Inventory Management iOS Application

Architecture review date	May 26, 2024
Project lead	@krish mittal
On this page	• 📋 Overview • 🚩 Architecture • 👥 Stakeholders • ✏️ File Details • 🎯 Goals • 👣 Next steps

📋 Overview

This Project is made primarily for Swipe's Take Home Project Assignment. The project is built using Swift and SwiftUI. No 3rd Party Packages were used in making of this project.

The project aims to full fill the following requirements:

1. **Listing Product Screen**

- Develop a screen that displays a list of products
- User should be able to do the following things on the screen
  - Search products
  - Look for all the products
  - Scroll through the list
  - Include a button that allows the user to navigate to the Add Product screen.
- Ensure that images are loaded for each product from a URL. If the URL is empty, use a default image instead.
- Populate all required fields, such as product name, product type, price, and tax, for each product in the list.
- Provide a visual indicator, such as a progress bar, to show loading progress.
- Use the provided API endpoint and HTTP method to retrieve the product data.

2. **Add Product Screen**

- Select the product type from a list of options.
- Enter the product name, selling price, and tax rate using text fields.
- Optionally select images in JPEG or PNG format with a 1:1 ratio.
- Validate fields such as product type selection, non-empty product name, and decimal numbers for selling price and tax.
- Submit the data using the POST method to API endpoint
- Use a user-friendly interface to display the screen.
- Document the code for future reference.
- Provide clear and concise feedback to the user upon completion of the action.

Installation and API endpoints

Just simply clone the project from Github and choose your development team.

GitHub Repo: GitHub - krishmittal21/InventoryManagement

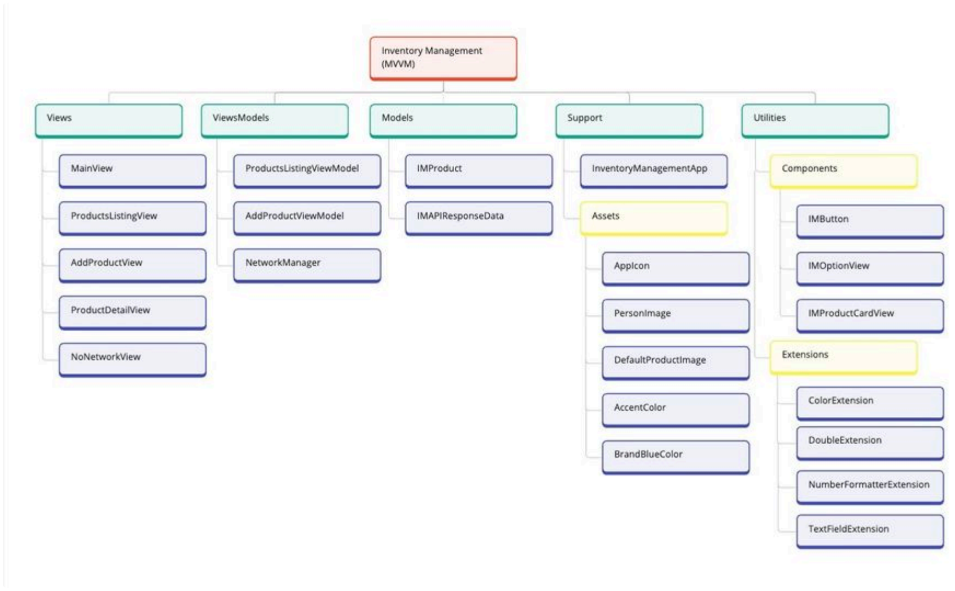
GET: https://app.getswipe.in/api/public/get

POST: https://app.getswipe.in/api/public/add

🚩 Architecture

The Project use MVVM Architecture

Miro Sitemap.pdf
26 May 2024, 08:57 AM



👥 Stakeholders

Name	**Role**
@krish mittal	Sole Dev
Vamsi Rudraraju	Tech Lead / Interviewer
Swipe	Interviewer

✏️ File Details

File	Definition	Notes
IMProduct	This struct represents product model for inventory management, including	It also contains the product type enum which helps fix product type the user can

	properties for image URL, price, product name, type, and tax percentage, along with enums for coding keys and product types	choose from. Also contains JSONKeys enum which helps in the GET request
IMAPIResponseData	This struct represents the data structure for API responses, including properties for message, product details, product ID, and success status, along with coding keys to map JSON keys to Swift properties.	
InventoryManagement App	Defining the main entry point with a WindowGroup containing the MainView.	
MainView	This view is for the main content view, which displays either a ProductsListingView if the network is connected or a NoNetworkView if not, with a state object for managing network connectivity.	
NetworkManager	This class is for managing network connectivity, using NWPathMonitor to observe network status changes. It includes a published property isConnected to reflect the current network connection status and an initialiser to start monitoring network changes on a designated queue.	
NoNetworkView	View for displaying a message when there's no internet connection and called in the MainView. It includes an icon representing a Wi-Fi symbol, a message informing the user about the need for an internet connection.	
IMButton	Custom button component, with customisable text, background color, and text color. It includes an action closure to perform when the button is tapped, wrapped within a RoundedRectangle shape.	
IMProductCardView	Custom view for displaying a card for a product in home screen (ProductListingView). It includes an image, product name, and price, with an optional button for additional actions. The card design is customisable, with rounded corners and a stroke overlay.	

