Scoops Ahoy

Group Number 83

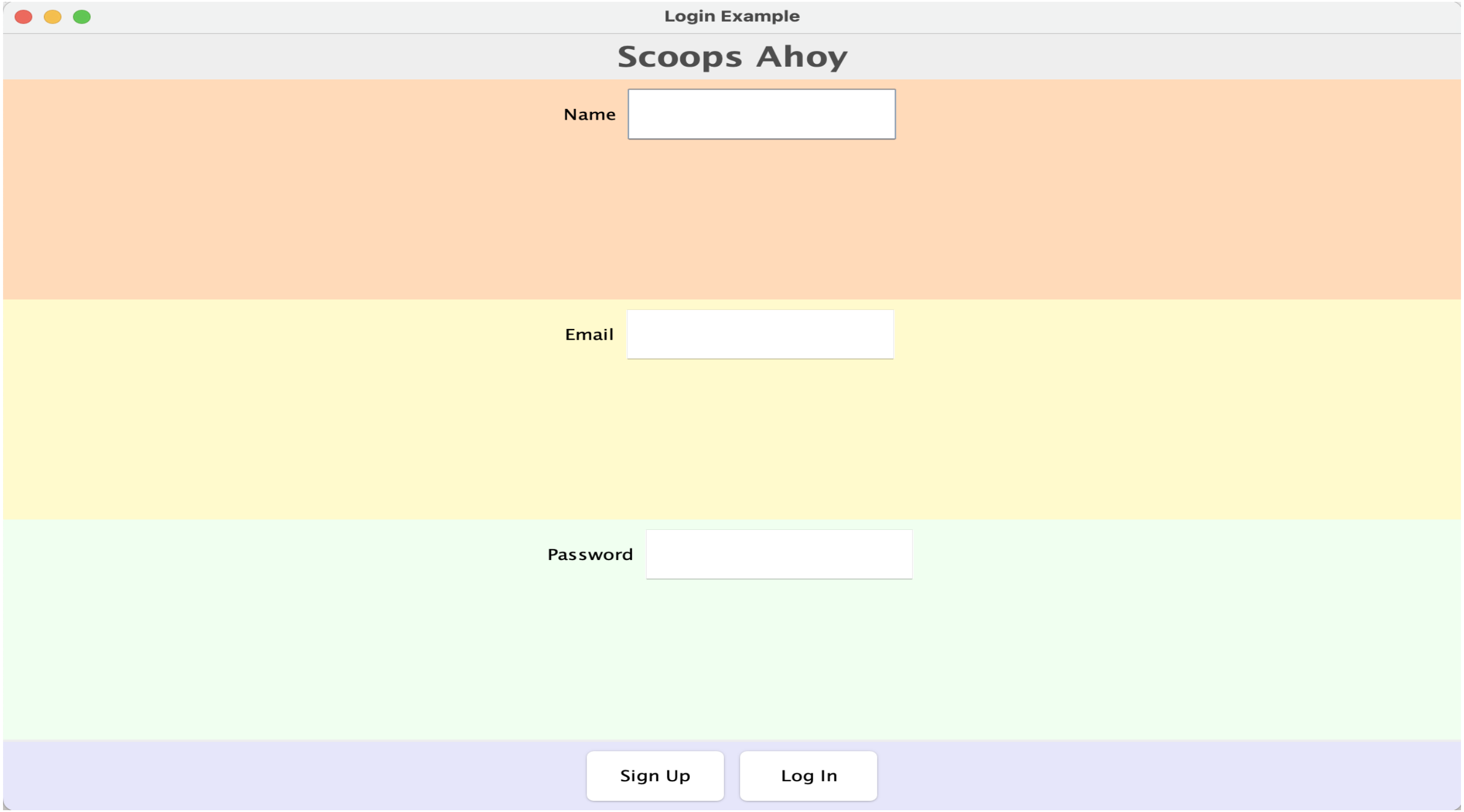
API usage

ABLY PUB/SUB CHANNELS API

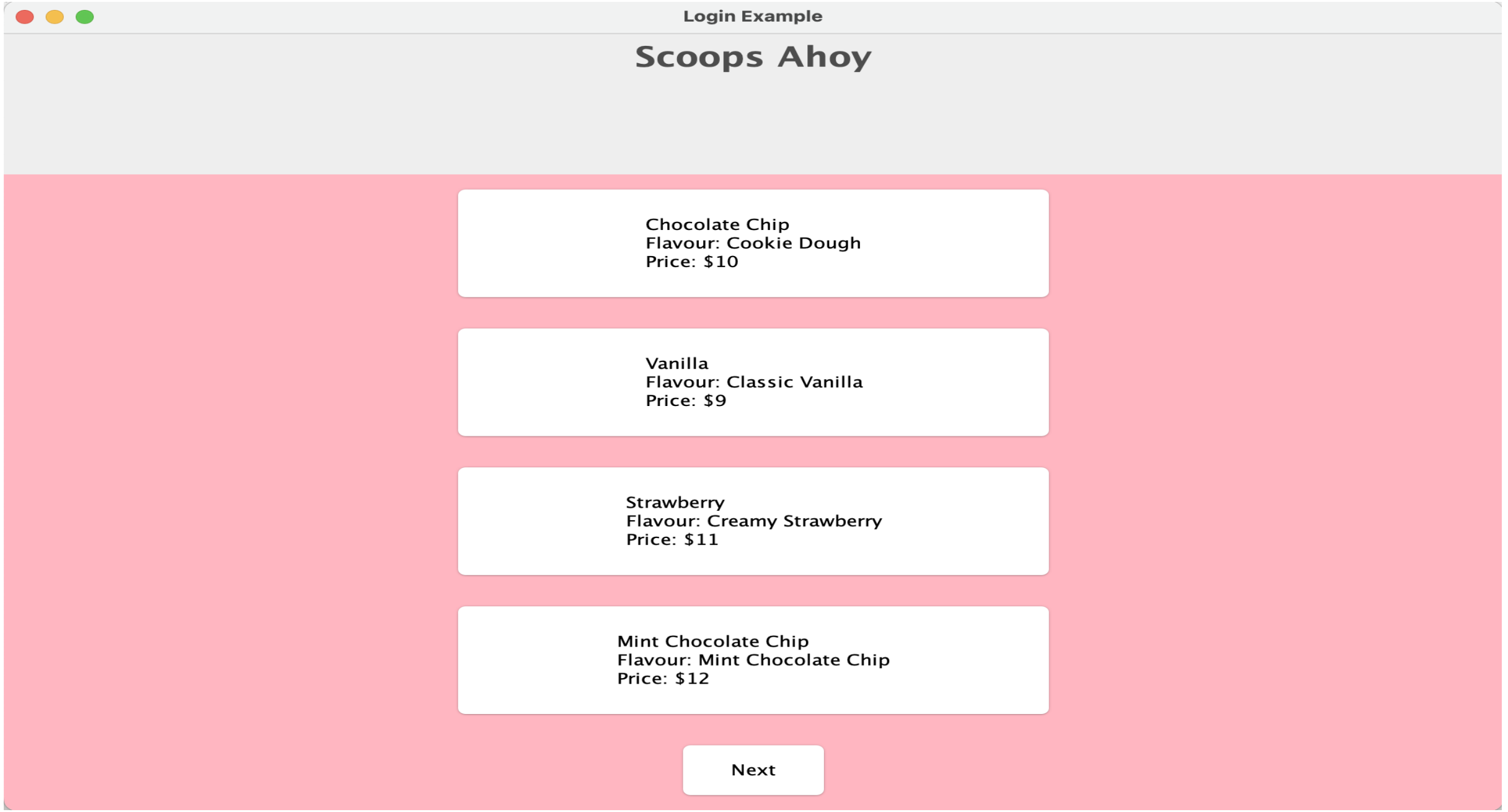
GOOGLE MAPS DIRECTIONS AND GEOCODING API

