Software Design and Architecture

Atec Mart - Online Shopping Store

Project Report

Hassan Raza Bukhari (247486)
Salman Inayat (263202)
Muhammad Umer Farooq (266086)

23rd January, 2021

Architecture

MVC

Model View Controller (MVC) architecture was used for the Atec Mart online application. It is an architecture quite popular for creating web and android applications which need to be scalable, or those applications which are already large. This architecture has three parts:

1. Model
2. View
3. Controller

Model is the part of the architecture which contains all the data related components like databases, etc. It is an essential part of the application from where data is fetched from, saved to and interacted with.

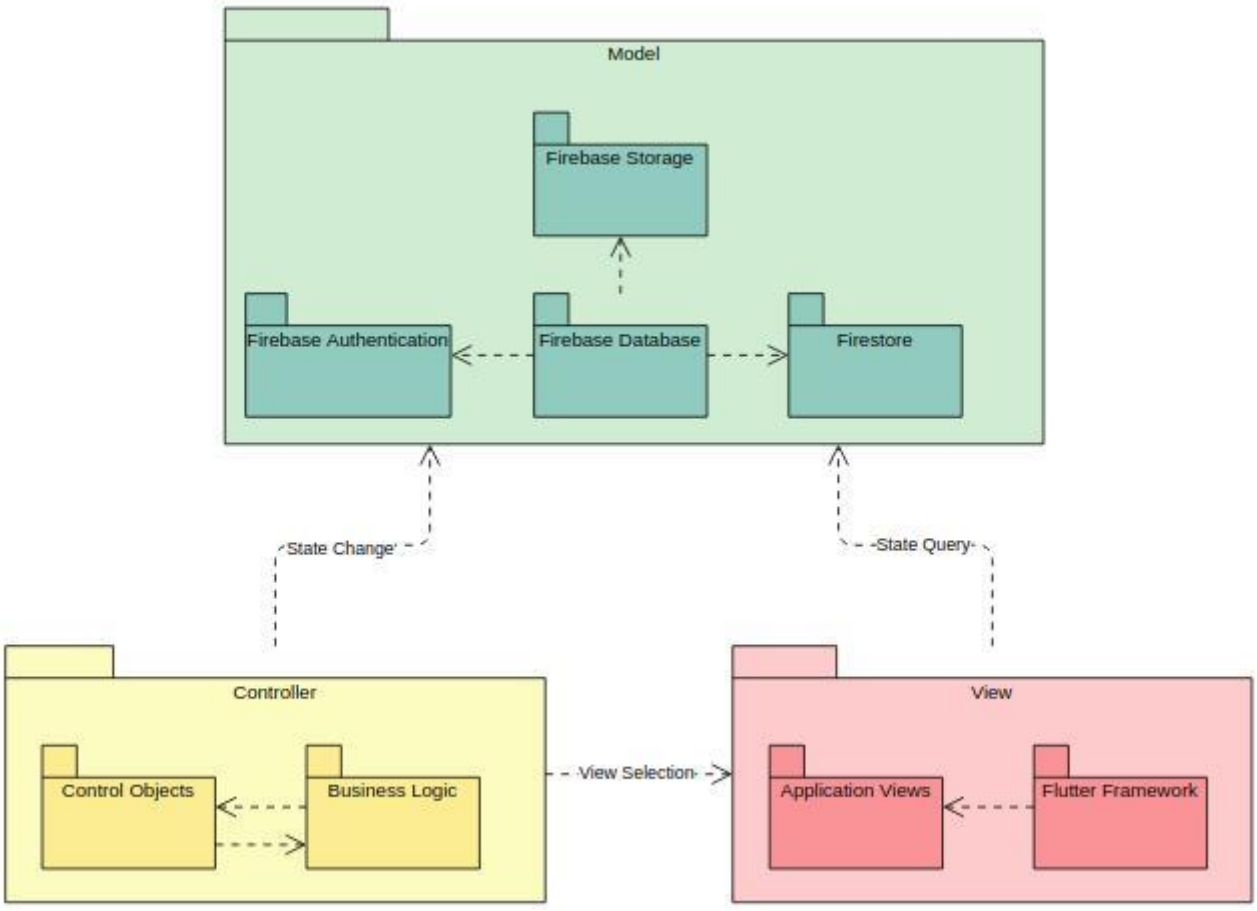
For example, a student object will retrieve grade information from the database or update current information.

View is the component of the system which contains the components responsible for the User Interface (UI) of the application. It is the part where a user interacts with the system.

Controller is the component connecting both the model and the view. It acts as a translator of requests. This part contains most of the back end logic of the application.

The reasons why MVC architecture was chosen for this application are as follows:

- Easy and fast deployment
- Collaborative development (multiple developers working on different components) is easier
- Relatively easier to debug
- Based on research on internet found that it is best for complex but lightweight applications



