1. **Introduction**
   1. **Background**

PlateRescue is an iOS application dedicated to managing food wastage in restaurants. We often see that restaurants may have some surplus food that might go to waste if not utilized. To ensure no food waste, our app offers a solution.

Restaurant owners can minimize food waste by offering discounts on remaining food items. This not only reduces wastage but also attracts potential customers who get excited by seeing the discounted foods. Additionally, this will also help familiarize customers with the restaurant.

PlateRescue serves the best possible solution to reduce food waste. It also refrains the restaurant owners to serve stale food to the customers. Because by discount, almost every food got sold out.

* 1. **Objectives & Goals**
     1. **Objectives**
* To develop an iOS application that solves a real-life problem
* To ensure minimum or zero food wastage in restaurants using the application
* To increase the popularity of a restaurant
  + 1. **Goals**

In future, we will try to add the features of map to know the location of the restaurants. Then the users will know about which restaurants are located where.

1. **Methodology**

**Important Swift Classes:**

* UILabel:

A view that displays one or more lines of informational text.

* UITextField:

An object that displays an editable text area in the interface.

* UIButton:

A control that executes custom code in response to user interactions.

* UIImageView:

A view that displays a single image or sequence of animated images in the interface

* JSONDecoder:

An object that decodes instances of a data type from JSON objects.

**Important User-defined Classes:**

* Restaurant:

class Restaurant {

var email: String

var password: String

var username: String

var name: String

var location: String

}

This Swift code defines a basic Restaurant class with properties for email, password, username, name, and location.

* Food:

class Food {

var restaurantId: String

var restaurantName: String

var name: String

var details: String

var tableNo: Int

var price: Int

var discount: Int

var imageURL: String?

}

This Swift code defines a Food class with properties representing information about a menu item, such as the restaurant's ID and name, food details, table number, price, discount, and an optional imageURL for displaying an image. It captures essential details about a food item.

* Currency Information:

class X: Codable {

var success: Bool

var timestamp: Int

var base: String

var date: String

var rates: [String: Double]

}

This Swift code defines a class X conforming to the Codable protocol, representing a response from a currency API. It includes properties such as success, timestamp, base currency, date of the rates, and a dictionary (rates) mapping currency codes to their respective exchange rates (as Doubles). This structure facilitates decoding JSON responses from a currency API into Swift objects for easy manipulation.

**Important Procedures:**

* Switching Views:

let vc = storyboard?.instantiateViewController(identifier: "Signup") as! SignupViewController

present(vc, animated: true)

This Swift code instantiates a SignupViewController from the storyboard using the identifier "Signup" and presents it modally with an animated transition. The forced type casting (as!) assumes the correct type for the view controller.

* Save Data to Realtime Database:

restaurantsRef.childByAutoId().setValue(restaurant.toAnyObject())

This Swift code interacts with Firebase Realtime Database. It creates a new child node with a unique ID under the "restaurants" reference and sets its value to the serialized representation of a restaurant object using the toAnyObject method. This effectively adds a new restaurant entry to the database.

* Fetch Single Object from Realtime Database:

restaurantsRef.child(food.restaurantId).observeSingleEvent(of: .value) { (snapshot) in

if let requestsData = snapshot.value as? [String: Any] {

let value = requestsData

if let email = value["email"] as? String,

let password = value["password"] as? String,

let username = value["username"] as? String,

let name = value["name"] as? String,

let location = value["location"] as? String {

var restaurant = RestaurantEntity(key: snapshot.key,

email: email,

password: password,

username: username,

name: name,

location: location)

print(restaurant.toAnyObject())

}

}

}

 This Swift code observes a single event from the Firebase Realtime Database under a specific restaurant reference (restaurantsRef). It retrieves data from the snapshot, extracts values for email, password, username, name, and location, and then initializes a RestaurantEntity object with this information, associating it with the unique key from the database (snapshot.key). This is typically used for fetching and initializing a restaurant entity based on stored data.

* Fetch List of Objects from Realtime Database:

foodsRef.observeSingleEvent(of: .value) { (snapshot) in

if let requestsData = snapshot.value as? [String: [String: Any]]{

for(key, value) in requestsData {

if let restaurantId = value["restaurantId"] as? String,

let restaurantName = value["restaurantName"] as? String,

let name = value["name"] as? String,

let details = value["details"] as? String,

let tableNo = value["tableNo"] as? Int,

let price = value["price"] as? Int,

let discount = value["discount"] as? Int {

let food = Food(restaurantId: restaurantId,

restaurantName: restaurantName,

name: name,

details: details,

tableNo: tableNo,

price: price,

discount: discount)

food.imageURL = key

print(food.toAnyObject())

foods.append(food)

}

}

}

}

This Swift code fetches a snapshot of data from the Firebase Realtime Database under the "foodsRef" reference. It iterates through the data, extracting details for each food item, and initializes corresponding Food objects. The created Food instances are then appended to an array called foods, incorporating additional information such as the image URL associated with each item.

