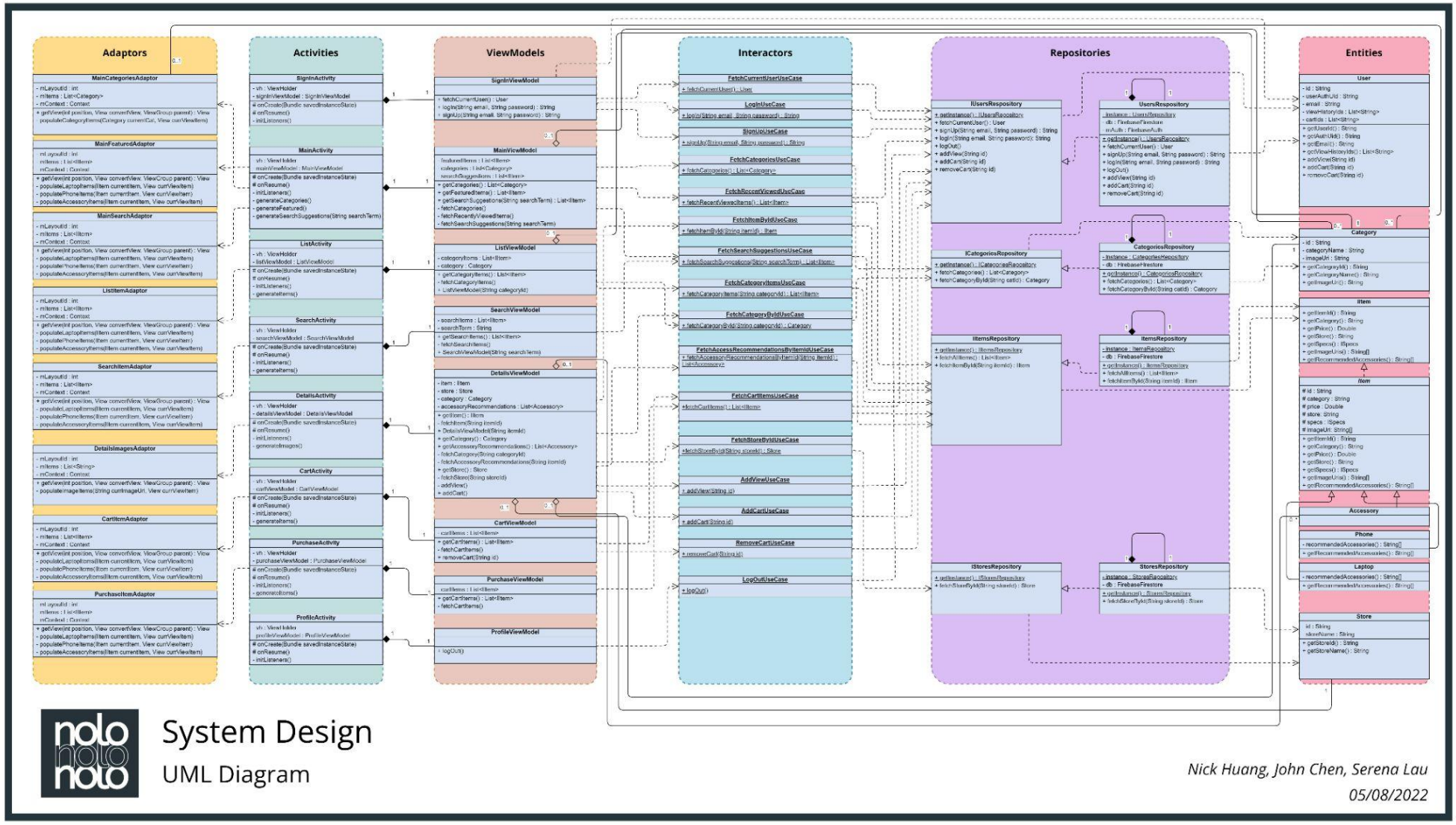


Figure 3. Design Document System Design UML Diagram

2.1.2 Realised System Design

The system design in which we employed in implementation did not differ greatly from that which we proposed in our design document. This is largely expected as we are working with an architecture deeply rooted in the official documentation for the software system that they support.

However, there were some areas to which we overlooked which we added in implementation. They are summarised below.

Fragments

- Our design requires a bottom In order to implement a bottom navigation bar for our app, fragments were required. Fragments essentially function in a similar way to activities but are all housed by a parent activity. This therefore does not pose an issue with our system design.

Enums

- Constant enumerations were required for things like category type, and repository paths. Therefore enums were employed. As they are used across all the classes, these exist on the same ‘level’ as entities. We do note that entities depend on enums as well.

Util

- We noticed in implementation, that there were certain functions that were used across our codebase and were not tied down to a particular view model, or activity. As such, in order to remove the code smell of duplicated code, we extracted such methods to a util package and invoked them through static methods.
- Such examples of util methods include the dynamic setting of height with list views; the retrieval of device width and height; and the retrieval of device location.

Data Provider

- In addition, a means was required to populate the firebase with data. No such means was catered for in our original design. Thus a data provider class was created which contained all the population (as well as clearing) logic for preparing firebase for app demonstration.

To see the forest for the trees, we take these into consideration, leaving us with an updated system design. Here we see that we have pretty much the same interaction as had prior. Now enums, util, and entity classes all feed into the other segments of classes. Fragments exist on the same level as activities (View segment) and talk to both its parent activity but also its view model as well as adaptors to handle dynamic UI generation. The data provider is isolated from the main functional classes as its sole purpose is to populate firebase programmatically.

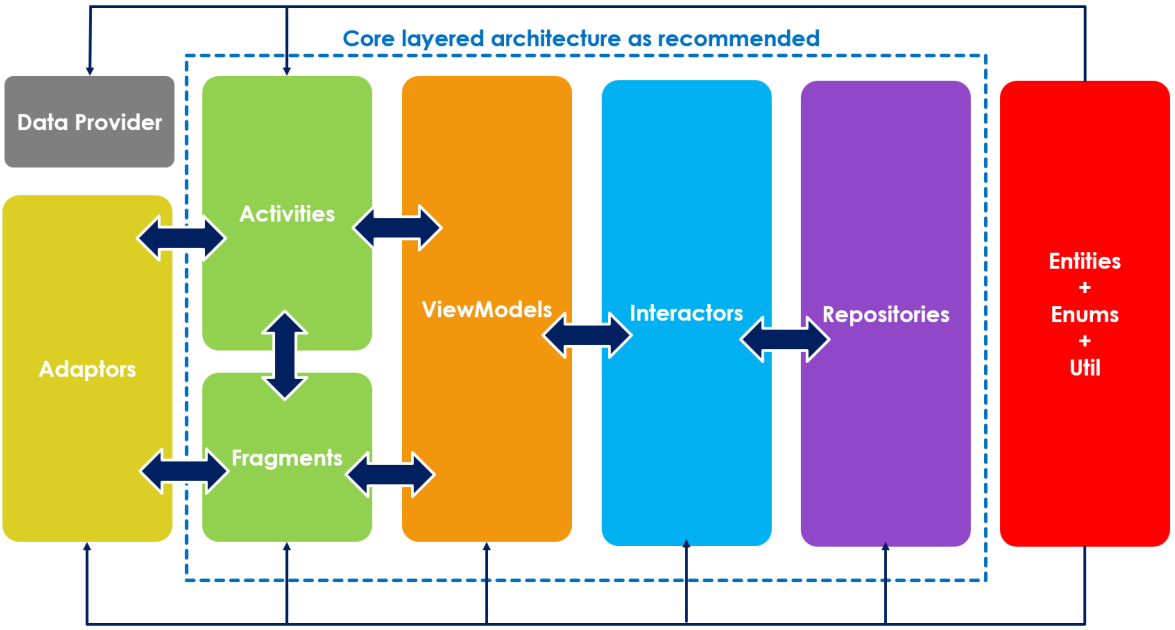


Figure 4. Realised System Design Architecture

We see that this implementation was employed in our GitHub repository as seen in our readme overview of our package structure.

README.md

```
├── ...
├── app/src
|       ├── ...
|       ├── main
|       |       ├── ...
|       |       ├── java/com/example/nolo
|       |       |       ├── ...
|       |       |       ├── activities          #all full screen UI elements
|       |       |       ├── enums               #enumerations
|       |       |       ├── fragments           #all partial screen elements, including bottom nav fragmen
|       |       |       ├── viewmodels          #associated view models to activities + fragments
|       |       |       ├── dataprovider        #for loading data to firebase
|       |       |       ├── entities            #entity representation
|       |       |       ├── interactors         #domain level use cases to interface view models and repo
|       |       |       ├── repositories        #for firebase interaction
|       |       |       ├── util                #for utility methods used across the application to reduce
|       |       |       ├── adaptors            #to house adaptors which map dynamically sized lists of da
|       |       |       └── ...
|       |       ├── res
|       |       |     ├── ...
|       |       |     ├── drawable              #all image assets and icons, as well as associated selecto
|       |       |     ├── dimen                 #all dimension constants
|       |       |     ├── strings               #all string constants
|       |       |     ├── layout                #all layout xml files
|       |       |     ├── navigation            #navigation with bottom nav
|       |       |     ├── menu                  #bottom nav xml
|       |       |     ├── font                  #fonts
|       |       |     └── ...
|       |       └── ...
|       ├── androidTest/java/com/example/nolo   #integration tests
|       ├── test/java/com/example/nolo          #unit tests
|       └── ...
├── assets_raw                                  #for all processed assets not stored in `res` directly
└── ...
```

Figure 5. Realised Package Structure

Due to the complexity of our final realised system, we neglect to show an updated system design UML diagram. However, we believe such a schematic is sufficient to show the broad consistency with our initial design as well as pinpointing areas of inconsistency and providing the required justification.

2.1.3 Inconsistencies

With this all said, there are some specific areas of inconsistency we think are noteworthy with regards to our system design. These are not so much the overall architecture, but implementation specific aspects which would have changed the inner members of our UML class diagram (e.g., methods and fields). We outline these here. Note also that due to the tight coupling between discussions of system modelling and system design, the inconsistencies outlined here encapsulate some inconsistencies seen with system modelling. We refer the reader back to our design document for discussion on system modelling.

In all our discussions we wish to solidify the idea that although there were some changes to our system design, all these are in line with the general architecture initially proposed in the design document and currently implemented in the codebase. Thus, all changes to the system design unless otherwise specified obeyed the SOLID principles and followed other good code conventions.

2.1.3.1 Caching

Initially, we didn't consider that firebase calls would be asynchronous. In our design document, we assume all calls would be synchronous. E.g., we would have a getItemByID(String id) method in our interactor class. We figured we would have to rethink how we make data access requests as data "takes some time" (is asynchronous) in nature.

One option would be to wait for each data access to complete with every data retrieval. This would work but poses issues for the following reasons. First, it requires that our user must wait for some small but still considerable amount of time when traversing the application, thereby degrading the overall user experience. Secondly, it means that our application will be a lot more verbose as every data access call - and there are alot of them - will require a consumer callback to be passed as a parameter and such a callback will need to be cascaded right from the activity through the layers and into the repository. Thus, this option is also not ideal in terms of complexity and ease of maintainability.