Requirements

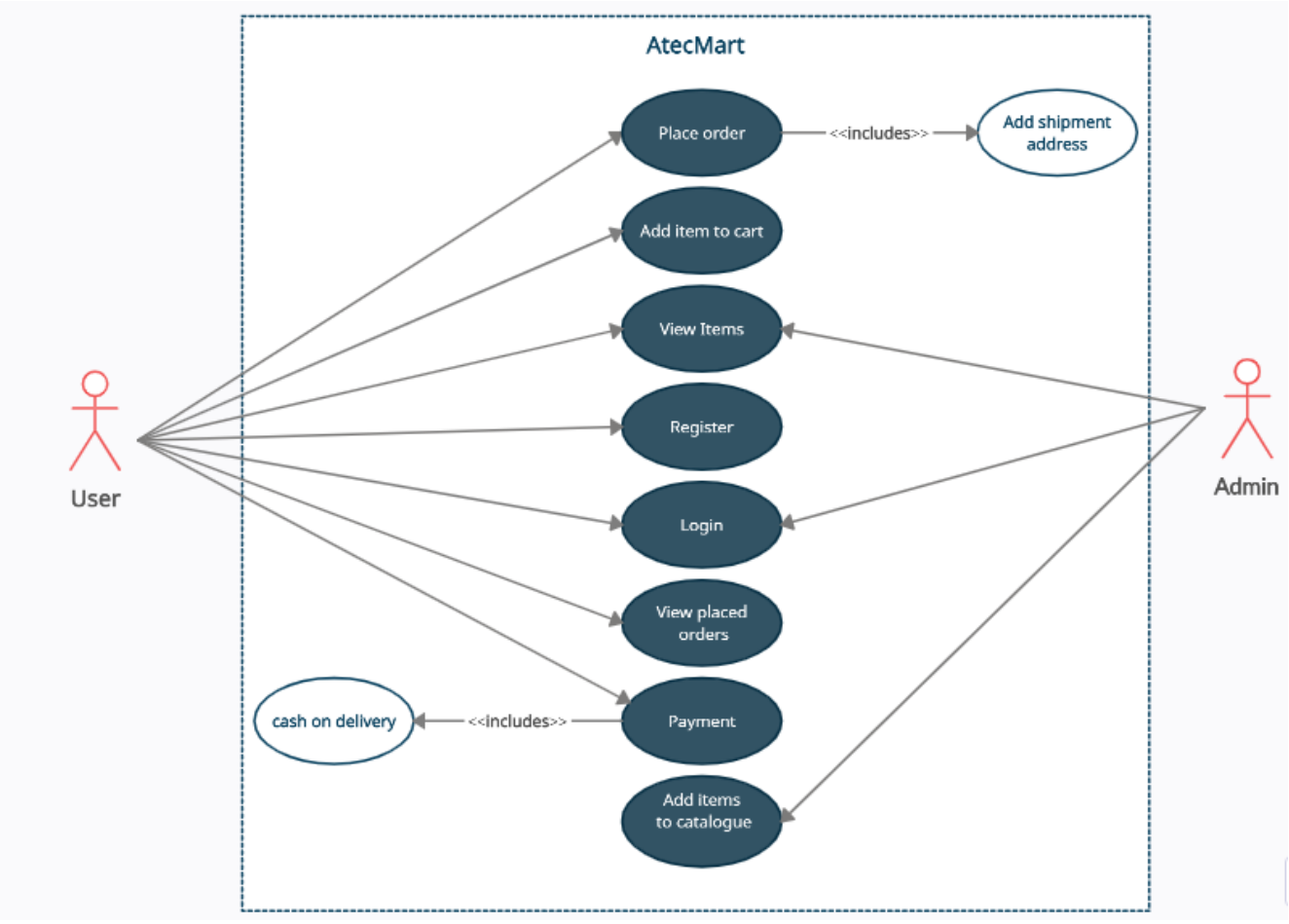
Functional

1. System shall allow a new user to register an account by providing information including their display picture and name that is stored in the database
2. System shall have an authentication system to log the user and admins in.
3. System shall allow the user to add items to the cart
4. System shall have the functionality to show item details on the item page when user clicks on an item from the main page.
5. System shall allow the user to remove an item from the cart.
6. System shall have functionality to allow user to enter their shipment address.
7. System shall have functionality to allow user to place an order after checking out items in the cart.
8. System shall have functionality for admin to add new items to the catalogue by providing necessary details.
9. System shall have functionality for user and admin to log out of the system.

10. System shall display the items present in the database in the form of a catalogue
11. System shall maintain user, order and shipping details

Non functional

1. System shall be able to support 10,000 authentications per month
2. System shall support cloud storage of 1 GiB (1.074 GB)
3. System server shall have network egress of up to 10GiB/month.
4. System shall have the server capacity that allows 20,000 document writes per day, 50,000 document reads per day and 20,000 document deletes per day.
5. System server shall support up to 40,000 CPU-seconds per month.
6. System shall ensure data integrity and security of customer data
7. System's database server shall be available ideally 99% of the time