* Save Data(Image) to Firestore:

let storage = Storage.storage()

let data = Data()

let storageRef = storage.reference()

let localFile = fileURL

foodKey = foodsRef.childByAutoId().key

let photoRef = storageRef.child(foodKey)

let uploadTask = photoRef.putFile(from: localFile, metadata: nil) { (metadata, err) in

guard let metadata = metadata else {

print(err?.localizedDescription)

return

}

}

This Swift code sets up Firebase Storage, creates a reference to the storage root, generates a unique key for a food item, and initiates an upload task to store a file from a local URL (fileURL) under the generated key in the storage. Any error during the upload is printed to the console.

* Retrieve Data from Firestore:

let storage = Storage.storage()

let storageRef = storage.reference()

let ref = storageRef.child(foodKey)

ref.downloadURL { (url, error) in

if let error = error {

print("Error getting download URL: \(error.localizedDescription)")

return

}

if let downloadURL = url {

self.imageDownloaded.sd\_setImage(with: downloadURL, completed: { (image, error, cacheType, imageURL) in

if let error = error {

print("Error setting image: \(error.localizedDescription)")

}

})

}

}

This Swift code sets up Firebase Storage, creates a reference to the storage root, and then retrieves the download URL for a specific file identified by foodKey. It asynchronously sets the image of a UI element (self.imageDownloaded) using the retrieved download URL, and any errors during the process are printed to the console. The sd\_setImage method suggests the use of the SDWebImage library for efficient image loading and caching.

* Register Account:

FirebaseAuth.Auth.auth().createUser(withEmail: email, password: password, completion: { [weak self] result, error in

guard let strongSelf = self else {

return

}

guard error == nil

else {

return

}

} )

This Swift code uses Firebase Authentication to create a new user account with the specified email and password. The completion handler checks for successful account creation (result), and if an error occurs, it is handled within the error parameter.

* Authenticate Account:

FirebaseAuth.Auth.auth().signIn(withEmail: email, password: password, completion: { [weak self] result, error in

guard let strongSelf = self else {

return

}

guard error == nil

else {

return

}

} )

This Swift code utilizes Firebase Authentication to sign in a user with the provided email and password. The completion handler checks for successful sign-in (result), and any potential errors are handled within the error parameter. The use of [weak self] helps prevent strong reference cycles in asynchronous callbacks.

* JSON parsing:

let urlString = "http://api.exchangeratesapi.io/v1/latest?access\_key=72af372fa80706691a117de75a528c8d&format=1"

let url = URL(string: urlString)

guard url != nil else {

return

}

let session = URLSession.shared

let dataTask = session.dataTask(with: url!) { (data, response, error) in

print(error)

print(data)

print(response)

if error == nil && data != nil {

let decoder = JSONDecoder()

do {

let x = try decoder.decode(X.self, from: data!)

dollarRate = x.rates["BDT"]

print(dollarRate)

}

catch {

print("Error in JSON parsing")

}

}

}

dataTask.resume()

This Swift code performs an asynchronous HTTP GET request to the specified URL using URLSession. It retrieves data from the response, attempts to decode it into a model (X), and prints the exchange rate for the Bangladeshi Taka ("BDT") against the US Dollar. Any errors during the process are printed to the console.

* 1. **Flowchart of methods**

Flowchart for User:

Start

Feed of the app

No

Select a food item?

Yes

Show details of food

Yes

No

Change currency?

Currency in $

Currency in BDT

End

Flowchart of Admin:

Start

Feed of the app

Press Login Button

No

Login Successful?

Yes

See Restaurant owner requests

Approve/Delete request

End

Flowchart of restaurant owner:

Start

Feed of the app

Press Login Button

No

Login Successful?

Yes

See Added items

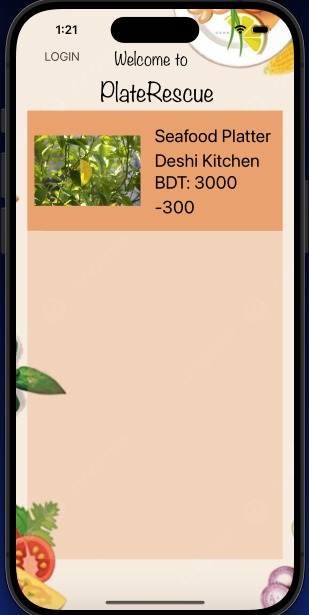
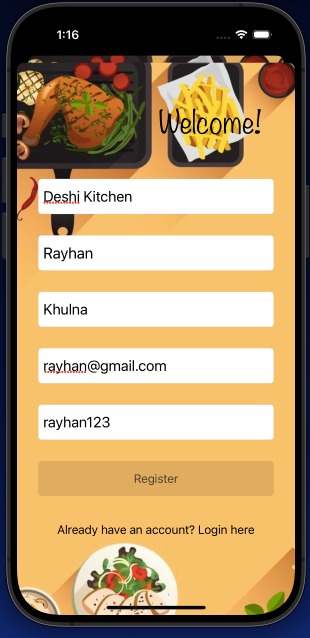
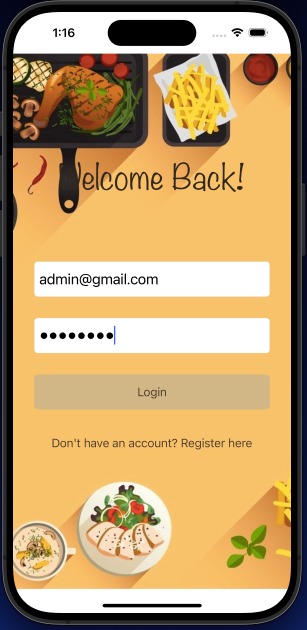
Press Add item button