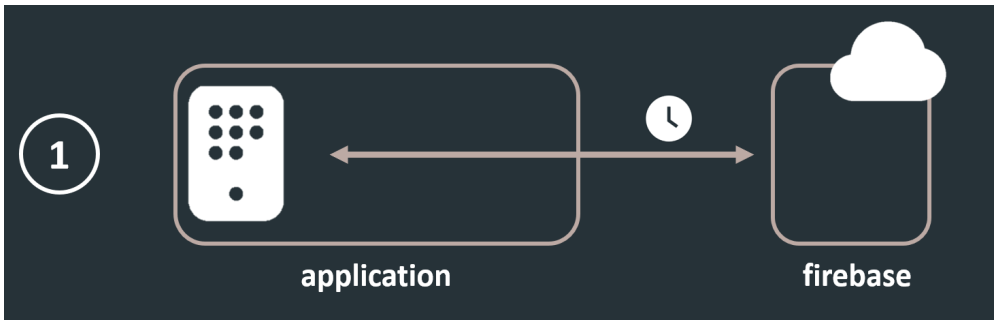


Figure 6. Option 1: waiting for asynchronous calls to finish

A second option and our chosen option is to use caching. Now, we load all data into a caching layer on the repository level. This occurs on app startup and progress is shown to the user in the custom loading animation shown on the splash screen. This allows fast

synchronous access in subsequent data access calls as we are retrieving straight from the cache. A time to live (TTL) token is used to ensure consistency in the cache with live data from firebase.

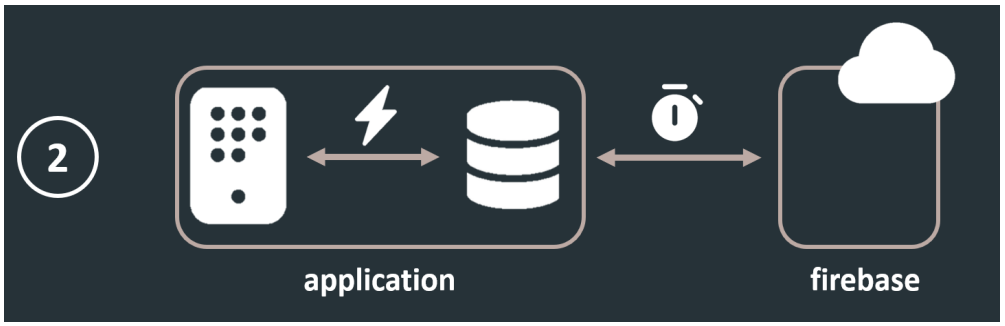


Figure 7. Option 2: caching layer with TTL token

In terms of implementation changes to system design, we had to create several classes, methods and fields for the loading, caching, timeout and retrieval mechanisms of data. I.e., a loadCategories interactor is needed to be called from the Splash activity view model which stores categories in a memory cache on the repository layer before signalling completion via a callback.

2.1.3.2 General Complexity

Since submitting our design document, we've added a considerable amount of features to implement for our application. These additions are summarised in Section 4.4 but they include the addition of wishlist, a map, historical purchases, and modification of user settings. Due to these additions, we have had to add more classes and relationships between such classes to our system design. Our system design has had to also accommodate for more complex GUI interactions than initially planned, such as snap scrolling on home fragment or the unique scrollable flow and expansion toggling in details activity.

2.1.3.3 UI state in ViewModels

In our initial system design, we decided to store all data accessed by an activity in the view model, even if it is not used again. We deemed that this was actually redundant and could fall under the code smell of needless complexity or speculative generality. Thus, our realised system design will only store the state of data that is either fetched multiple times or modified in interaction. For example, ItemVariant is managed in state for DetailsActivity as we must keep track of what variant (what colour, quantity, customisation etc.) we are dealing with in the state.

2.1.3.4 Interfaces

In our design document, we mentioned how we would be employing interfaces for all classes with the exception of interactors, and activities. In practice, the only classes we added interfaces for were view models, entities, and repositories. These interfaces ensure that compliance with SOLID principles is achieved and that a clear contract is provided for these classes. Other classes such as enums, adaptors, and interactors were deemed unnecessary recipients of adaptors. This is as they either housed a very definable single responsibility (interactors and enums) or were highly coupled with UI display (adaptors and

activities). Thus, this is how our use of interfaces have changed between our design documents .

2.1.3.5 Base Activity

Another important consideration with regards to SOLID is the use of a base activity custom class. The base activity is used to hide the title bar, set the status bar colour and lock the screen to portrait. Although this seems to violate Liskov's Substitution Principle as the base activity does not implement all the methods in its children activity classes (that would be unnecessarily complex), we decided to use a base activity anyways. We believe this is justified as activities are by nature tightly coupled with the Activity class provided by android and thus the UI. As such, we would never run into a situation where we would want to represent a child activity by its parent.E.g., BaseActivity activity = (BaseActivity) getActivity(). Thus, we felt it was appropriate to use BaseActivity as a parent class. Furthermore, the activity classes provided by android to which we extend from do not implement all the methods added to our activity classes anyways so we deemed it fine to continue with this notion by inserting a base activity class within the hierarchy.

2.1.3.6 Firebase Matching with Interfaces

There is a particular case where we have had to forgo using interfaces in the type for fields in entity classes. These instances are where entity classes stored in Firebase have fields which are themselves developer-defined entity classes. For example, Consider the Item class which houses a list of StoreVariants. We are unable to use List<IStoreVariant> and are forced to use List<StoreVariant>. This violates DIP. The reason for this violation is to allow Firebase to serialise and deserialize objects properly. Firebase requires that the data type being referenced has a 0-argument constructor. However, a 0-argument constructor cannot be provided in interface classes so we arrive at this dilemma.

The team decided that we were willing to violate this SOLID principle in favour of having code that works with Firebase with nice encapsulation of child objects.

2.1.3.7 Code Replication

We wish to take the time to quickly discuss the replication of code. An effort to minimise code replication was employed when developing and refactoring. However, there are some remnants where code has been repeated. This is often due to functionality being tightly dependent with the responsibility of that class. For instance, we had considered extracting much of the logic of the sign in, log in, and change password activities to some util class as these all shared similar functionality (e.g., toggling password). However, because toggling passwords are so tightly related to various views, these views would need to be passed in parameters. The team decided we preferred some small amount of code replication over unnecessary complexity and clutter in passing large amounts of views into util static methods. This rationale was followed throughout development and justifies why some code repetition may be present.

2.2 Application of SOLID principles

We now turn our discussion into how our system design complies with SOLID principles. Note that noteworthy inconsistencies with our design document have already been covered in Section 2.1.3, namely in 2.1.3.4 and 2.1.3.5. This aside, our realised compliance with SOLID principles matched closely with that in our design document.

2.2.1 Single Responsibility Principle

All the classes have one responsibility. All classes in the Entities section are responsible for storing the data for a given entity and house its associated operations. Likewise, each Repository class is responsible for accessing data in Firebase for a given Firebase collection. In a similar way, each Adaptor class is responsible for converting a collection of entity objects to dynamically generated views. Moreso, each Activity handles only the UI elements for a single screen and each ViewModel class manages the state for a single Activity class. We see that this pattern continues for each layer of our system design, where each layer designates a specific responsibility over a single element of the layer's domain. Furthermore, all the methods in each class have one responsibility as well. A good example of this is the Repository classes. Each method in the Repository class performs a single function. For instance, in ItemsRepository, there are four distinct methods

```
void loadItems(Consumer<Class<?>> onLoadedRepository);
List<IItem> getAllItems();
IItem getItemById(String itemId);
List<IItem> getCategoryItems(CategoryType categoryType);
```

These are clearly defined and distinct in responsibility. In addition, the use of interfaces which abide by Single Responsibility Principle (SRP) enforces a contract so that each class which realises the interfaces conforms to SRP as well. Thus, each class and method has been designed to have only one responsibility, therefore no SRP has been violated.

2.2.2 Open-Closed Principle

Open-Closed Principle (OCP) means that objects or entities should be open for extension but closed for modification. This can be seen in the Repositories and Interactors sections. Each Repository class has its own interface. So adding new Interactors classes will only result in the extension of the Repositories classes and it should not modify the existing code/existing UseCase methods. Therefore, adding new functionality into the classes in our class diagram will not violate OCP in our system design. The use of interfaces also enforces a contract on those which realise those interfaces, making those classes open for extension and closed for modification.

2.2.3 Liskov's Substitution Principle

Liskov's Substitution Principle (LSP) indicates that a subclass should be able to be used in the same manner as the superclass without violating any of the superclass policies and methods. In other words, any subclass implementation can be substituted for its superclass and still behave expectedly. This can be seen in the Item class in the Entities section. Item implements getRecommendedAccessories() from IItem such that it throws an exception as not all subclasses implement this method. Then, all three subclasses (laptops,

phones, and accessories) inherit the exact form of the abstract superclass. Laptop and Phone classes override the getRecommendedAccessories() method as they both provide recommended accessories. Accessory inherits the parent method implementation which throws an error. Thus, by implementing it in this manner, all subclasses, whether Accessory or not, can be used in the same manner as the Item superclass without violating its policies and methods (e.g., the getRecommendedAccessories() is always present). By doing this, it prevents violating LSP. Note our justification of an exception in Section 2.1.3.5.

2.2.4 Interface Segregation Principle

ISP states that a client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use. This can be seen in ViewModels, Interactors and Repositories sections. Each Repository class has all the methods that are related to accessing Firebase data for a given Firebase collection. Each ViewModels class only requires some of the methods in the Repository class and not all of them. To prevent violating ISP, we decided to use middle-man classes between ViewModels and Repositories, and it is called Interactors. This means that the client (ViewModel) classes are not forced to depend on the entire Repository and therefore have access to methods it will not and should not have permission to use.

2.2.5 Dependency Inversion Principle

Our class diagram follows the Dependency Inversion Principle (DIP) which states that no class should be associated with concrete classes and the classes should always be associated classes with interfaces. Most classes have interfaces associated with them. Exceptions are justified in Section 2.1.3.4. Because other classes are always associated with interfaces, we know that no class will be associated with concrete classes directly, therefore we do not violate DIP here either. Exceptions are justified in Section 2.1.3.6.

2.2.6 Code Smells + Coupling + Cohesion

Our final system design uses onion-layered design from Android app architecture to separate the modules into the layers or sections. Such sections are depicted in the schematic, Figure 4. The classes are divided based on their purpose and then grouped into their respective distinct layers. The reasons for grouping them into layers is to decrease the coupling between the modules, making each class in charge of a single distinct and well defined responsibility and also to make each class only able to be associated with classes through its defined interfaces. Furthermore, only adjacent layers are able to talk to each other allowing for a modular, systematic view of the entire system. All these assist in solving the code smells in our initial design.

Rigidity, fragility, immobility, viscosity and opacity smells are minimised by applying the layered design as well as the use of interfaces. Because our design follows the SOLID principles, it naturally will be less fragile (less fragile to change/altercation), less rigid (receptive to change), more mobile due to its modular and well defined classes. The clear and clean system design removes opacity and viscosity smells as we can easily understand the roles of all classes and all logic is done the right way because careful design has been employed from the outset. As classes are separated into layers, each layer has its responsibilities, such as Entities classes being responsible for holding data of each object,

and with ViewModels classes being responsible for the UI of its corresponding screen. This makes the design more transparent as it is now easier to see where the data is accessed from and where it is used. The modular layered architecture also increases both testability and maintainability. We see this in action with the modular integration tests (one for each repository) employed in our repository.

We will not go into an extensive analysis of the software metrics for our new system design. This is as our system design is largely consistent in nature as that described in the design document. Moreover, the design document metrics analysis concluded that our initial design was highly maintainable, highly reusable, low coupled and highly cohesive. We can therefore be confident that the system design we realised is much the same. We refer the reader to our design document for further inquiry.