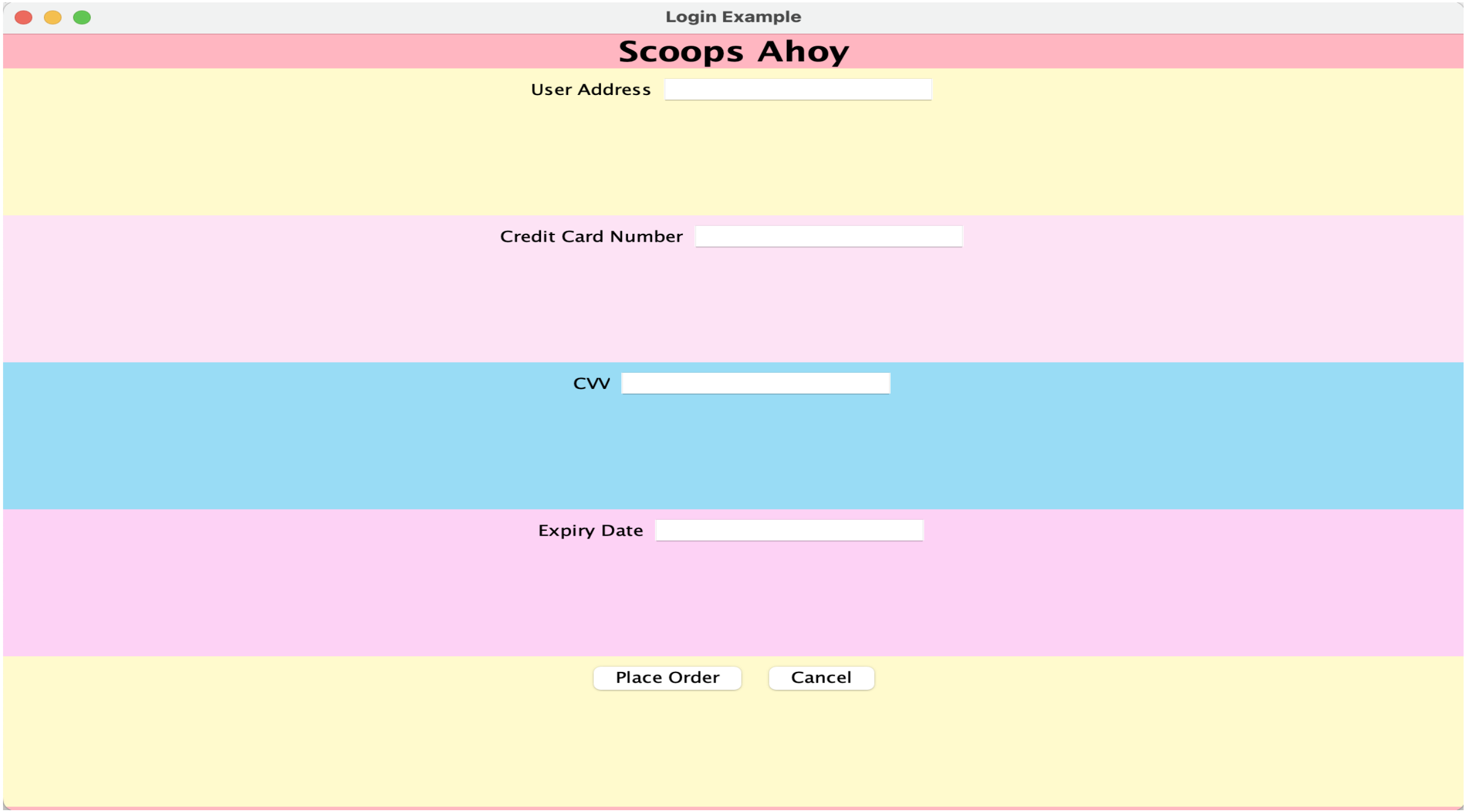
Specification

- The Scoops Ahoy – your friendly local ice cream shop!
- Login / Signup with Email and Password.
- Browse through a menu catalogue, add items from their shopping cart.
- Place orders
- Track real-time delivery progress on a map.
- Receive notification when order is delivered