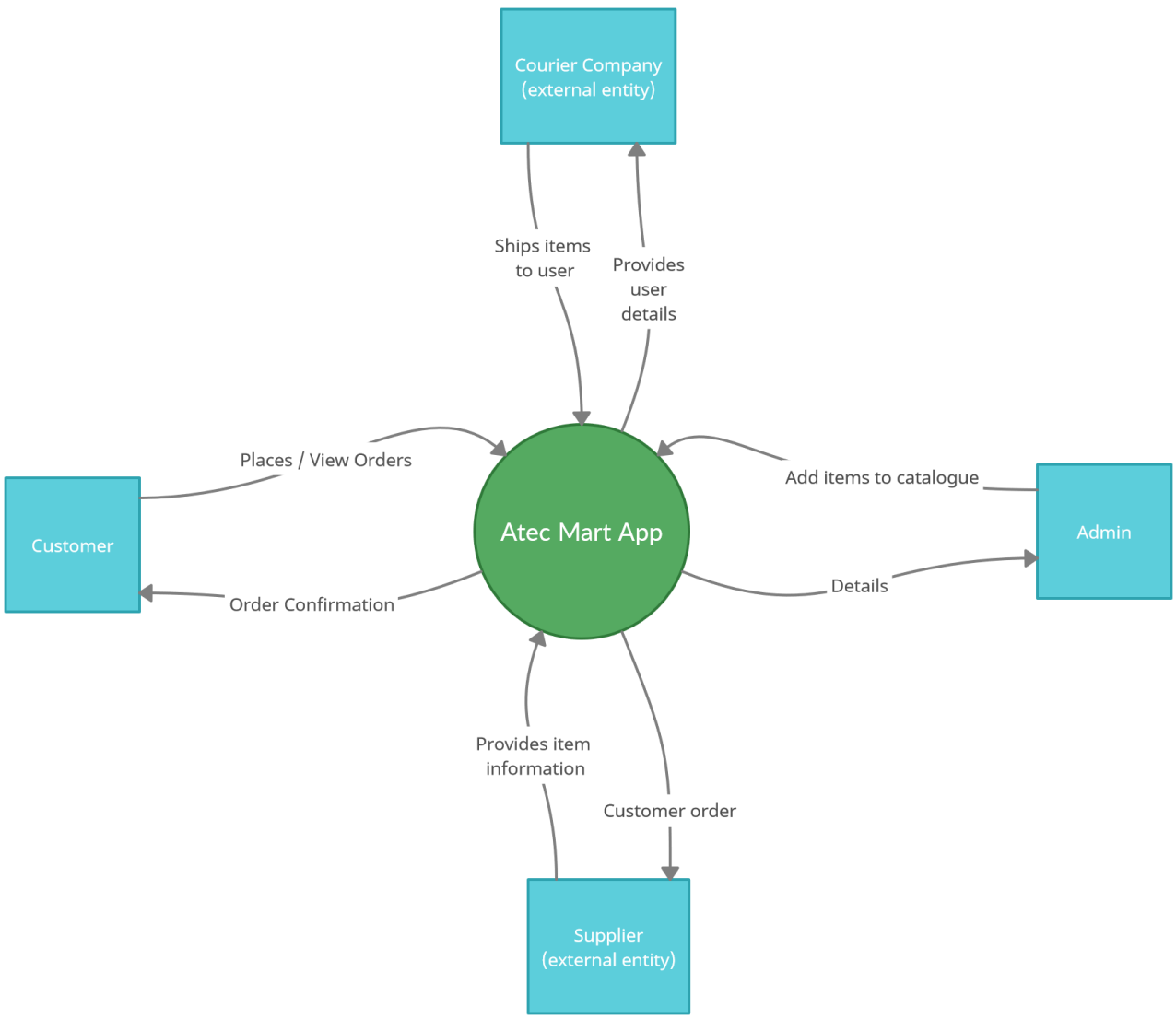


Fulfillment of Non-functional requirements

The non-functional requirements were decided keeping in mind the smooth working of the application for a small number of users, and it can be increased based on needs later on by increasing the

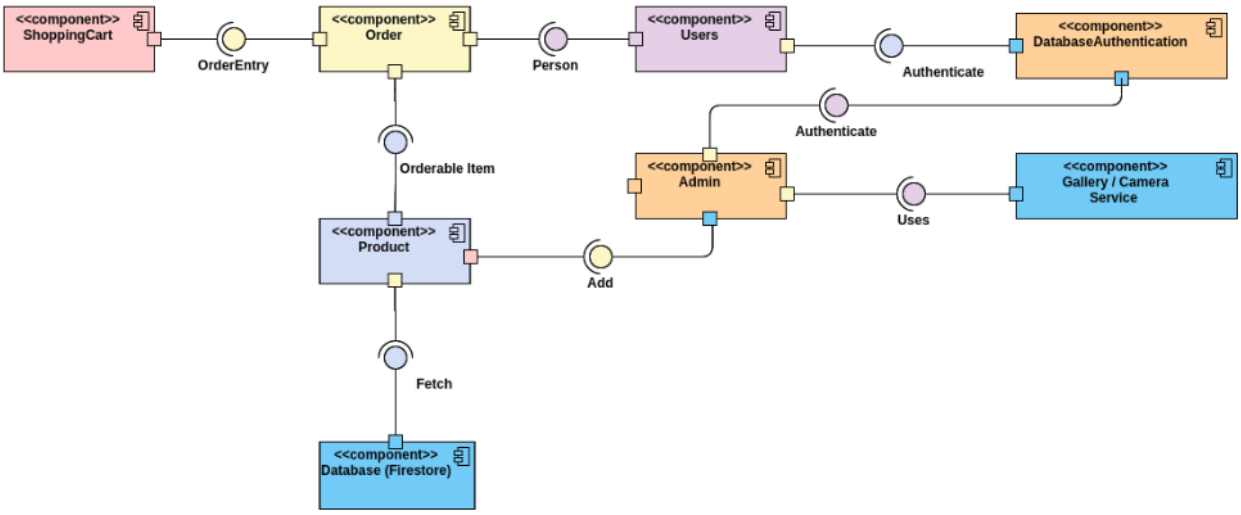
specifications of the server. Currently, the application's server is the Firestore cloud server provided by Firebase which provides all of the mentioned functionality, thus fulfilling all the mentioned non-functional requirements. In the project proposal, very basic non-functional requirements were mentioned, but as the understanding matured while developing and getting hands on with the application, the non-functional requirements were refined and made more specific.

Context Diagram



Identification of all components of the system

All the components of the system were identified using a Component diagram.



Different component interaction within component diagram is as follow:

1. **User** before accessing the **product** page is authorized through database **authentication** using email and password.
2. **User** can view the **product** on **home screen** fetch through Firestore **database.**
3. **User** can add the **product** to **cart** and proceed to checkout moreover, **user** can view the detail of the **orders.**
4. **Admin** can add the **product** and its respective details which will store partially on **Firebase** storage and partially on **Firestore**.
5. **Admin** can add **product** image with **Camera service** or through **Gallery service** of the device.
6. Within **Homepage** logic is implemented to fetch **product** detail from **Firestore**.