2.3 Development Workflow

To facilitate good code practice, we employed various tooling such as Git, GitHub actions, extensive testing and documentation, and conformity to roles and responsibilities. We make note of any inconsistencies here and provide the appropriate justifications.

2.3.1 Git

We have used Git throughout to manage version control and history. We have also used it for the creation and merging of feature branches which facilitate the “sprint-like” methodology expanded upon in Section 2.4. All developers were proficient in using Git so this was easily employed. Git was particularly useful when breaking changes were made (there were a few!) and rollback was required.

2.3.2 GitHub actions

GitHub actions were added to run automated tests on the pushing of staged commits to GitHub (as well as on pull request). These ensured that all unit tests passed but, more notably, that the application successfully built. This provided a quick and easy means to gauge whether there were breaking changes that needed to be fixed.

2.3.3 Testing

Integration tests were written to test the majority of the interactor use cases to ensure that repository methods are working as expected. Unit tests could be used to test entity manipulation methods but these were deemed unnecessary given the simplicity of entity manipulation and time constraint the team faced.

2.3.4 Naming

Consistent naming was used throughout to ensure maintainability and readability. A summary of the naming is given below.

Aspect	Convention
packages	Lowercase (no spaces)
branches	AB#[ticket number]_some-description
commits	[commit type]: [msg] *commit types are expanded on in Section 2.4
fields/methods	Camelcase
classes	Pascalcase

Packages were appropriately named to reflect the layered architecture shown in our system design architecture.

2.3.5 Documentation

Javadoc comments were used throughout to ensure readability and maintainability of the codebase. Particular care was taken in explaining the rationale behind some of the more non-standard additions to the codebase.

2.3.6 Roles & Responsibilities

The table and discussion below outlines the roles and responsibilities that we coined in the design document.

Name	Roles	Responsibilities
Everyone	Documentation and constructing report	Actively participate in the documentation of code repository and sprint tickets as well as the construction and refinement of the final report.
	General testing	Actively participate in general use case testing across various devices and emulators. This serves as the integration tests.
Nick Huang	Secretary*	Records discussions during meetings as well as any other key project decisions.
	Backend Lead*	Leads the development of Repository, Entity, Enum, Interactor, and Data classes as well as their respective interfaces. This also includes creating an extensive suite of automated unit tests.
	Frontend	Assists with the development of ViewModels, Activity, Entity, and Adapter classes; their respective interfaces, and the implementation of view transitions.
Serena Lau	Scrum Master*	Leads sprints, maintaining a view of tickets, their individual progress, and team responsibilities across sprints.
	Design Lead*	Leads the creation of XML layout, theming, and style files, as well as the continued refinement of UX design prototype.
	Frontend	Assists with the development of ViewModels, Activity, Entity, and Adapter classes; their respective interfaces, and the implementation of view transitions.
	Backend	Assists with the development of Repository, Entity, Enum, Interactor, and Data classes as well as their respective interfaces. This also includes creating an extensive suite of automated unit tests.

John Chen	Project Manager*	Manages the project, with a holistic view of project deadlines and ensuring deliverables are met in due time and of sufficient quality.
	Frontend Lead*	Leads the development of ViewModels, Activity, Entity, and Adapter classes; their respective interfaces, and the implementation of view transitions.
	Backend	Assists with the development of Repository, Entity, Enum, Interactor, and Data classes as well as their respective interfaces. This also includes creating an extensive suite of automated unit tests.

The rationale behind these role allocations was influenced by the following considerations.

Admin roles

- John has had experience leading software teams in the past and has a keen eye for detail and meeting deadlines. Thus, he will take on the role of project manager for the duration of the project.
- Serena has had experience in sprints, utilising kanban boards and GitHub workflows, and other Agile methodologies. Thus, she will lead the development progress of Nolo for the duration of the project.
- Nick is proficient in keeping track of what has been discussed and extracting ideas of significance. Thus, he will be our secretary for the duration of the project.

General

- Due to the nature of the final report, all individuals will be involved in its construction. Moreso, documentation and testing are integral parts of the software system so will need to be partaken by all parties extensively. It is also suitable as team members are responsible for certain proper subsets of the codebase so, with everyone documenting and testing, we can ensure that those who understand the code best are there to test and document that code. Of course, pair programming and other collaborative strategies will also be employed.

Leads

- We have divided up the technical roles into three sections, namely design, frontend and backend.
- Leads were then assigned one to each team member based on their proficiency and interest. Leads will be those responsible for that segment of the software product.
- However, we do envision that all members will be across the entire codebase which is why we've assigned secondary roles as well. For instance, Nick will be leading backend development but also assisting with frontend. This ensures that there lies effective collaboration among the team. In practice, sprint tickets will be assigned to individuals to facilitate teamwork.
- Of note, Serena will be assisting with two secondary teams on top of leading design. The rationale for this is that design will be completed rather early on and so more work can be provided to her as we move into the late stages of development.

2.3.7 Consistency

We found that the realised roles and responsibilities were largely consistent with the design document for the duration of the project. We found that we knew each other quite well at the genesis of the team assignment process and thus were able to allocate roles effectively that played to each individual's strengths.

2.4 Project Management

We now begin a quick discussion into our team's project management.

2.4.1 Sprints

We attempted to employ a variation of Agile methodology for the duration of the project. To assist with this, we set up an Azure DevOps board and used tickets to create and track the progress of tickets. Tickets were organised under feature labels and visualised in a kanban board. Here's a snapshot of the board.

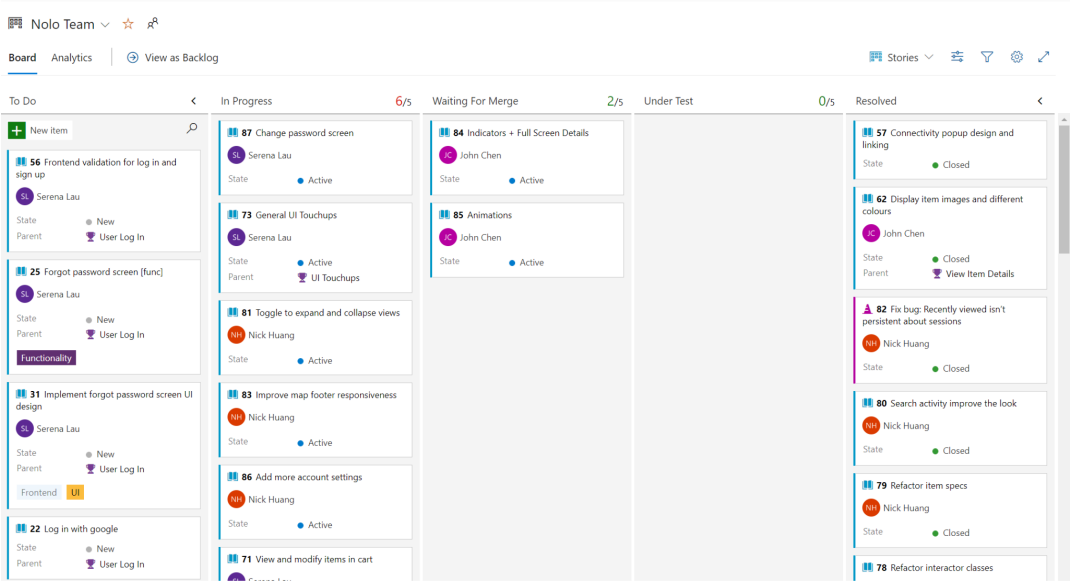


Figure 8. Azure DevOps board

For each ticket, we create a feature branch off main with the naming convention shown in Section 2.3.4. All feature branches were then merged back into main via pull requests with the mandatory approval of all other developers.

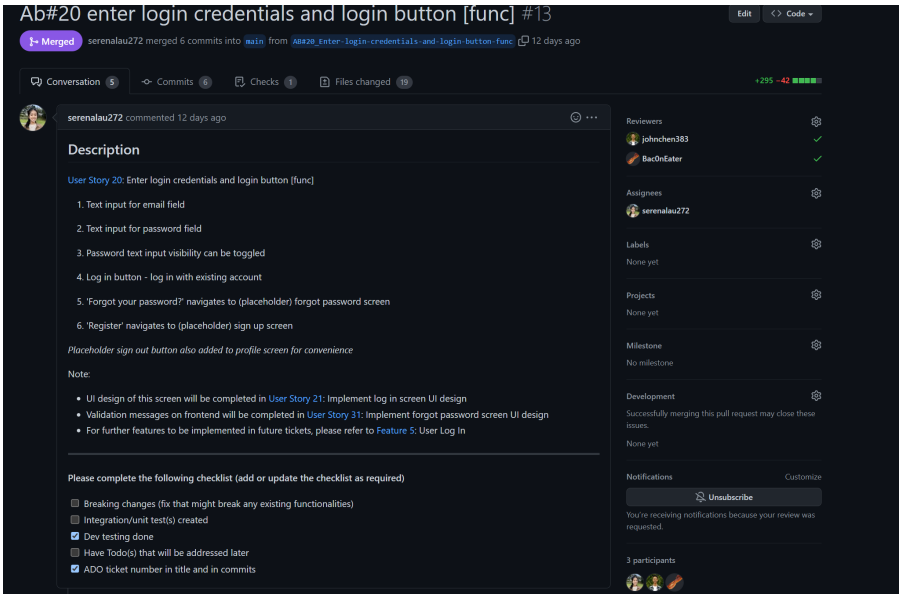
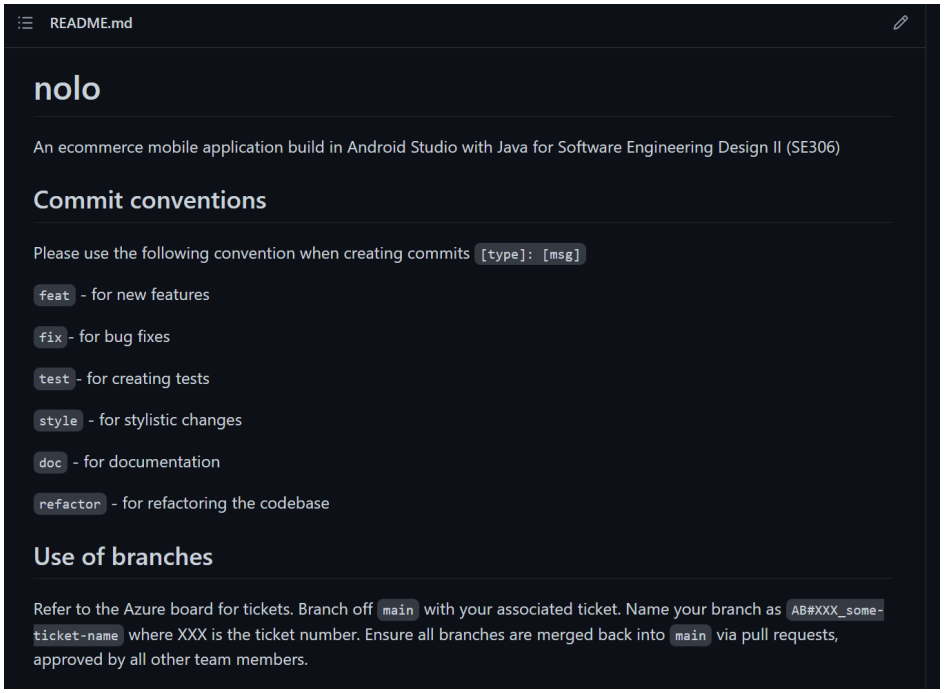


Figure 9. Example pull request

To assist with the readability of changes to the codebase, and therefore maintainability of the repository, commit naming conventions were applied - also outlined in Section 2.3.4. The commit types are given below.