Login Example
Scoops Ahoy
Name
Email
Password
Sign Up
Log In

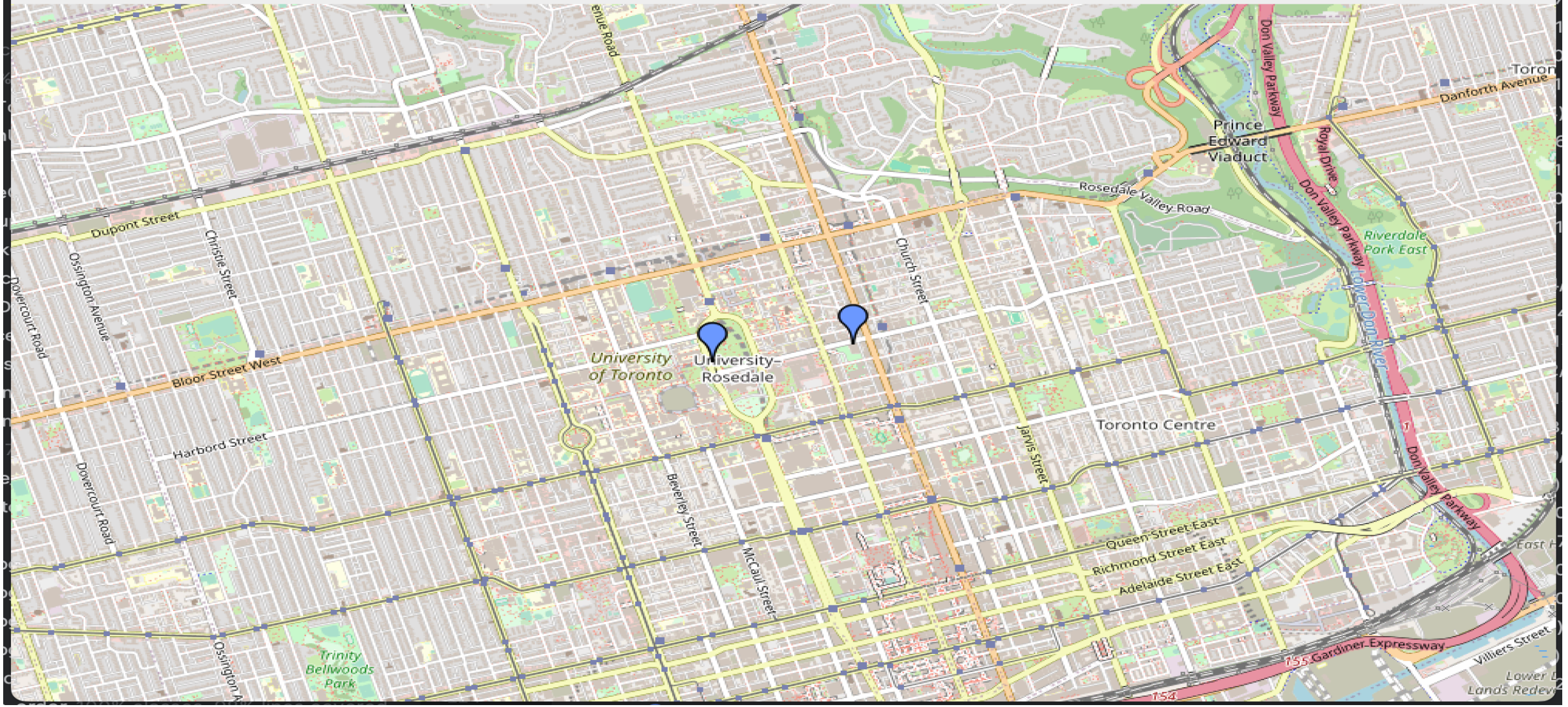

Login Example
Scoops Ahoy
Email
Password
Log in


Login Example
Scoops Ahoy
Chocolate Chip
Flavour: Cookie Dough
Price: $10
Vanilla
Flavour: Classic Vanilla
Price: $9
Strawberry
Flavour: Creamy Strawberry
Price: $11
Mint Chocolate Chip
Flavour: Mint Chocolate Chip
Price: $12
Next


Login Example
Scoops Ahoy
User Address
Credit Card Number
CVV
Expiry Date
Place Order
Cancel

Login Example

Scoops Ahoy – Track Order



Functionality Demonstration

How is our program adhering to the SOLID principles?

Single Responsibility Principle:

In our program, each class has exactly one responsibility and only one reason to change.

For example:

AddToCartController: this class receives inputs in their very raw form i.e., an array list of IceCream objects and prepares them in a format suitable for the use case i.e., it creates AddToCartInputData object which