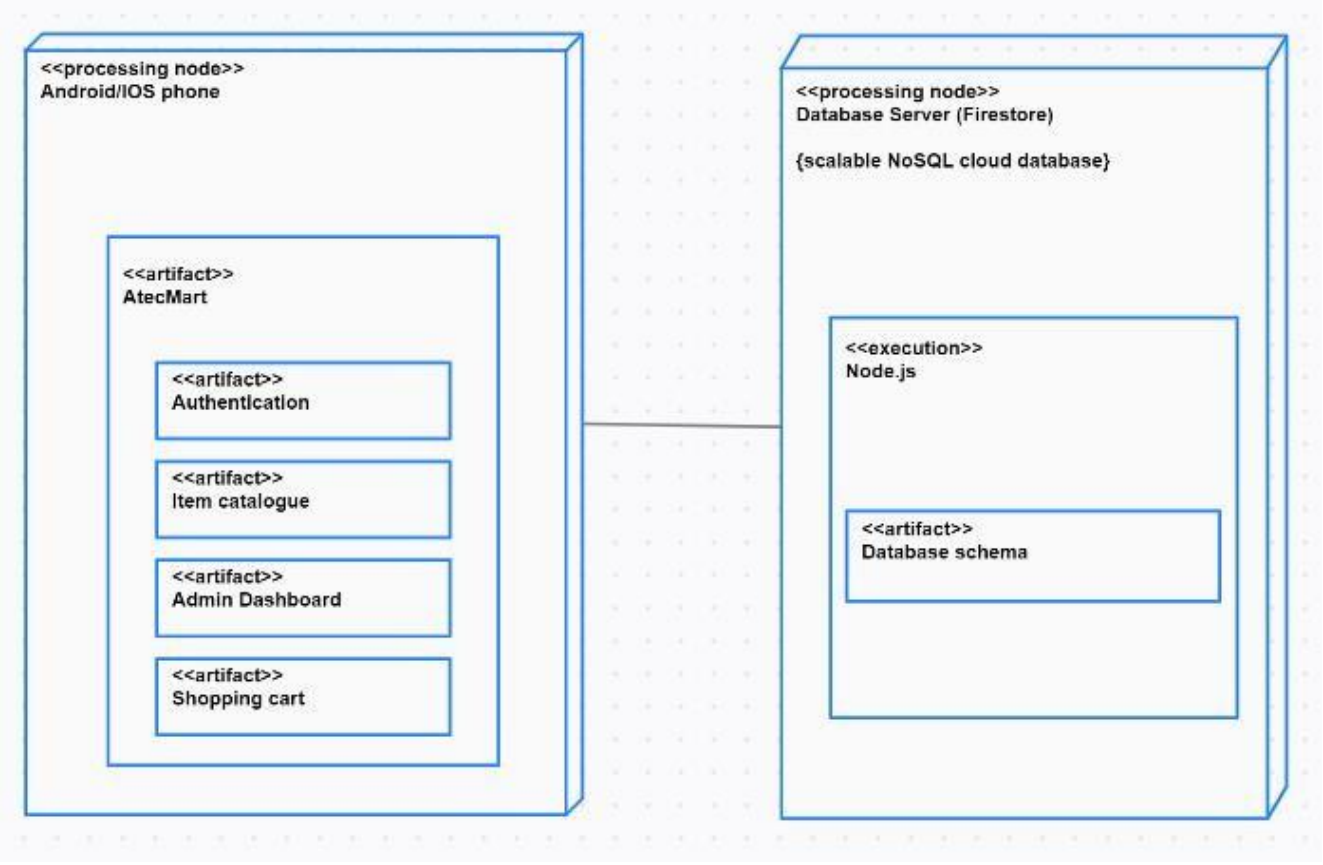
Physical Representation of System

The physical representation of the system can be given by using the deployment diagram. It gives an idea about which physical processing nodes actually make up the system.

A deployment diagram describes the physical hardware on which the software is to be executed. It maps software pieces of a system to the device which are going to execute them. In implemented online shopping store, there are 2 processing nodes which represent computational resources on which the system is running. These two processing nodes are client's phone and a database server, which is Firebase in our case. A client processing node runs on either android or IOS operating system. While a firebase database node uses Node.js as its execution environment. The deployment diagram also specifies artifacts which represent software elements of the system. In AtecMart application, Firebase

processing node contains Database schema as artifact. While client node contains the following artifacts:

- Authentication
- Item Catalogue
- Admin Dashboard
- Shopping Cart



Design

The discussion on design can be started by looking at the design patterns that were used in the Atec Mart application and then the discussion can be developed by bringing into account the implementation details that a junior programmer can use to implement some subset of the functionality.

Key Patterns

Based on research, the most common design patterns used for MVC architecture are behavioral patterns, therefore they constitute an important part of the application. Although, the application is not limited to just behavioral patterns only. Since the application is divided into several components, therefore different components make use of different design patterns.

One pattern being used is the Facade pattern. Whenever the Firebase API is accessed to retrieve, update, delete information from the database. A facade interface is used to perform this action, this facade is in fact provided by Flutter classes for Firebase. The application is not concerned with how the memory is managed on the server, and how the actual retrieval, updating is done on the database, but rather it is concerned with the interaction with the Facade interface and the result it provides, which is usually just an indication of success or failure. The rationale for the use of this pattern is that the component accessing the information from database, does not require the implementation details for the actual functionality of retrieval, it is common practice for retrieval from all the different databases.

One other important pattern is the Command pattern. It is being used in most of the important parts of the application. A user object issues a command by clicking a button on the UI, for example the home page, the application logic then executes the relevant functionality and fulfills the command. All the details of how the task was done is encapsulated from the user object, it just issues the command and receives the results based on that. It is not concerned with how the actual execution is performed or how it was implemented. The rationale for using this pattern is that it is a very suitable and intuitive approach to address the interaction of a user with the user interface in any application.

The observer pattern is also being used in the application such that whenever an admin updates the details of a product, the changes are automatically reflected at all the pages where products information is available for example, on the home page, on the products page, and on the cart page as well. The rationale for this pattern is the ease of updating all the different instances where a product's information is being used. It makes the task easier to perform.

Description of Classes

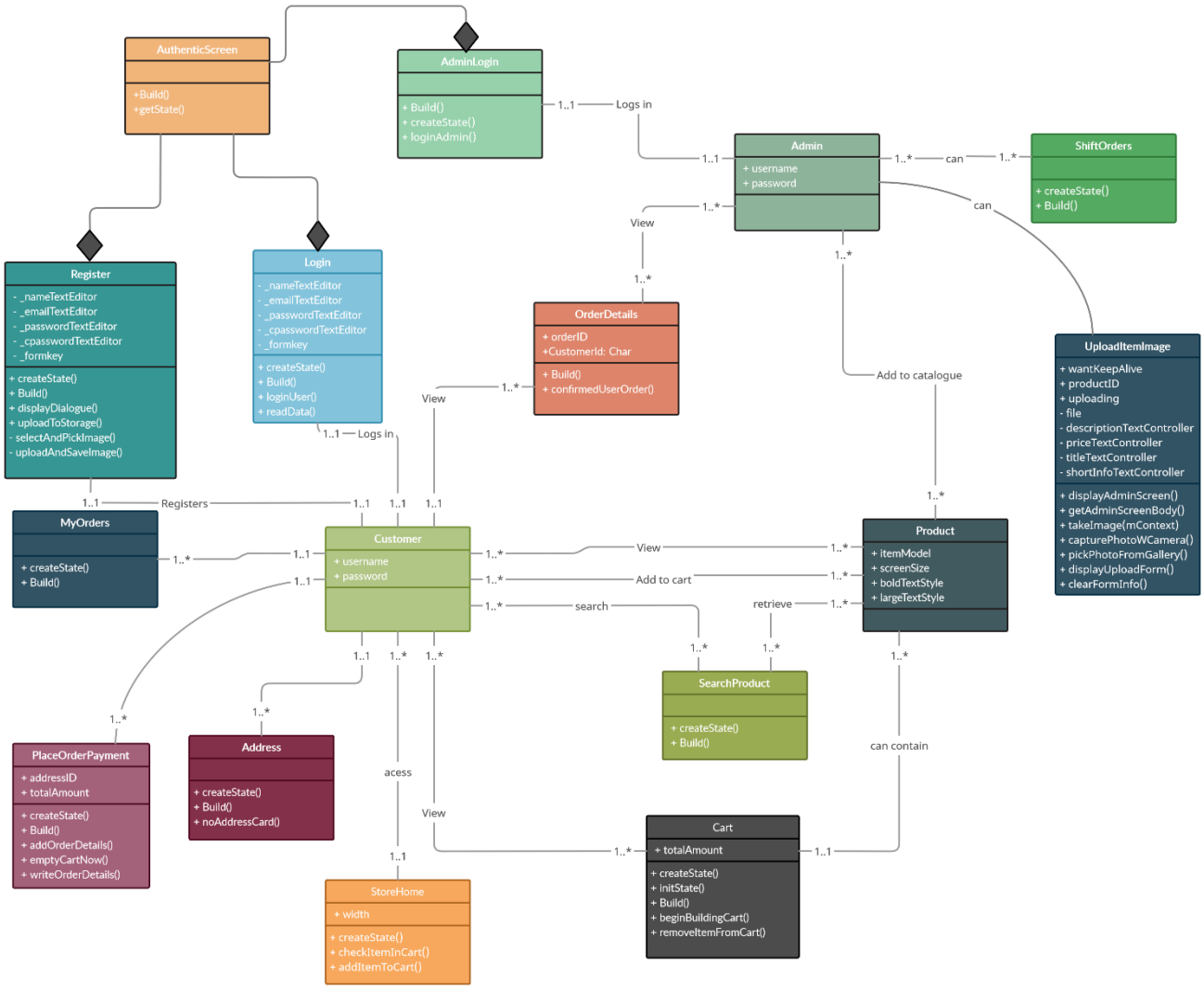
Relationships between classes are as follow