README.md

nolo

An ecommerce mobile application build in Android Studio with Java for Software Engineering Design II (SE306)

Commit conventions

Please use the following convention when creating commits `[type]: [msg]`

`feat` - for new features

`fix` - for bug fixes

`test` - for creating tests

`style` - for stylistic changes

`doc` - for documentation

`refactor` - for refactoring the codebase

Use of branches

Refer to the Azure board for tickets. Branch off `main` with your associated ticket. Name your branch as `AB#XXX_some-ticket-name` where XXX is the ticket number. Ensure all branches are merged back into `main` via pull requests, approved by all other team members.

Figure 10. Dev conventions (from readme)

2.4.2 Meetings

Meetings were conducted using Notion, which also housed our minutes and any supporting documents. We also utilised Slack as our communication platform of choice with bots such as Geekbot to facilitate daily standups. We felt that these ensured we were able to keep on track in meeting the deliverables at hand.

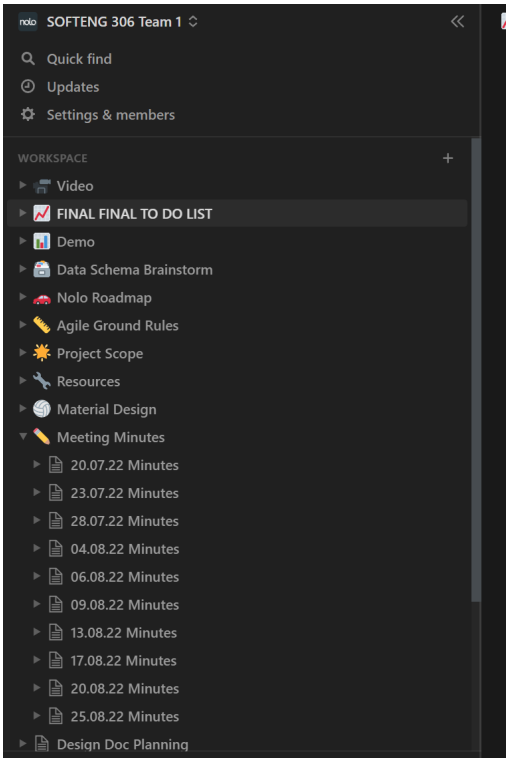


Figure 11. Notion

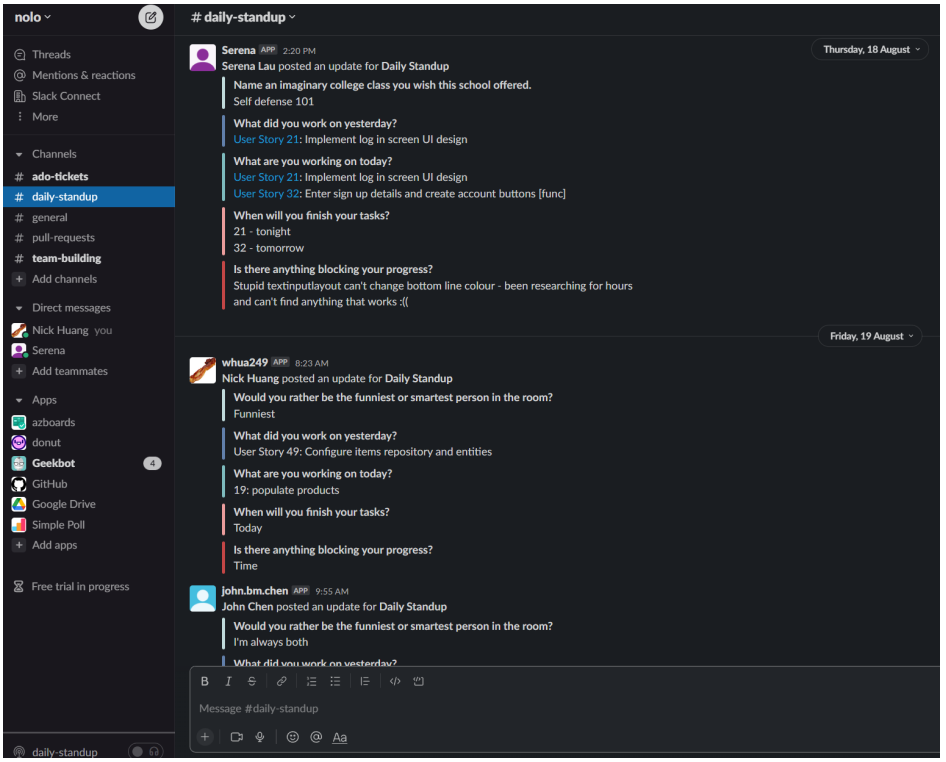


Figure 12. Slack daily standup channel.

2.4.3 Inconsistencies

In our design document, we constructed a Gantt chart to set up when we would like items completed by. It is found below.

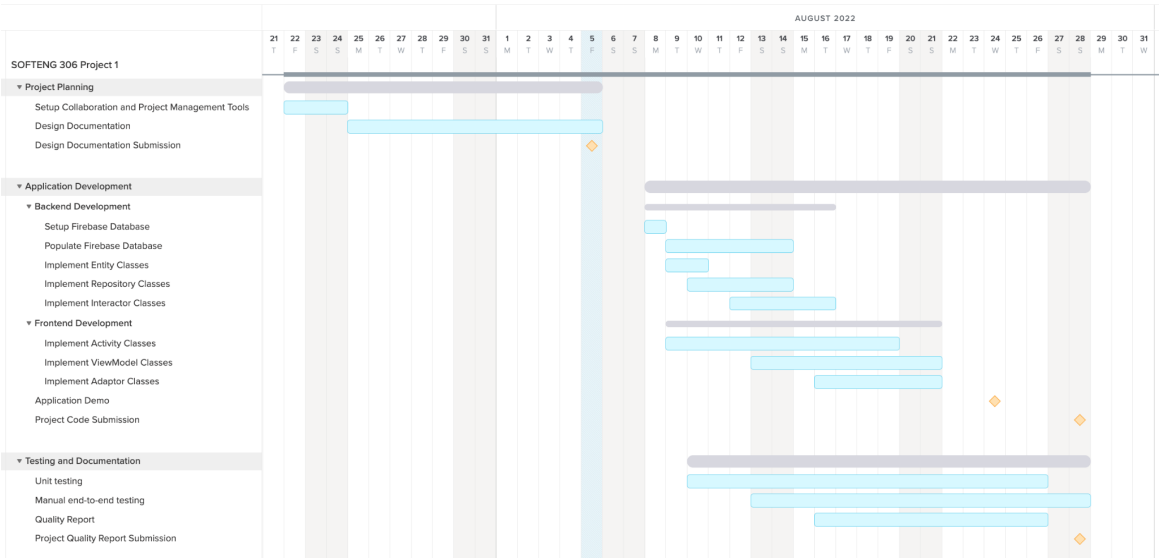


Figure 13. Design document Gantt chart.

We found that the realised implementation schedule stayed largely consistent with the Gantt chart. The order of tasks stayed the same but as the feature scope grew, deadlines were all pushed back slightly. Nonetheless, the team was proactive in reevaluating deadlines and ensuring a balance exists between meeting deadlines and personal wellbeing. We were able to finish the application before submission to a satisfactory standard.

In terms of inconsistencies, the team probably did not refer back to the Gantt chart as much as would have liked. The creation and tracking of tickets were often created with a general intuition of what we should work on next and where we should be if we are to be tracking well. In the future, to ensure greater consistency between our design document and realised schedule, we would like to ensure more corroboration with our Gantt chart is achieved. This can be perhaps done with some Azure DevOps or slack plugin for the Gantt chart which would have integrated well with our existing communication stack.

3.0 Functional Requirements

The functional requirements outlined in the brief are discussed in the following subsections. Additional features are briefly covered in section 3.5. Other aspects of significance such as responsiveness and transitions/animations are also considered.

When testing our application, we direct the marker to use the credentials shown in Section 3.9.

3.1 Main Activity

Our Main Activity is the Home Fragment fragment within our Main Activity.

It houses a recycler view which provides a selection of featured items. These items are selectable which navigates the user to Details Activity. If the user is a new user or has not interacted with any items under their account, the featured items are random “top picks” from the database. As the user begins to interact with the items (i.e., navigating to and from details activity), the featured panel shows the recently viewed items, in order of most recently viewed to least recently viewed, capped at a size of 5. Thus, as the user interacts with the application, the featured panel is updated, as required. The exact variant of the item (e.g., colour, customisation, branch, store) last seen is stored in the recycler view to promote optimal UX.

Three categories are provided. They are phones, laptops, and accessories. These are presented in a full screen snap scrolling list view. Tapping on each category takes the user to List Activity for the appropriate category.

3.2 List Activity

At least 10 items exist and are presented for each category in List Activity. Tapping on each item will take one to the corresponding Details Activity. The list activity is populated with the “default” variant of the item which is the variant at the cheapest location, with no customisations if applicable, and the first colour.

The data items are all populated by a ListView utilising a single adaptor. Dependency injection is shown in this adaptor where a list of IItems are passed into the adaptor and different views are rendered and configured based upon the class type of the IItem (Laptop, Phone, or Accessory). Each view is slightly different in the following sense. Their distinction allows us to demonstrate the concept of dependency injection.

- Laptop: laptops are grouped by brand and these groupings are sorted alphabetically. Brand groups are displayed in a child recycler view.
- Phone: phones are grouped based on os (apple or android) and these groupings are toggled on the main ListActivity. The ListView holds a custom staggered two column xml item for displaying phones.
- Accessory: accessories are simply listed in the ListView as wide xml items.

3.3 Details Activity

All detailed information for an item is shown in the Details Activity. This includes colour, images, name, price, branch, store, specifications, customisation options, and quantity to be ordered. There also exists an option to add this item to cart. Moreso, an option to view a map to select the most preferable branch to purchase the product from is available.

The activity is able to navigate through 3 images of the items by swiping on the view pager or using the bottom tab button toggle.

3.4 Search Activity

Users are able to search from both the Home Fragment (Main Activity in brief) and the Search Fragment.

Upon searching, dynamic search suggestions are provided with a custom list view. These suggestions are clickable which takes one to the appropriate Details Activity. Alternatively, one is able to search with the given suggestions which takes the user to the Results Activity.

Exceptions are handled appropriately. For example, if there is no search term, and the user tries to search, a toast is displayed informing the user no search term has been provided. If no items are found, a message is displayed in the results screen appropriately. Search functionality is intuitive and follows common conventions. E.g, the search icon changes to a cross to clear the input when the user focuses on the text entry field.

3.5 Additional Features

The team also added several additional features to the application. As these are not core functional requirements, we only provide a quick summary of each.

3.5.1 Splash Activity

The splash activity is displayed on app start up. During this time, all data is loaded into cache and a custom progress indicator shows the progress in this asynchronous action. Once all data is loaded and appropriate application permissions are checked, the user is taken to the login screen if not already logged in or straight to home fragment if otherwise.

3.5.2 LogIn/SignUp Activity

The login and signup activities are self explanatory in nature. They allow the user to log in and sign up using Firebase Auth. The sign up activity also permits the user to see a web hosted terms and conditions pdf through a web intent. Upon successful login or signup, the user is navigated to the home fragment.

3.5.3 Cart Fragment

The cart fragment allows users to see items placed in the cart. Users are able to increase or decrease the quantity of these items whilst viewing the items in cart. The user is also able to remove an item from the cart entirely. The quantity of items being ordered is displayed as a badge in the bottom navigation view cart icon.