is the suitable format for the AddToCartInputBoundary interface. Therefore, the controller is focusing on only one aspect of the overall functionality and hence adhering to the single responsibility principle.

```
11 usages  1 inheritor  falakrastogi
public class AddToCartController {

    2 usages
    final AddToCartInputBoundary addToCartUseCaseInteractor;

    3 usages  falakrastogi
    public AddToCartController(AddToCartInputBoundary addToCartUseCaseInteractor) {

        this.addToCartUseCaseInteractor = addToCartUseCaseInteractor;
    }

    1 override  falakrastogi
    public void execute(ArrayList<IceCream> iceCreams) {

        AddToCartInputData addToCartInputData = new AddToCartInputData(iceCreams);
        addToCartUseCaseInteractor.execute(addToCartInputData);

    }
}
```

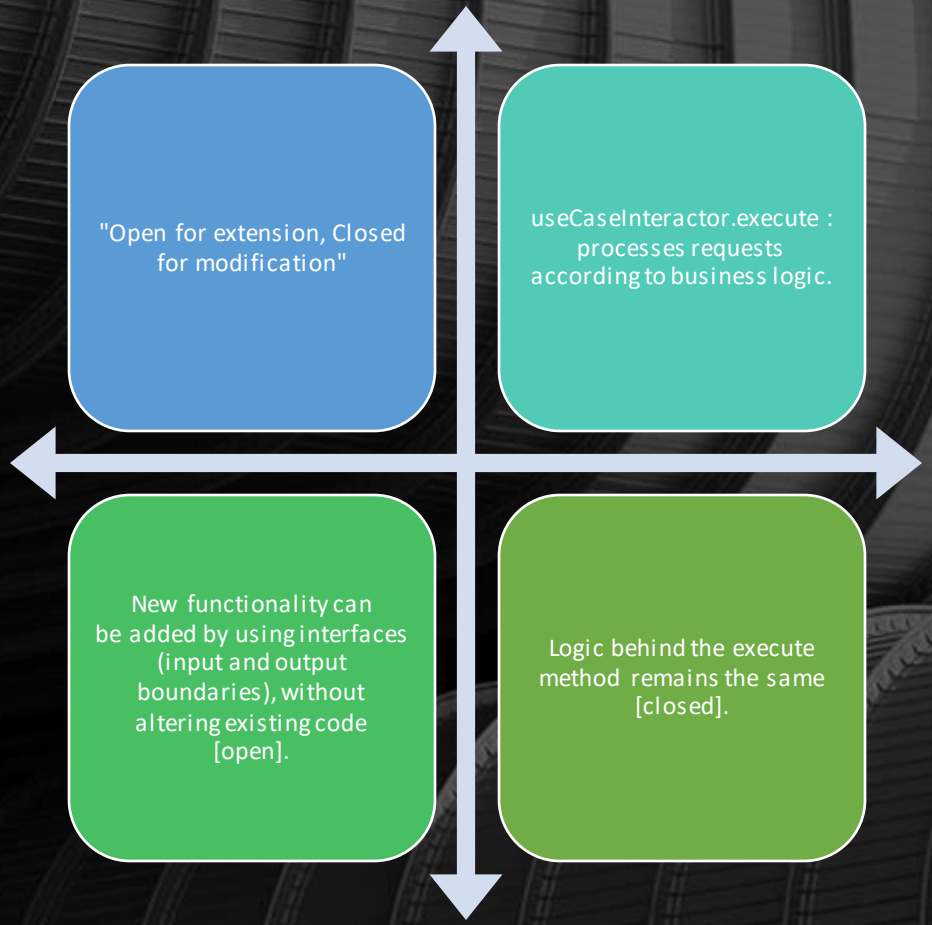
Dependency Inversion Principle

According to this Principle, high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