1. **AuthenticScreen** class Provides view for **register**, **login** and **AdminLogin** class as there is a composite relation among these classes.

2. **Customer** class allow user to register through **Register** class by providing valid email, name, password and image field.
3. **Register** class uploads the text fields on Firestore through **uploadToStorage()** method and image field on firebase storage through **uploadAndSaveImage** method.
4. **Login** class authorizes the **user** by using **readData()** method and tracks number of users through **loginUser()** method.
5. **OrderDetails** class shows details of order placed by logged in **user** through **confirmedUserOrder()** method.
6. **Admin** gets authorization through **loginAdmin()** method by providing username and password.
7. **Address** class allows **Customer** class to add shipping address through **noAddressCard()** method.
8. **MyOrders** class allows **Customer** class to track orders placed through **Build()** method.
9. Through **Customer** class user can add multiple items in the cart through **addItemToCart()** method and get notified through **checkItemInCart()** method of **StoreHome** class if the item is already in cart.
10. **PlaceOrderPayment** class allows **Customer** class to empty cart through **emprtyCartNow()** method.
11. **Customer** can search product through **Build()** method of **SearchProduct** class.
12. **OrderDetail** class shows the details of the order to **Customer** class through **ConfirmedUserOrder()** method.
13. **UploadItemImage** class allows the **admin** class to upload **product** by providing.

    - Product Id
    - Image of Product
    - Title
    - Short Information
    - Price
    - Description

14. **Admin** class views details of confirmed **orders** through **ConfirmedUserOrder**() method in **OrderDetail** class.



Mapping of Design entities with Architectural Components

The design level entities are classes which can be mapped with the architectural components as follows:

Component	Class
Authentication	AuthenticScreen
	Register
	Login
	AdminLogin
Orders	OrderDetails
	MyOrders
	PlaceOrderPayment
Admin	Admin

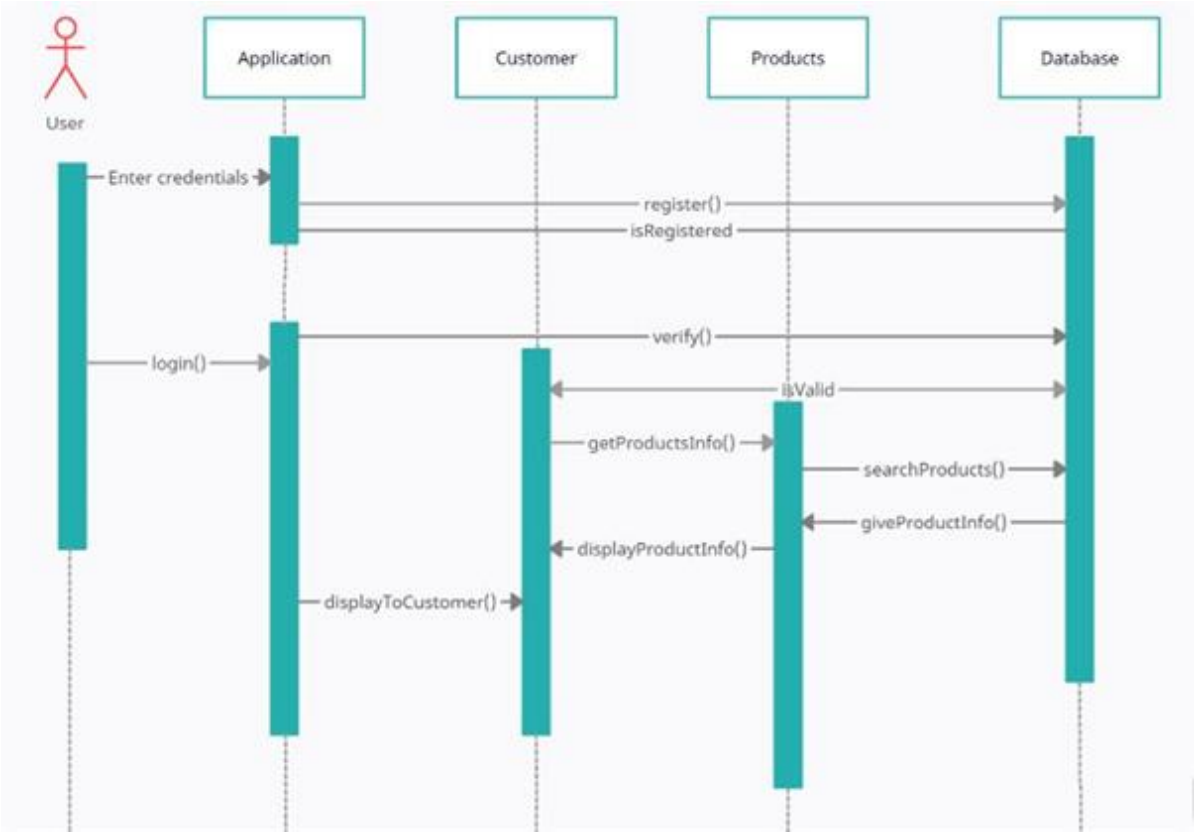
	ShiftOrders
	AdminOrderCard
	AdminOrderDetails
	uploadItems
Customer (User)	Customer
	Address
	Checkout
Cart (ShoppingCart)	Cart
	ProductPage
Other	StoreHome
	Search

Sequence diagram

A sequence diagram is a UML diagram used to specify flow of operations that are carried out. We modelled our system in sequence diagrams for each scenario.