3.5.4 Map Activity

The map activity can be navigated to from the Details Activity to assist users in selecting the most preferable location to order the selected item. Common map navigation is configured to assist with ease of use and to minimise cognitive load. This is done with the Google Maps package for android.

3.5.5 Profile Fragment

The profile fragment is the fourth tab which houses profile related information. This includes the ability to see historical purchases, users wishlist, and to manage account settings. These are expanded upon in the following sections.

3.5.6 Wishlist Fragment

The wishlist fragment allows users to view items in their wishlist. Users are able to remove these items from their wishlist from this fragment. Users are able to also add items to their wishlist from the Details Activity.

3.5.7 Purchases Fragment

The “purchases” fragment allows users to see a record of all historical fragments, and what state they are in (in transit or delivered).

3.5.8 Account Fragment

The “account” fragment allows users to view the email associated with their account, log out, as well as change their password.

3.5.9 ChangePassword Fragment

Provides users with the ability to change their password.

3.6 Responsiveness

Responsiveness has been ensured by testing our application on various devices. We often reference the util static methods to get the device width and height to programmatically set responsive styling to certain UI elements. This ensures that our app looks great on screens of all resolutions.

3.7 Transitions & Animations

Transitions and animations were employed throughout the application to support the look and feel of the application. They are as follows.

Transitions

- Fade transition. We set a fade transition to be the default transition when switching between activities if not otherwise specified as well as when switching between tabs.
- Horizontal slide transition. We use a horizontal transition between the sign up and log in activities.
- Vertical slide transition. We use a vertical slide up and down transition between activities.

Animations

- We employ a series of custom animations including:
    - Splash screen progress indicator when our cache is being loaded.
    - Snap scrolling of category list on Home Fragment
    - Animated expandable view in Details Activity for image carousel
    - Animated popup for Map traversal.

3.8 Misc

ViewHolders are properly employed across all activities, fragments, and adaptors to house their respective UI views.

All app data is supplied from Firestore.

Proper inheritance is shown, namely in the inheritance of Laptop, Phone, and Accessory classes from an Item abstract parent class.

3.9 Notes for Marker

We have noted that in testing that some emulators/devices fail to support the animations and/or communication with Firebase. However, the great majority (or varying build versions and support) do support our application. We have deduced this is not a result of the target/compile sdk or any other gradle property. We advise the marker to use another emulator in the case of unexpected build failures.

The maps API token is currently procured through a free 90-day trial licence so will cease to work in approximately 60 days from the submission day. We advise the marker to take this into consideration if indeed this is relevant.

For optimal performance, we advise the marker to use a physical Android device with both internet services and location permissions turned on.

We recommend that the marker uses the following credentials when logging in:

- Email: fraserisalegend@gmail.com
- Password: password1234

4.0 UX

4.1 Considerations

4.1.1 Motivation

General vision

- Our vision for the application is to be the ‘go-to’ app for technology shopping application
- For us, go-to entails:
    - Minimalistic clean UI
    - A high-end and sophisticated vibe so users would feel good using and looking at it - we want users to feel as if they’re going through a magazine
    - Efficient and smooth UX
- Due to the nature of selling technology, we also wanted to make sure it was modern, so we combined all of these factors to curate a sophisticated, elegant, modern GUI design
- Modern and elegant can often be quite conflicting design perspectives, so it took a lot of careful consideration and selection of design elements to effectively portray a union of the two moods.

Common design components

- The use of both sharp and rounded edges were carefully selected to produce the optimal amalgamation of modern and elegant, while ensuring it didn’t create a clash of style.
- The sharp edges are used only for the item ‘picture-frames’ which emphasise the high-end character, portraying the items in a sophisticated manner.
- Round edges are used across the rest of the app e.g. buttons, icons, tags
    - Smoother and easier on the user’s eyes, and follows the trend of modern application designs
- Important or header text (excluding the artistic screen headers) are always in upper case to create a more formal, rather than light-hearted or casual, atmosphere to the app.

- **Colours:**
    - The app’s primary colours are navy, white and light brown. These colours are all selected from the recommended Material Design palettes and are chosen to complement each other in harmony with the app’s design vision.
    - Navy as the base of the app creates the dark atmosphere of a modern technology app, while the brightly-contrasting white ensures content is effectively displayed on the app. Light brown is used as an accent colour to bring out important buttons, features, and app state; this colour’s softness adds to the app’s elegance and sophistication.

Screen headers

- Each main screen (home, search, cart, profile) have a large artistic header to fit with the magazine vibe
- Branching off the profile screen, purchases, wishlist and account also have an artistic header - but smaller to indicate it’s a secondary screen and identical to show their relationship as profile elements

Categories list

- Each category has its own full-screen title page, giving the user the impression of flicking through a magazine
- Index indicator used to help users feel in control of their navigation and visibility of the system status, especially with the use of the full-screen
    - Also snap scroll to help users efficiently navigate through the categories

4.1.2 Conformity to Material Design

We ensure that all design elements conform well to material design guidelines. These were namely.

- All font sizes are in sp and conform to specific recommended values
- Colours are from the suggested colour palettes defined
- Dimensions are all in dp.
- Touch inputs must be at least 48dp in both dimensions
- There is 16dp for margin/padding around text content when placed against the screen border

4.2 GUI

Here is an overview of the Figma designs we used to realise our implementation. As one can see, our implementation matches closely with these designs.

[Here is an online walkthrough of our figma prototype.]

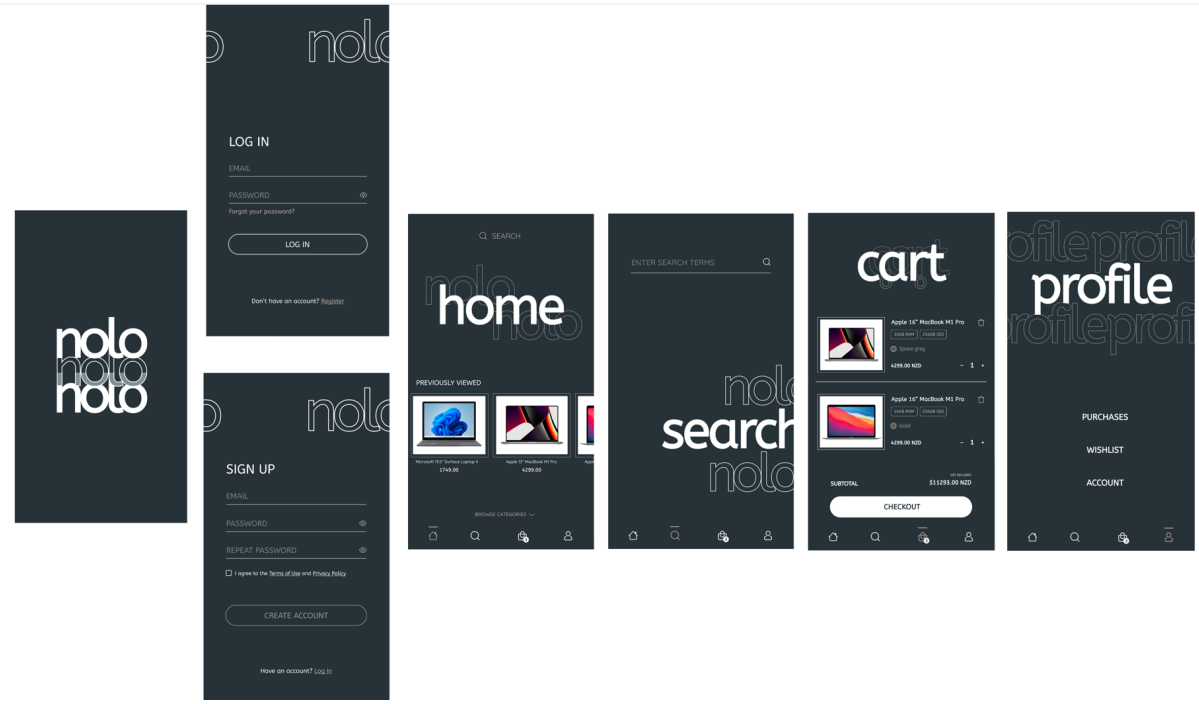


Figure 14. Core Flow GUI (Splash -> User Entry -> Home, Search, Cart, Profile)

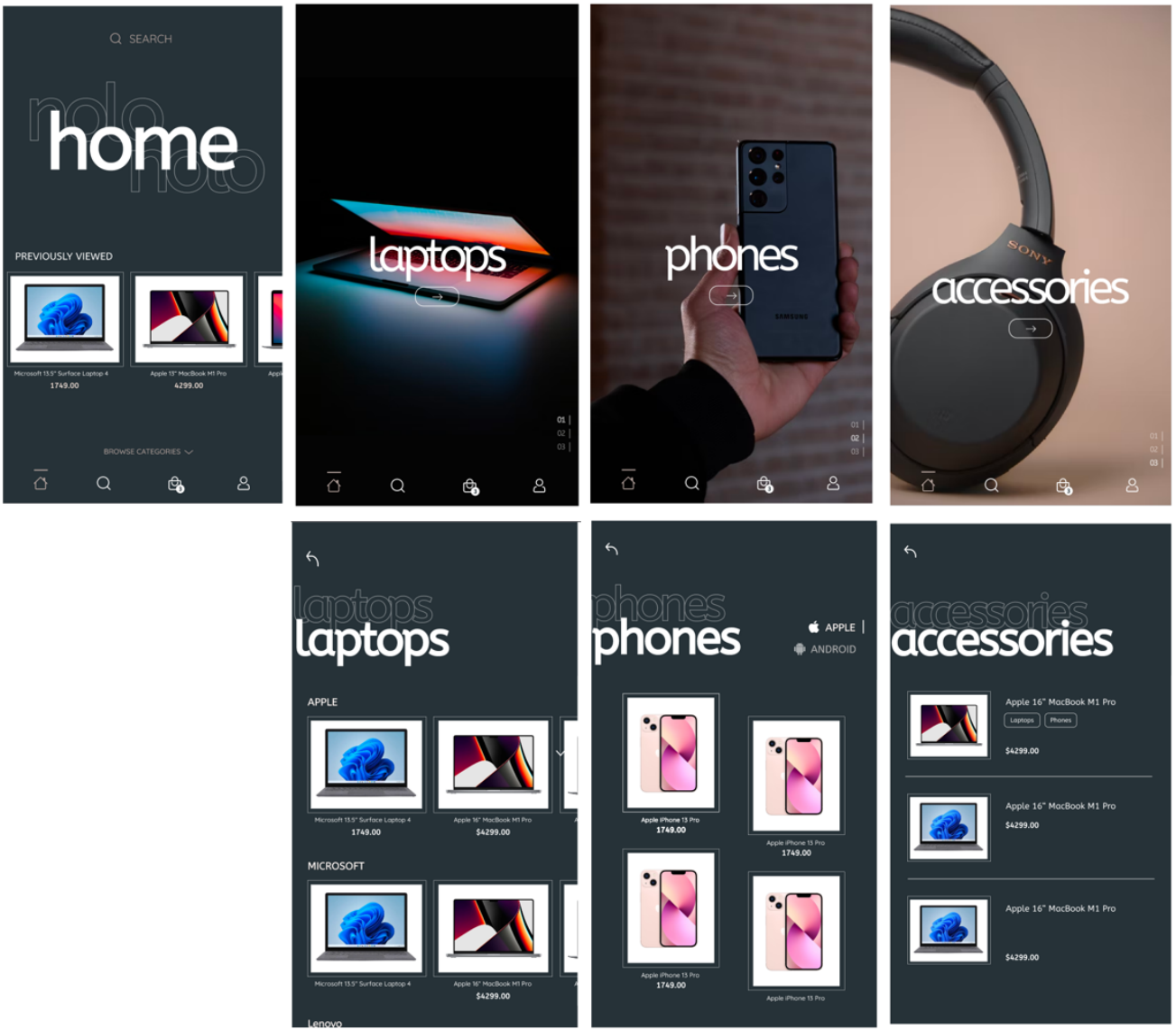


Figure 15. List Flow GUI (Laptops/Phones/Accessories List Activity)