- In the given code:
- The **PlaceOrderInteractor** is a high-level module that orchestrates the process of placing an order.
- It depends on abstractions represented by the interfaces **PlaceOrderOutputBoundary**, **PlaceOrderDataAccessInterface**, **PlaceOrderUserDataAccessInterface**, and **TrackOrderInputBoundary**.
- The concrete implementations of these interfaces are injected into the **PlaceOrderInteractor** through its constructor, which adheres to the principle of depending on abstractions.

```
package use_cases.place_order;

import ...

4 usages  Eren +2 *
public class PlaceOrderInteractor implements PlaceOrderInputBoundary {

    4 usages
    final PlaceOrderOutputBoundary placeOrderPresenter;
    3 usages
    final PlaceOrderDataAccessInterface placeOrderDataAccessObject;
    3 usages
    final PlaceOrderUserDataAccessInterface placeOrderUserDataAccessObject;
    2 usages
    private final TrackOrderInputBoundary trackOrderInteractor;

    3 usages  Eren +1
    public PlaceOrderInteractor(
            PlaceOrderOutputBoundary placeOrderPresenter,
            PlaceOrderDataAccessInterface placeOrderDataAccessObject,
            PlaceOrderUserDataAccessInterface placeOrderUserDataAccessObjec
            TrackOrderInputBoundary trackOrderInteractor) {
        this.placeOrderPresenter = placeOrderPresenter;
        this.placeOrderDataAccessObject = placeOrderDataAccessObject;
        this.placeOrderUserDataAccessObject = placeOrderUserDataAccessObjec
        this.trackOrderInteractor = trackOrderInteractor;
    }
```

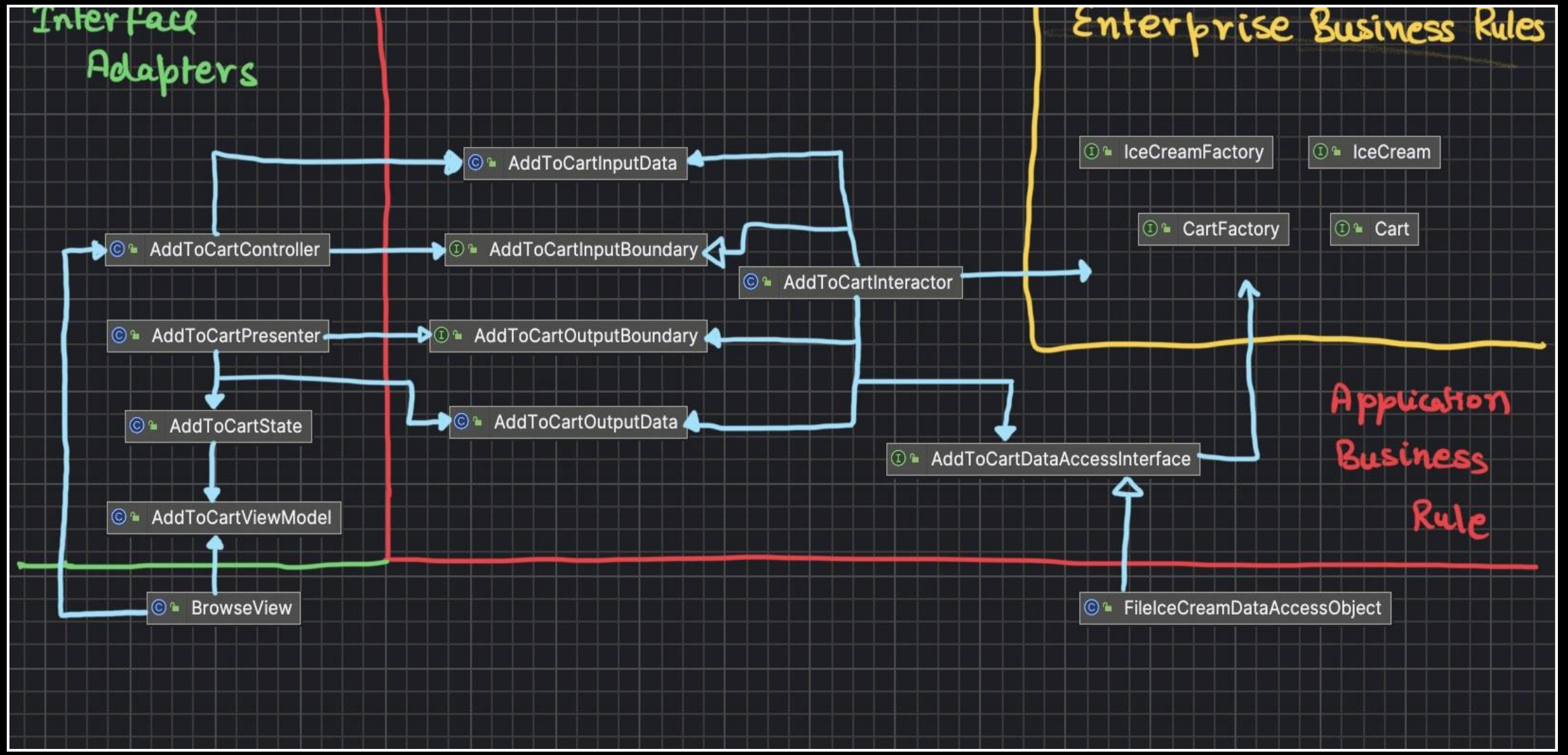
Open/Closed Principle



```java
@Override
public void execute(SignupInputData signupInputData) {
    if (userDataAccessObject.existsByEmail(signupInputData.getEmail())) {
        userPresenter.prepareFailView( error: "User already exists.");
    } else {

        LocalDateTime now = LocalDateTime.now();
        User user = userFactory.create(signupInputData.getName(), signupInputData.getEmail()
        userDataAccessObject.save(user);

        SignupOutputData signupOutputData = new SignupOutputData(user.getEmail(), now.toStri

        userPresenter.prepareSuccessView();
```