Add New items

End

1. **Result**

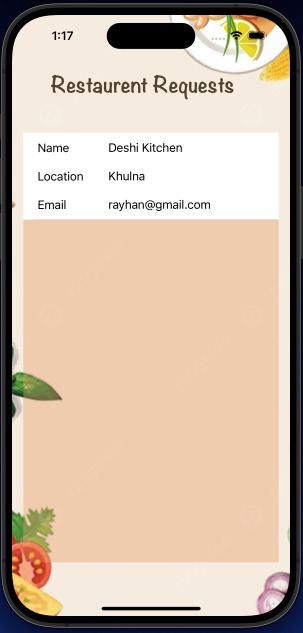
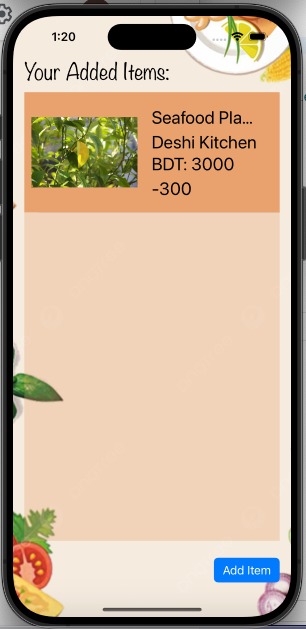
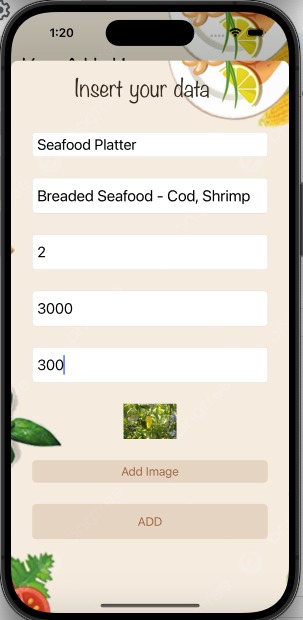
In the application admin can approve restaurant request and include the new restaurant in the system. Once approved the restaurants can add their offers which are dynamically showed on the feed. General users can access the offer details by simply tapping on the feed. So, the application functions smoothly and is able to complete tasks of admin, restaurants and general users.

Screenshot 3.1: Screenshot 3.1: Screenshot 3.2:

User Homepage Restaurant owner Login page

registration page

Screenshot 3.4: Screenshot 3.5: Screenshot 3.6:

Admin page Restaurant owner Restaurant owner

Homepage insert items page

Screenshot 3.7: Screenshot 3.8:

Item details page Item details page

In taka in dollar

1. **Discussion**

The main objective of this app was to minimize food wastage by allowing restaurant owners to offer surplus food to users at discounted prices. We tried to make a user friendly and intuitive design for both restaurant owners and consumers. We tried to include JSON parsing for enhancing more features for users outside of our country by ensuring exchange of currency. Our app will promote sustainable practices and contributes to a more environmentally conscious approach to food consumption. This will also provide restaurants with flexible solution to manage their surplus food inventory. During the making of the app, we faced quite difficulties because of shortage of lab facilities and lacking of personal laptop facilities where to run Xcode. Therefore, we had to spend hours in the lab in the limited capacities of iMac. At last, we completed our app with required features. We also learnt new languages, processes, structures, functions for making such application.

1. **Conclusion & Future Work**

**5.1 Conclusion**

PlateRescue application helps to ensure proper use of the prepared food in restaurants. It also gives opportunity to the new restaurants to become popular among the customers. By using the application, all the customers, restaurant owners and developers will be able to make profit.

**5.2 Future Work**

Integrating map in the application would help the customers to find the restaurant easily. It will be user friendly when the user will see the restaurants near his location and will be able to book the tables from the application. Developer of the application will get profit by making the application popular among the restaurant owners. He can give the owners of the restaurants free trail for 1 year and later include paid subscription.

1. **Reference**

i. Contents provided in the laboratory

ii. iOS Academy youtube channel