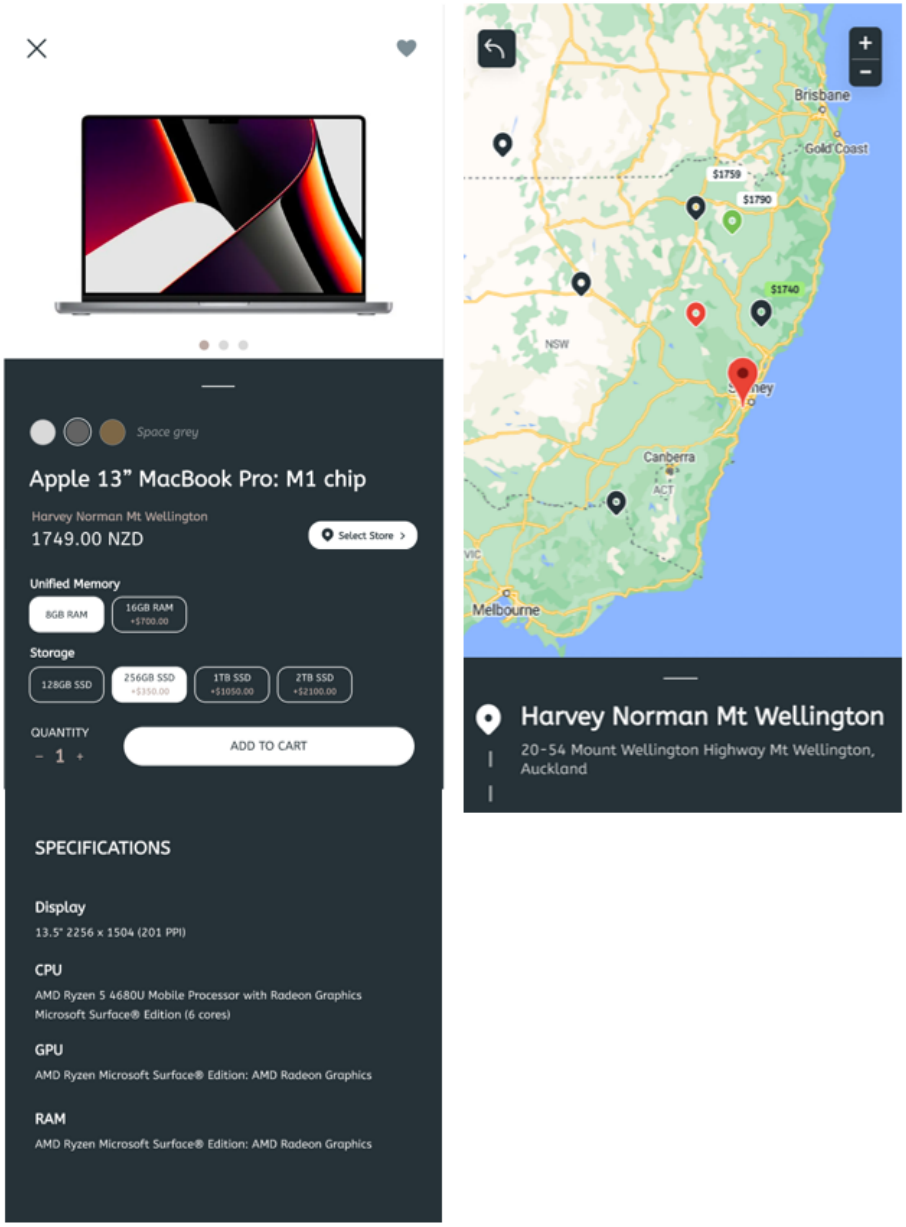


Figure 16. Details Activity and Map Activity

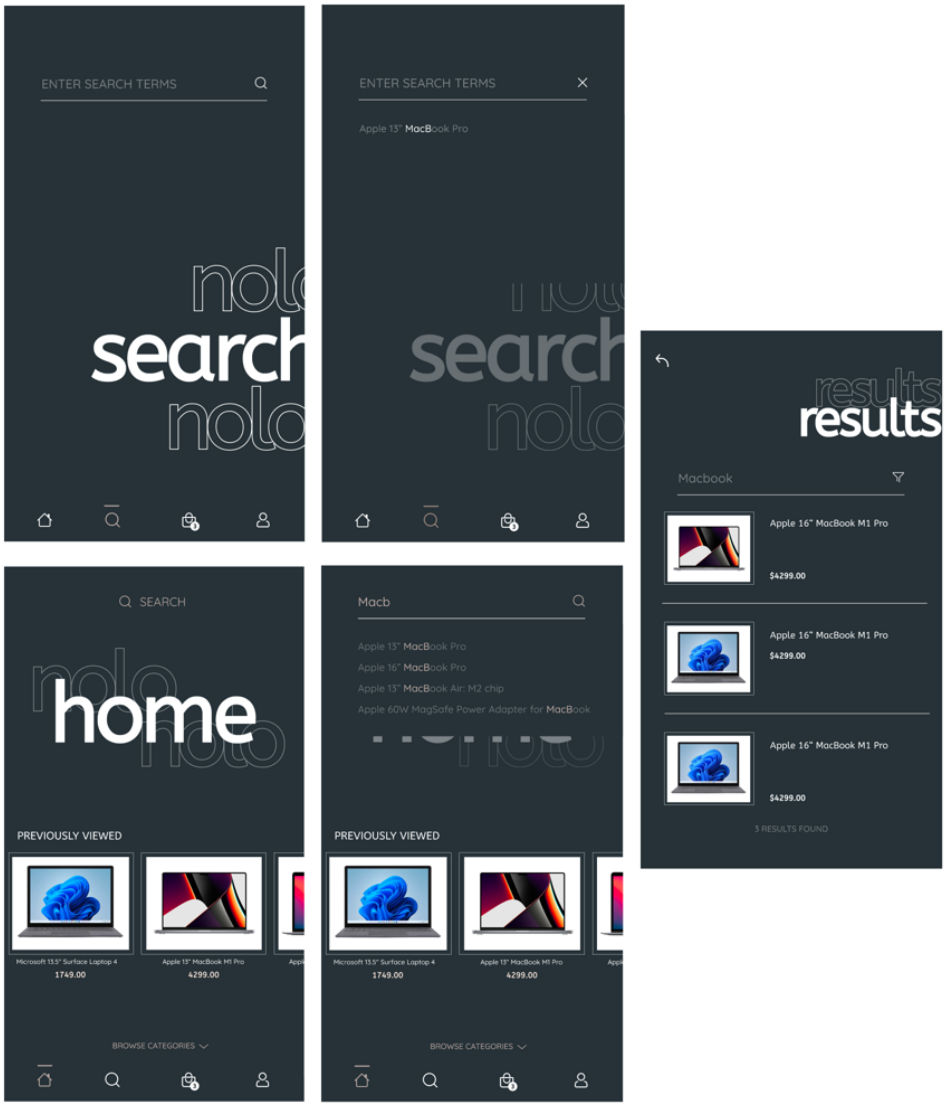


Figure 17. Search Item Flow GUI

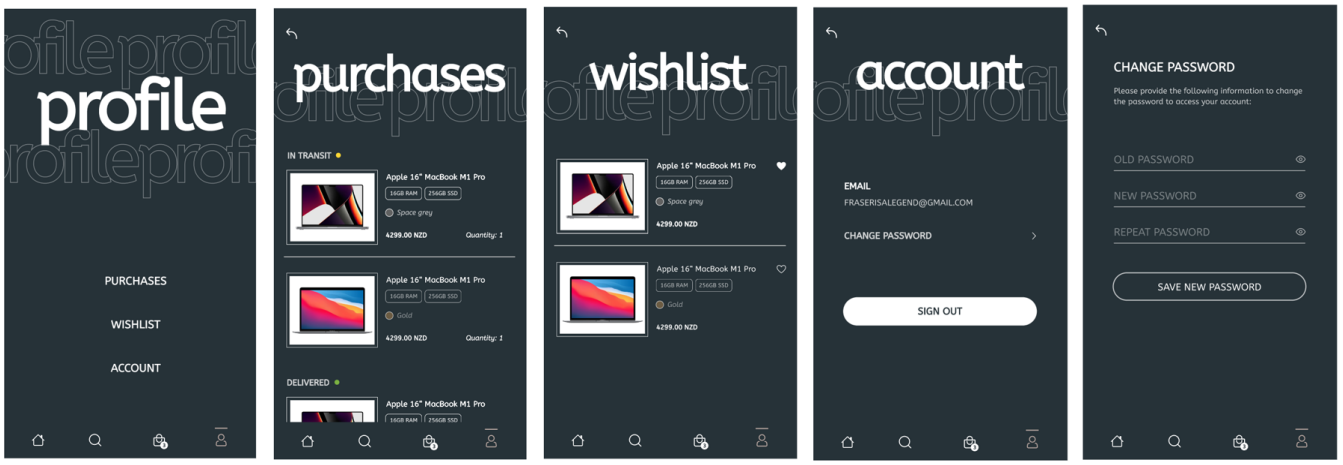


Figure 18. Profile flow GU

4.3 Design Doc GUI Designs

We provide the GUI designs that we had in our design document below. This aids the discussion on inconsistencies in Section 4.4.