An instance of SignupDataAccessInterface

An instance of SignupOutputBoundary



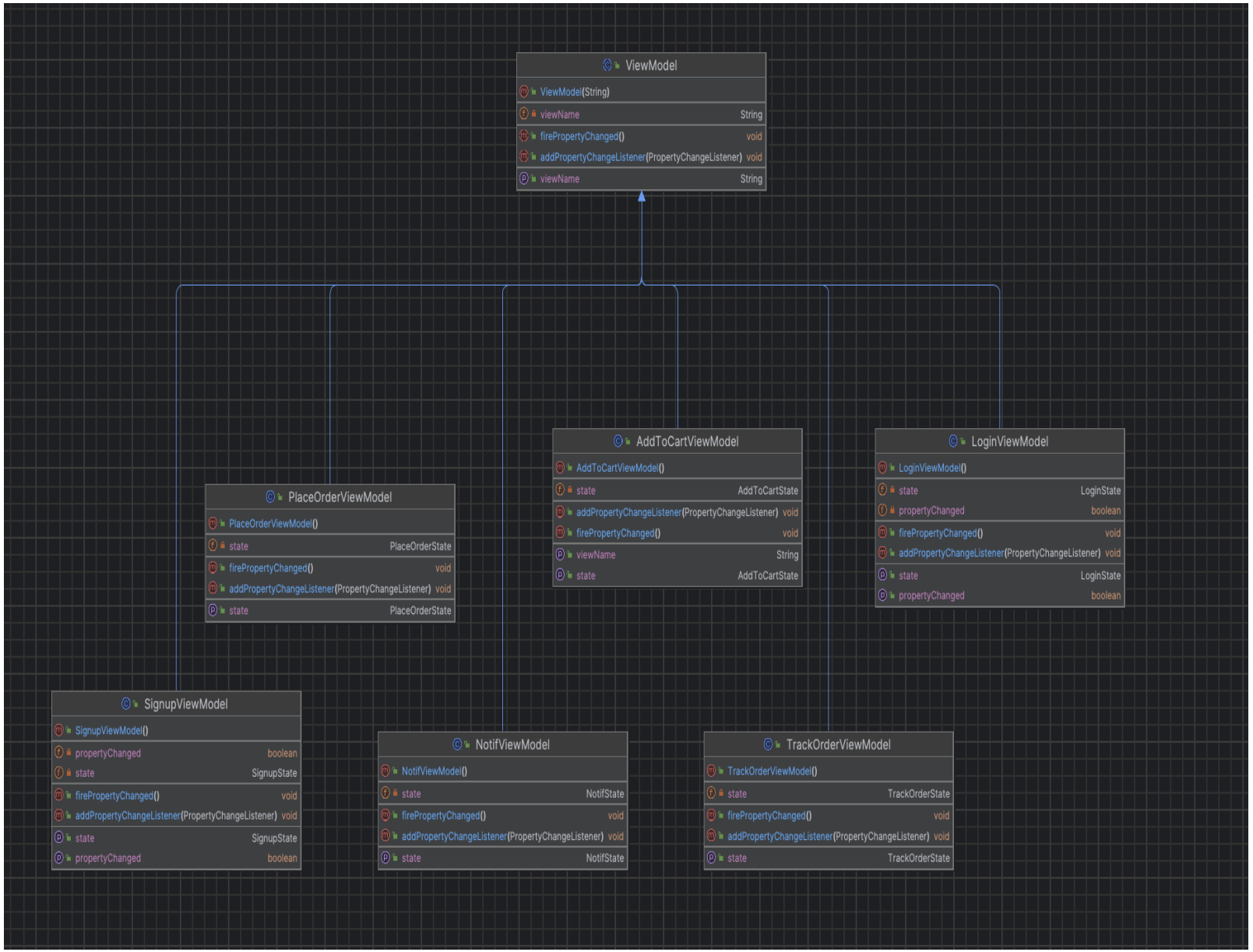
How Does Our Program Adhere To CA Engine?