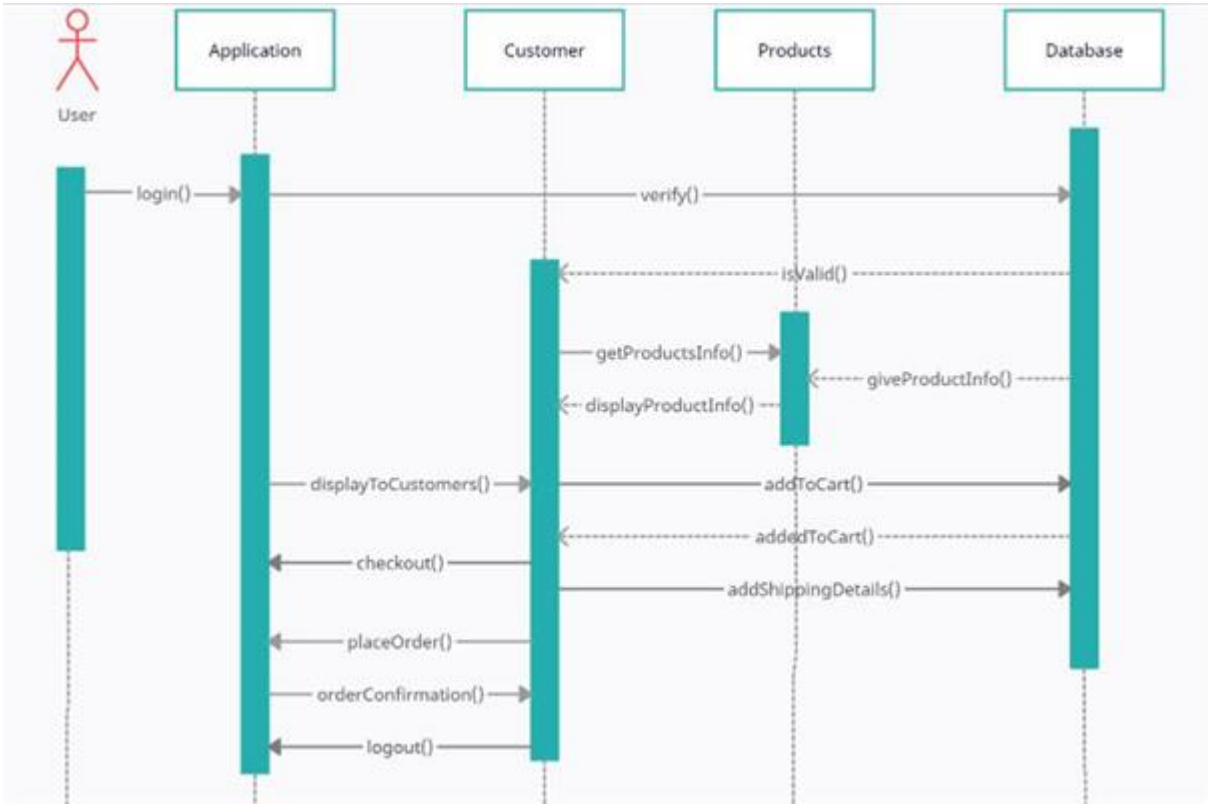
Register and login Scenario

- User enters credentials into the application which are sent to the firebase to register the user. When the operation is successful, database responds with successfully registered return message.
- Once user is registered, he can login in order to use the system. Application takes the login details from the user and send it to the firebase to verify the credentials.
- Once verified, user is logged in and can use the system.
- A user asks for product details which are fetched from the database and displayed to the user.