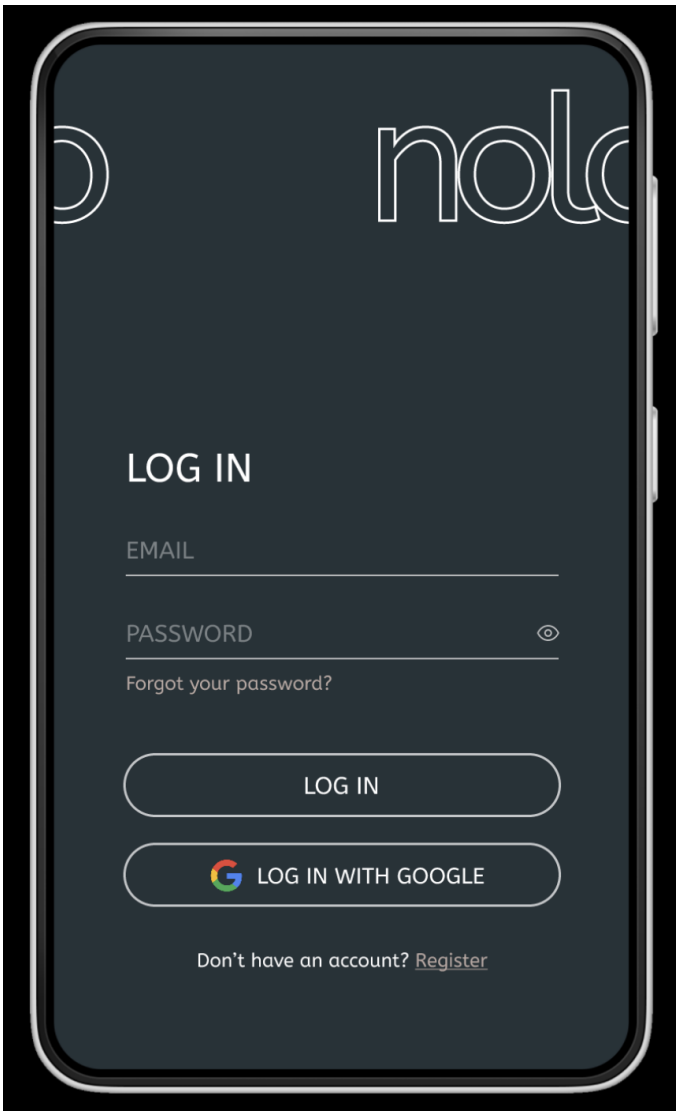
Splash Screen

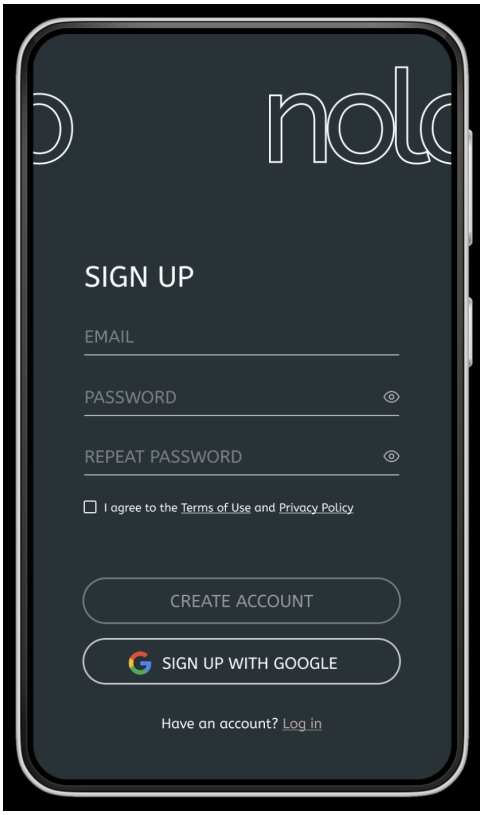
This screen is displayed when the user first opens up the application, before the login page is automatically loaded up.

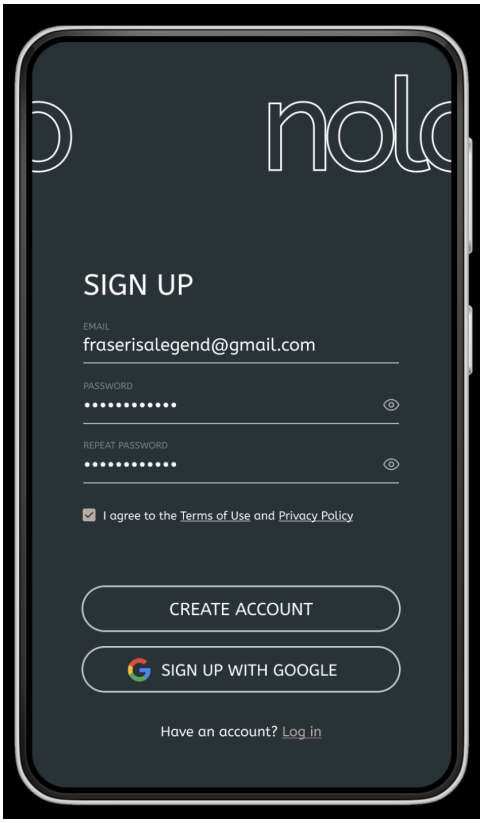
Login and Signup Screens

If the user is not currently logged into the app, the login screen (left) is the first interactive screen presented to the user when opening up the app. Otherwise, if the user is already logged into the app, this screen will not be presented to the user.

If the user desires to create a new account, they can click the ‘Register’ button from the login screen to navigate to the signup screen (right) to create a new account.



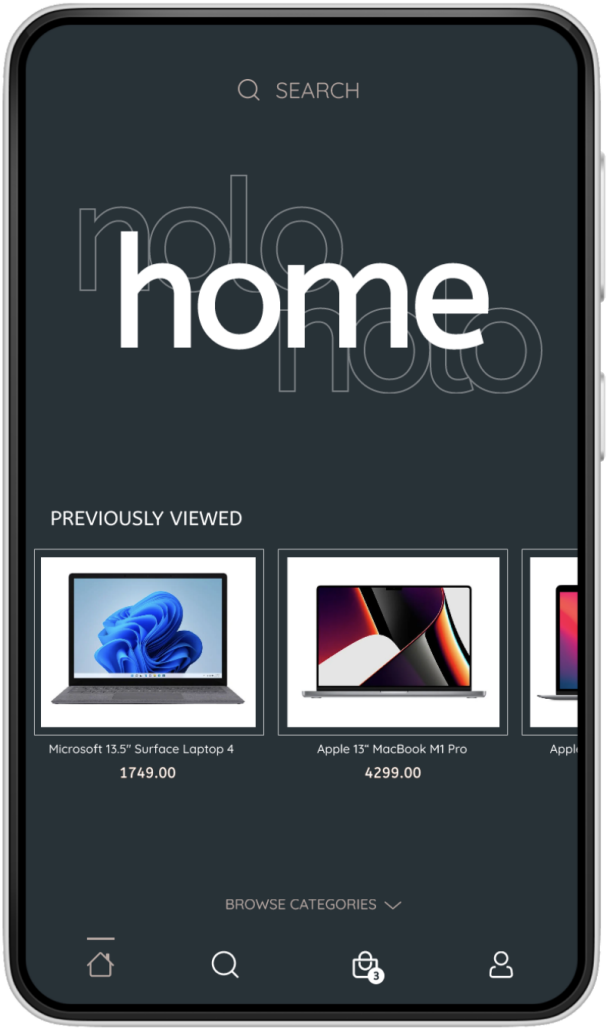




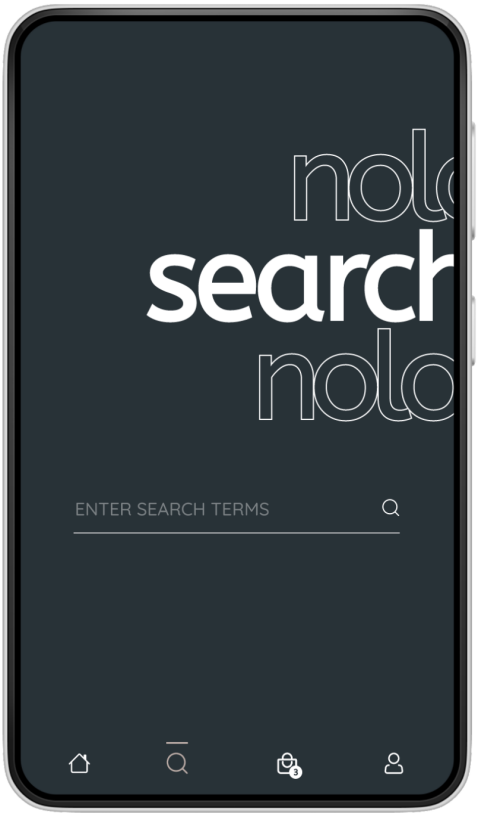
Home screen

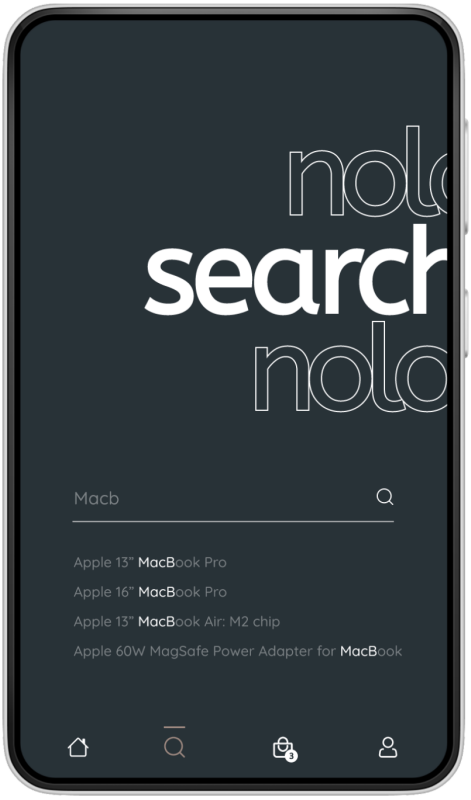
This is the first screen displayed to the user after login, or on opening the app if the user is already logged in.

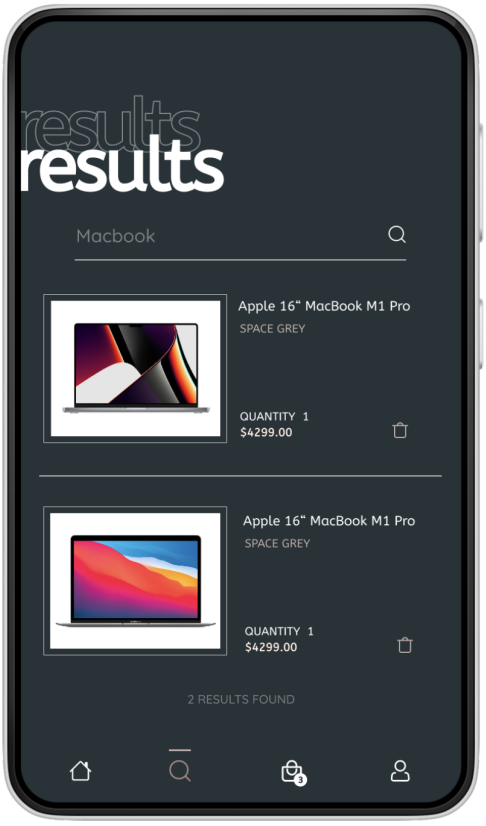
Scrolling down will reveal the category lists one after the other (stacked vertically).