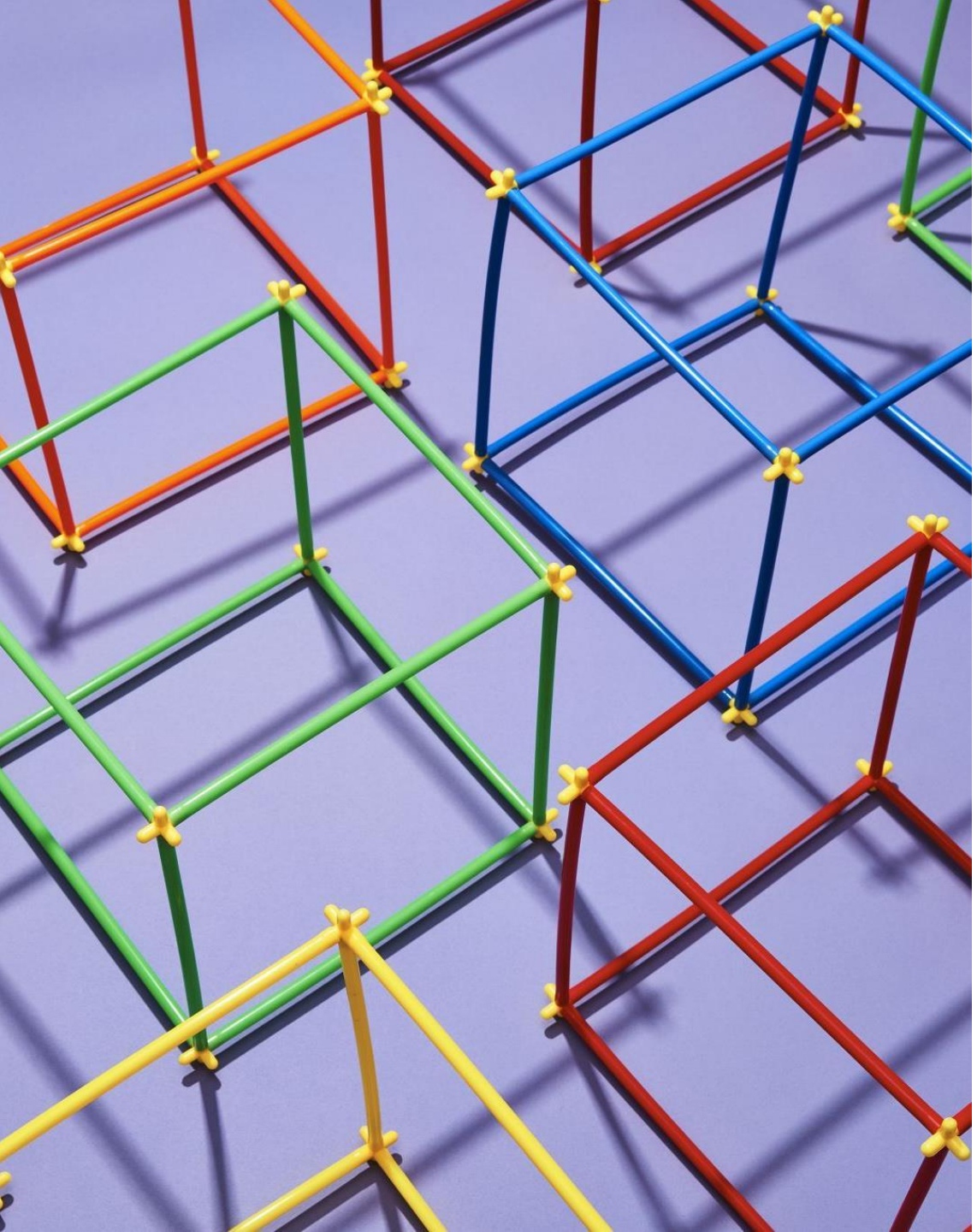
```
1  name,flavour,price
2  ChocolateChip,Cookie Dough,10
3  Vanilla,Classic Vanilla,9
4  Strawberry,Creamy Strawberry,11
5  MintChocolateChip,Mint Chocolate Chip,12
```

Database for the AddToCart Use Case

Design Patterns

Observer Pattern (firePropertyChanged, actionListener)