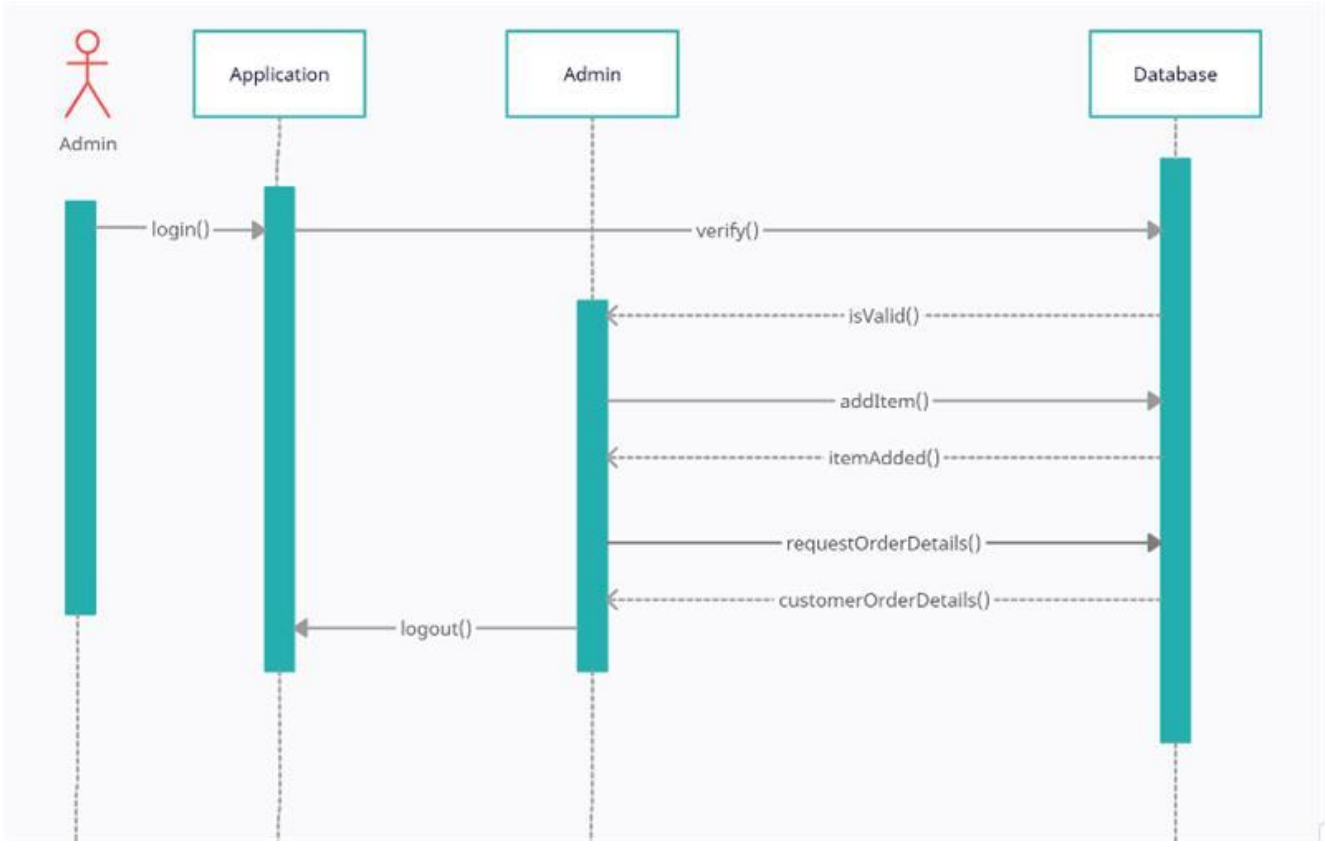
Add item to cart

- When user login into the system, he asks for product details which is fetched from the database.
- User adds item to cart and database responds with a message of item added to cart.
- User can check out by providing shipping details to the database.
- Once shipping details are provided, user can place order and is provided with successful message of order placed by the application.
- User can logout from the system.



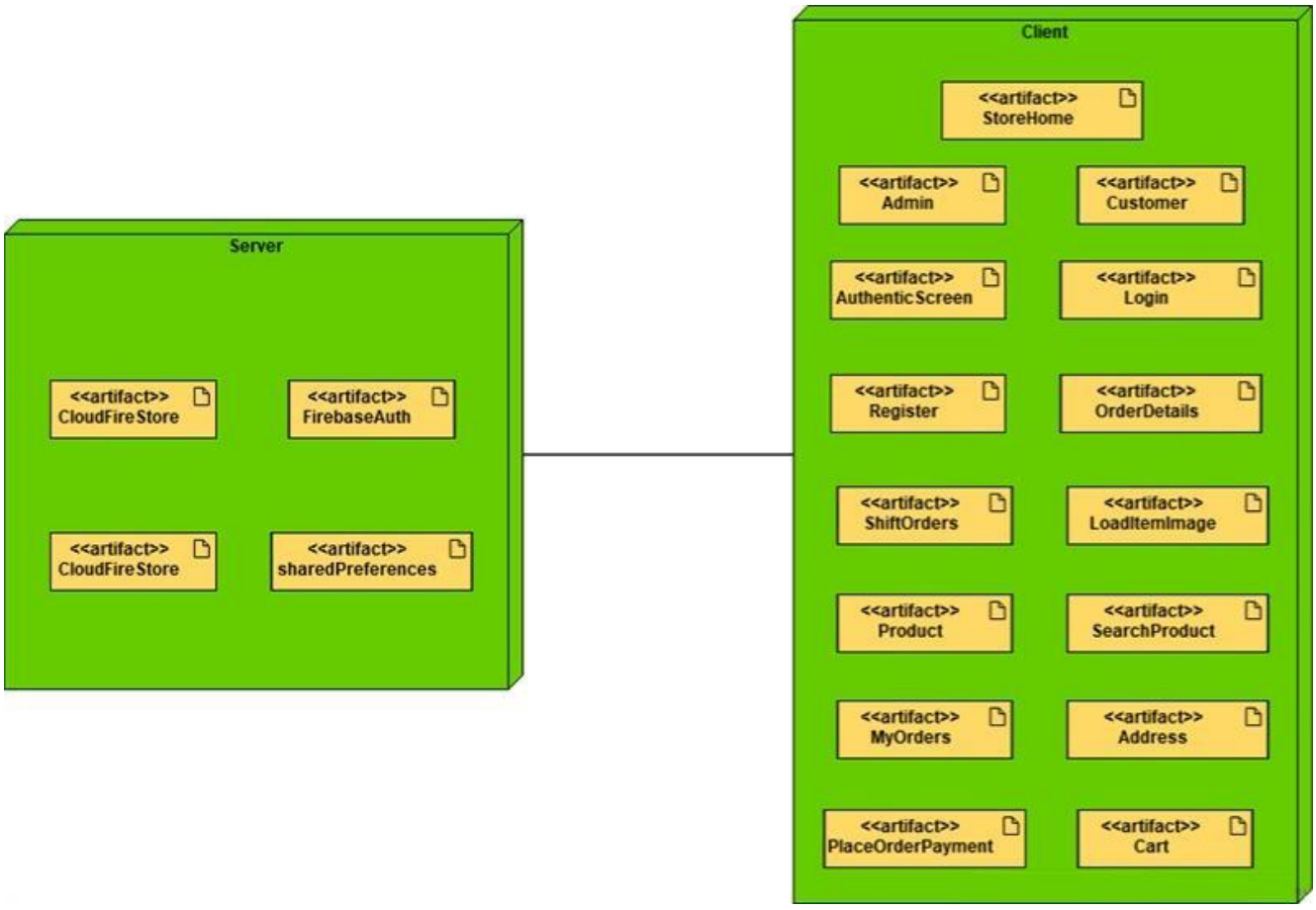
Admin adds item to catalogue

- Admin enters login credentials which are verified from the database.
- Once logged in, admin can add items to the catalogue.
- Items to be added are sent to the database and database responds with message of items added to catalogue.
- Admin can also request details of orders placed by the user which are provided by the database.
- Finally, admin can logout from the system.