Search (top) and Search Results (bottom) screen

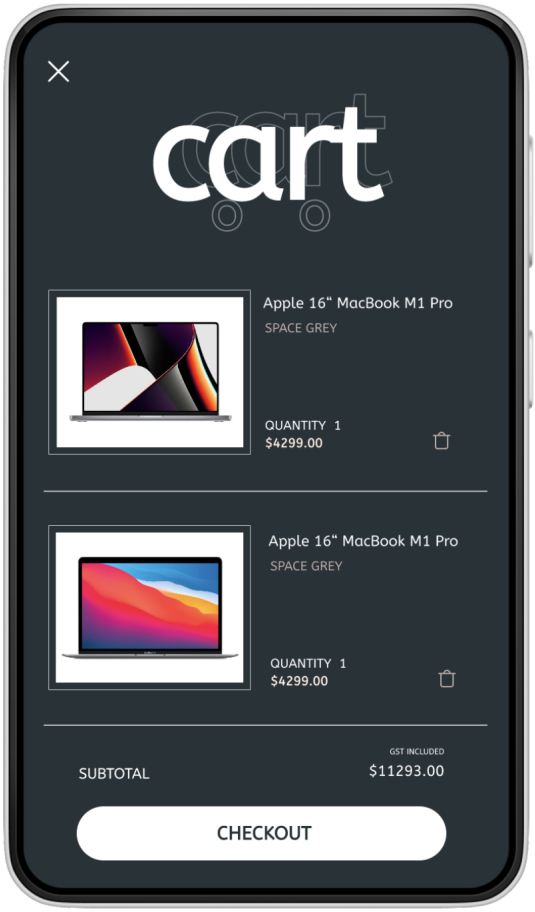


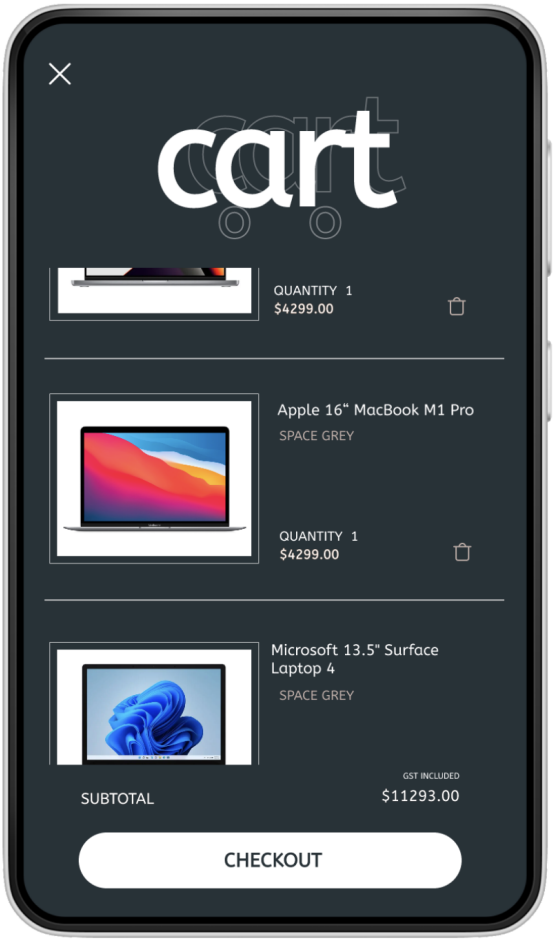




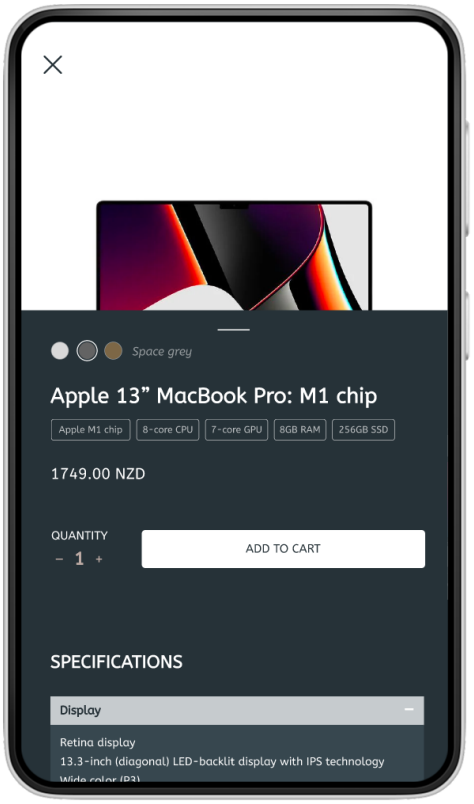
Cart screen

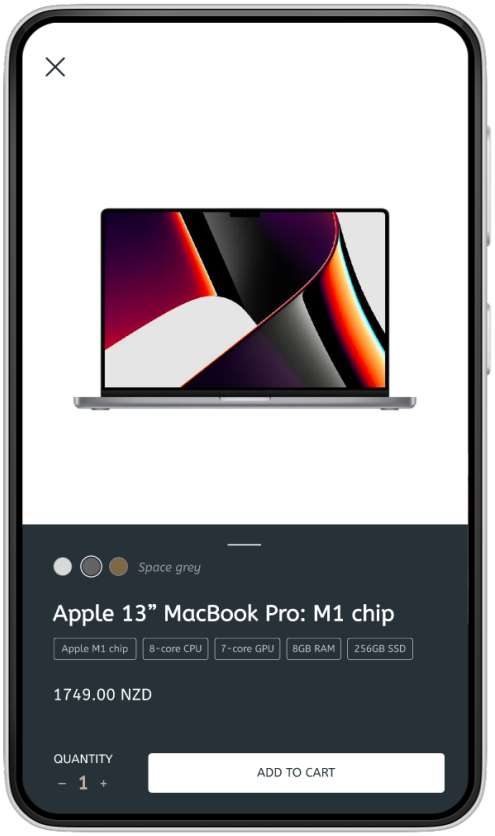
This screen opens up as an overlay over the user’s current screen.

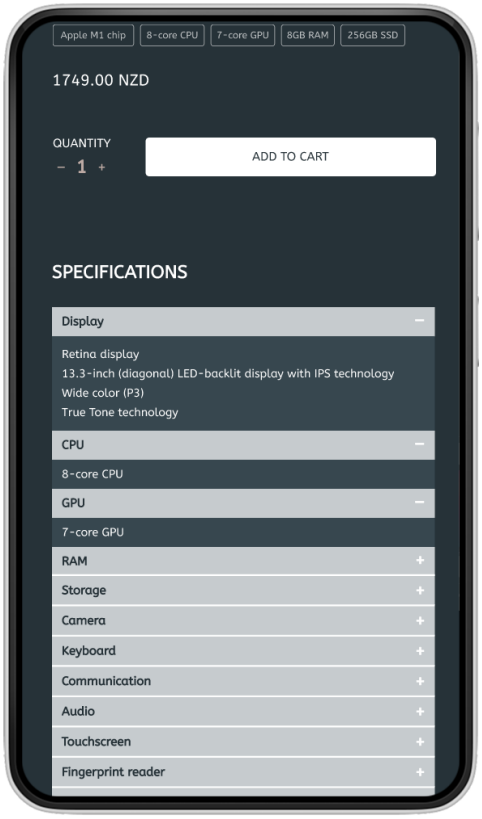




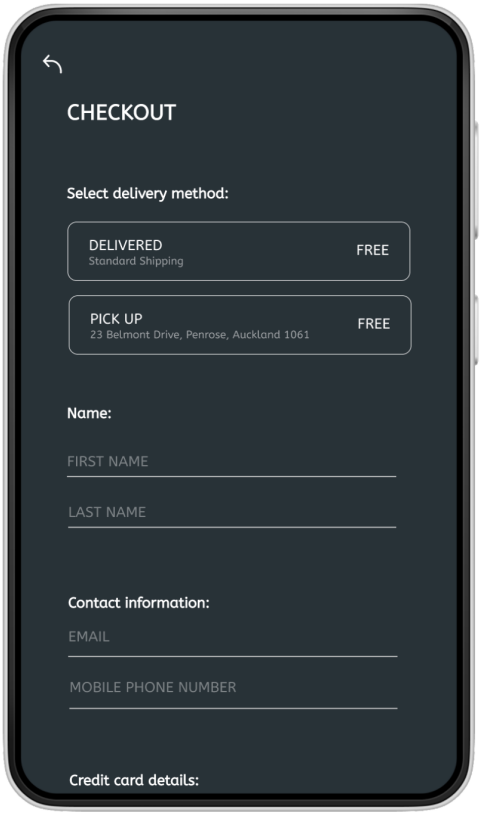
Detailed screen

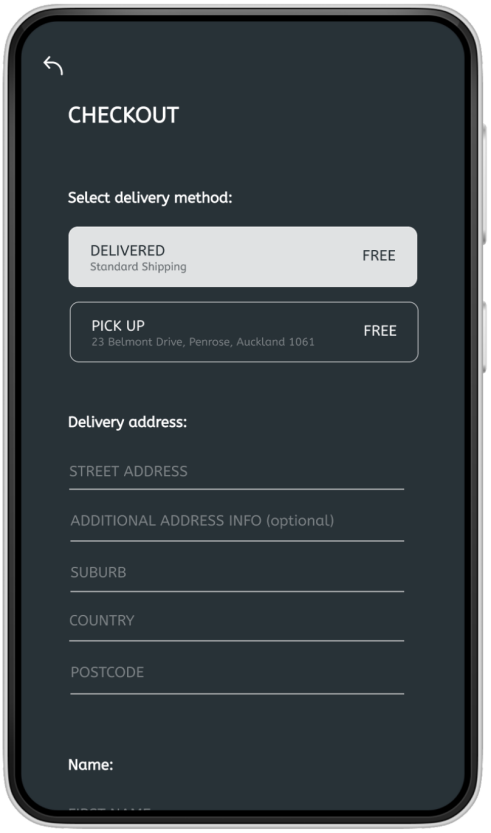


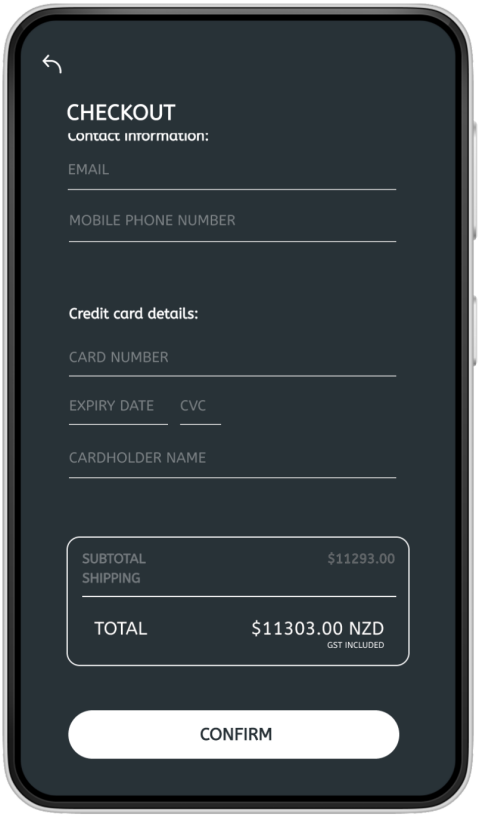




Checkout screen







Profile screen

4.4. Inconsistencies

We can see that, between our design document and realised GUI designs, and thus by transitivity between our design document and implemented GUI, design has stayed largely consistent.

There are however some notable changes which we outline here.

- Wishlist + Map + Historical Purchases
    - Wishlist, historical purchases and map features were added to the scope after the design document was submitted which therefore added more GUI designs and changes to Details Activity and Profile GUI.
- User settings
    - User settings such as changing password as well as viewing your account email was added which warranted changes and additions to design.
- Search modify
    - The Search Fragment had some layout changes after the team realised that the onscreen keyboard would cover the search suggestions if the search field was positioned near the bottom of the screen as we did prior.
- Sign up with google
    - The sign up with google button was removed as we found this functionality too much work for too little benefit in the scope of what we wished to achieve in the timeframe we were given.
- Checkout
    - Checkout fragment was removed from the initial design for a similar reason to that of “Sign up with google”. Ideally these missing features would have been implemented given more time.

5.0 Acknowledgements

We wish to take the time to thank many of those who have given feedback/input into our work with nolo.

- Reza Shahamiri. For teaching us on SOLID design principles, good code practices, identifying code smells, and Android app development. For providing quality feedback in our presentation demo.
- Fraser McCallum + Osama Kashif. For assisting us with project queries. For providing quality feedback in our presentation demo.
- Ivan Ko. For feedback and inspiration regarding the application's visual design.
- Khoda Sisodia. For assisting with our understanding of system design and SOLID principles.

6.0 References

[1] Why we need Solid Principles and it's types. (n.d.). Retrieved August 2, 2022, from https://blog.knoldus.com/why-we-need-solid-principles-and-its-types

[2] Guide to app architecture. (n.d.). Android Developers. Retrieved August 1, 2022, from https://developer.android.com/topic/architecture