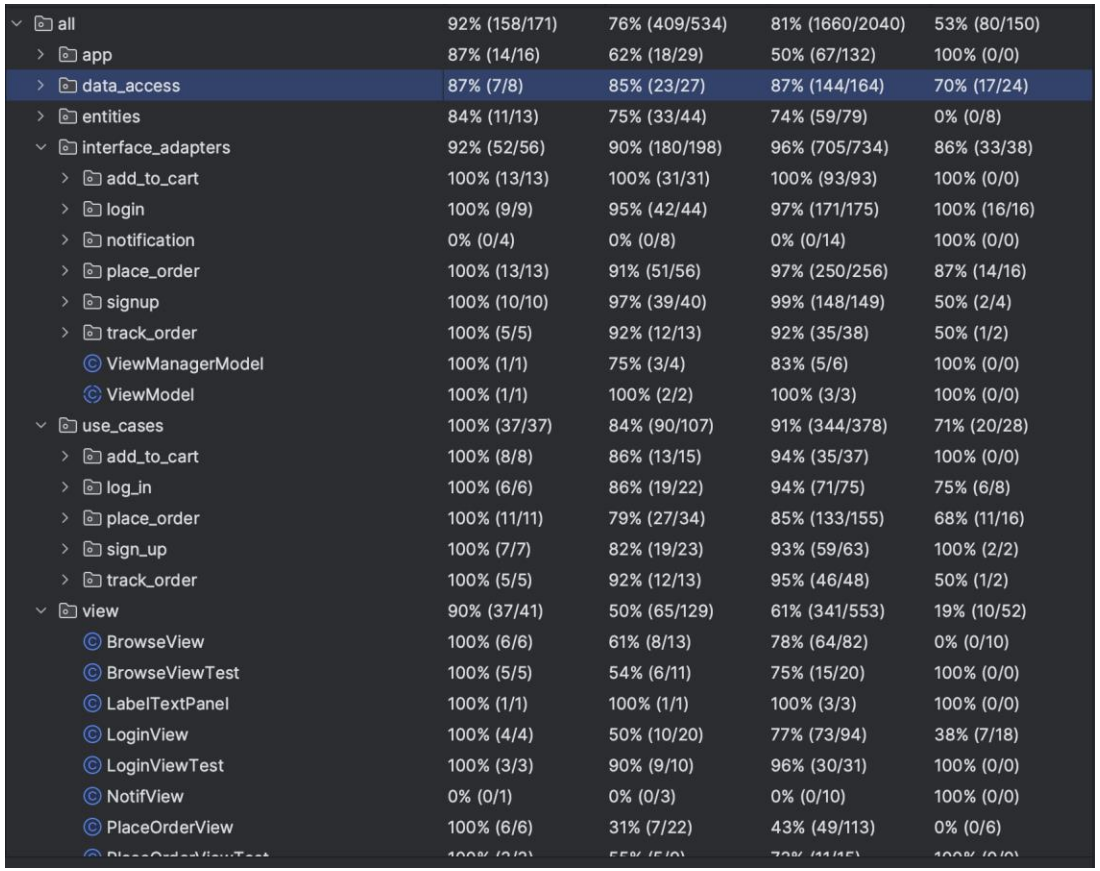


Dependency Injection

- Multiple examples throughout the code (e.g.- usage of interface adapters through boundaries)
- A lot of the test files use dependency injection too!
- Example: SignupDataAccessInterface is used in SignupInteractorTest methods, so the input can be either SignupPresenter or MockSignupPresenter!
- (SOLID Returns -- This is also an example of Liskov's Substitution!)

Testing

Testing



all	92% (158/171)	76% (409/534)	81% (1660/2040)	53% (80/150)
app	87% (14/16)	62% (18/29)	50% (67/132)	100% (0/0)
data_access	87% (7/8)	85% (23/27)	87% (144/164)	70% (17/24)
entities	84% (11/13)	75% (33/44)	74% (59/79)	0% (0/8)
interface_adapters	92% (52/56)	90% (180/198)	96% (705/734)	86% (33/38)
add_to_cart	100% (13/13)	100% (31/31)	100% (93/93)	100% (0/0)
login	100% (9/9)	95% (42/44)	97% (171/175)	100% (16/16)
notification	0% (0/4)	0% (0/8)	0% (0/14)	100% (0/0)
place_order	100% (13/13)	91% (51/56)	97% (250/256)	87% (14/16)
signup	100% (10/10)	97% (39/40)	99% (148/149)	50% (2/4)
track_order	100% (5/5)	92% (12/13)	92% (35/38)	50% (1/2)
ViewManagerModel	100% (1/1)	75% (3/4)	83% (5/6)	100% (0/0)
ViewModel	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)
use_cases	100% (37/37)	84% (90/107)	91% (344/378)	71% (20/28)
add_to_cart	100% (8/8)	86% (13/15)	94% (35/37)	100% (0/0)
log_in	100% (6/6)	86% (19/22)	94% (71/75)	75% (6/8)
place_order	100% (11/11)	79% (27/34)	85% (133/155)	68% (11/16)
sign_up	100% (7/7)	82% (19/23)	93% (59/63)	100% (2/2)
track_order	100% (5/5)	92% (12/13)	95% (46/48)	50% (1/2)
view	90% (37/41)	50% (65/129)	61% (341/553)	19% (10/52)
BrowseView	100% (6/6)	61% (8/13)	78% (64/82)	0% (0/10)
BrowseViewTest	100% (5/5)	54% (6/11)	75% (15/20)	100% (0/0)
LabelTextPanel	100% (1/1)	100% (1/1)	100% (3/3)	100% (0/0)
LoginView	100% (4/4)	50% (10/20)	77% (73/94)	38% (7/18)
LoginViewTest	100% (3/3)	90% (9/10)	96% (30/31)	100% (0/0)
NotifView	0% (0/1)	0% (0/3)	0% (0/10)	100% (0/0)
PlaceOrderView	100% (6/6)	31% (7/22)	43% (49/113)	0% (0/6)

- AblyDataAccessObject tests are dependent on whether the delivery simulator is running or not.
- We mocked this object for the interactors.

Code Organization

Code Organization

We used Gradle. We used a slightly different file structure that was a better fit for this build tool.

Gradle can run, build and test our app.

We used specific projects (such as API clients) from Maven.

The End