Physical Location of Classes

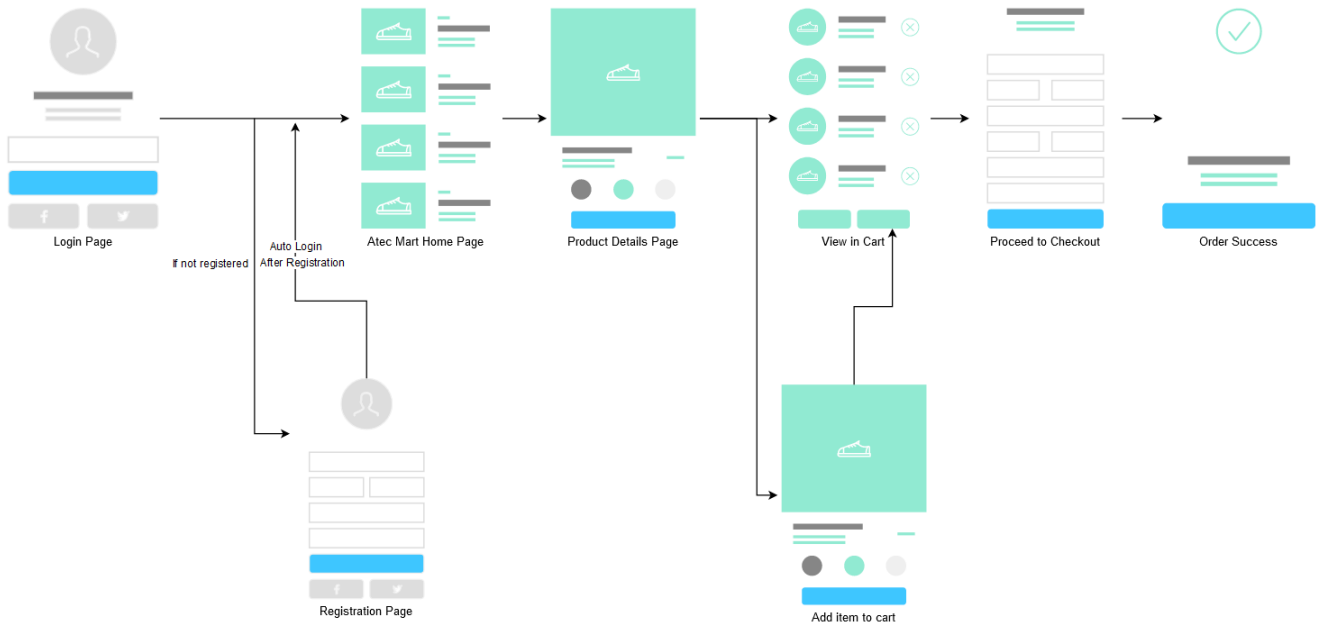
There are two major physical locations in the Atec Mart application. One location, containing most of the logic of the application is the Android / iOS device running the client side of the application, while the other location is the server or database end which fulfills the client requests.



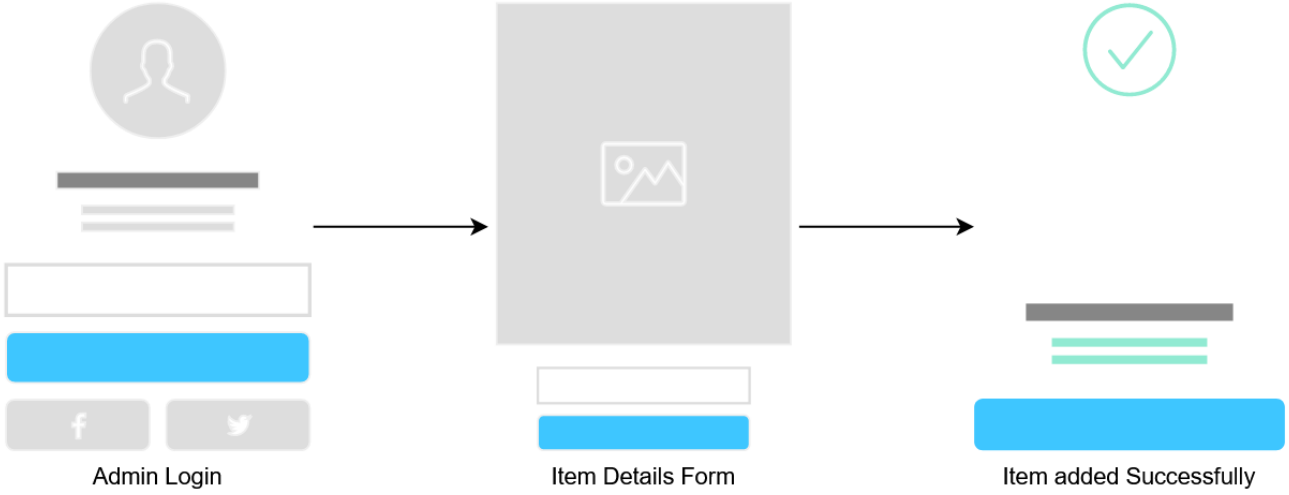
Flow Diagrams

These flow diagrams make the use cases very clear for the implementation. There are two flow diagrams, one for user and one for admin.

User Flow



Admin Flow



Analysis of Design for Coupling

Coupling is the degree of interdependence between different software modules. Coupling can be reduced by following best programming practices but still there remains a small level of coupling between different components.

- We have used **Composition** to evolve the code in future. Like **authentication** class **composite register** and **login** class so we can add further credentials as per require for example sign in with google.

- Most of the classes have their **own responsibility** (functional approach) and they **communicate** with other classes with **data** as an argument through constructor like we have implemented functions for **product detail** and **add to cart** which differ by data as per the product provided as an argument.
- Different **components** in our app have same design like buttons use in our app, background color and font style of text on different screens we have define these constant and then reuse them (data coupling). Further we can update them all at the same place.

Design supporting change requirements

- If there is a requirement where elements of our systems are to be re-usable in other systems, our system design can support this changed requirement. Our system design is based on modular approach and has less coupling between elements.
- If a requirement wants the system to be secure, we can protect inter-element communication and allow access of information to the concerned component only. This way, our design can support this changed requirement.
- A requirement where the developed system is supposed to run on different operating system is also supported by our design. Our application does not use core operating system functionalities which makes it independent of operating system.

Scenario of System Evolution

One very probable scenario for system evolution is the change of one of the parts of the system. For example,

In this scenario, it is assumed that the cart page has to be modified.

Change Statement: The way a cart displays the items and the pieces of information it shows needs to be modified and a more suitable version of it is to be developed

Design's Supportability to Change: The design for the application is highly modular having a separate file and class for the cart page, thus changing the functionality within the shopping cart class, this modification can be achieved. Due to **high modularity** and **minimum coupling**, this change can be supported by design with minimum change of code.

Individual Contributions

Team Member	Components
Hassan Raza Bukhari	Admin
	Customer
Salman Inayat	Authentication
	Others
Muhammad Umer Farooq	Orders
	Cart