IMOptionView	This view is used for displaying an option in ProductListingView with a title, selection indicator, and a tap gesture to trigger an action. The option view allows users to select or deselect an option, with an animation applied to the selection indicator.	Logic behind getting different product listing after selecting different option yet to be implemented.
ColorExtension	An extension for the SwiftUI Color type, defining a custom brand color named "BrandBlue". This extension allows easy access to the brand blue color throughout the inventory management application.	Allows for easy change in colors. Just have to change color in extension and instantly applied to the entire app.
NumberFormatterExtension	This provides a static computed property currencyFormatter that returns a pre-configured instance of numberFormatter with the number style set to .currency. This extension facilitates easy formatting of numbers as currency values throughout the application.	Mostly used in text fields
DoubleExtension	An extension for the Double type, adding a method formattedCurrency to format the double value as a currency string. It utilizes the currencyFormatter, converting the double value to a formatted currency string.	Used for price and tax
TextFieldExtension	An extension for TextField, adding a method customTextFieldStyle() to apply a custom styling defined in the TextFieldStyle view modifier. This styling includes a specific font, padding, corner radius, and border overlay.	Used in Add Product View text fields
ProductDetailsView	View for displaying detailed information about a product after tapping on a product card view in ProductListingView. It includes product details such as type, name, rating, total price, price breakdown, and description. The view also features an image of the product, a toggle button for indicating availability.	The Rating and Description are hard coded and not fetched from the given API, it's something which can be implemented in the future in the API as these are important details for a product. The toggle button does not perform a function for now, can be implemented in the future
ListingProductsView	View for listing products retrieved in the ListingProductsViewModel. It includes a header displaying the user's products, a search bar for	The option selector for filtering products based on availability or category is not functional for now. The Share button also not functional.

	filtering products, and an option selector for filtering products based on availability or category. The main content area displays a grid of product cards, each representing a product in the inventory. Additionally, there's a button for adding new products. The view also supports refreshing the product list and navigating to the detailed view of a selected product.	
ListingProductsViewModel	The ProductsListingViewModel Class is responsible for fetching and managing the products list data from the API. It provides the necessary data and functionality to the products listing view.	Properties: • products: [IMProduct]: A published property that holds the array of products fetched from the server. • isLoading: Bool: A published property that indicates whether the data is currently being loaded from the server. • search: String: A published property that stores the user's search input for filtering the products list. • filteredProducts: [IMProduct]: A computed property that returns the filtered list of products based on the search property. If the search property is empty, it returns the complete products array. • urlString: String: A private constant that holds the URL string for the API endpoint to fetch the products list. Methods: • init(): The initialiser for the view model, which immediately calls the fetchProducts() method to fetch the products list from the server. • fetchProducts(): This method is responsible for fetching the products list from the server. It sets the isLoading property to true, constructs the URL, and makes a URLSession data task request to the API endpoint. Upon receiving the response, it parses the JSON data using the parseJSONData(_:) method and updates the products and isLoading properties on the main queue. • parseJSONData(_:): A helper method that takes the response data from the server and decodes it into an array of IMProduct objects using

		JSONDecoder. If the decoding is successful, it returns the array of products; otherwise, it returns nil.
AddProductView	View called after pressing Add Product button in ProductListingView for adding a new product. It includes input fields for product details such as category, name, description, price, and tax rate. Additionally, it provides an option to select a product photo. The view also features buttons for saving the product or adding another product.	The save button will dismiss the page after alerting the user after a result but the add another button will keep on the AddProductView and clear all fields allowing users to add another product
AddProductViewModel	The AddProductViewModel class handles the logic for adding a new product, including data validation, request construction, and updating the UI based on the server response. Properties: • errorMessage: A published property to store any error message that needs to be displayed. • name, description, productType, sellingPrice, tax, category: Published properties to store the user input values for the new product. • selectedPhotoData: Published property to store the data of the selected photo for the product. • isLoaded: A published property to indicate if the data is being loaded or not. • alertMessage: A published property to store the message to be displayed in the alert. • isSuccess: A published property to indicate if the product was added successfully or not. • finalPrice: A computed property that calculates the final price of the product based on the selling price and tax rate. Methods: • clearFields(): This method resets all the input fields to their default values. • uploadProducts(): This is the main method that handles the logic for	The two Append functions for multipart/form-data request: 1. appendFileField(boundary:name:data:filename:contentType:): ◦ This function is responsible for appending a file field (e.g., an image) to the request body when uploading data using the multipart/form-data format. ◦ It takes the following parameters: ▪ boundary: A string that represents the boundary separator for multipart/form-data requests. ▪ name: The name of the file field. ▪ data: The actual data of the file (e.g., image data). ▪ filename: The filename to be used for the file. ▪ contentType: The content type of the file (e.g., "image/jpeg"). ◦ Inside the function, it constructs the field data string with the necessary headers and separators using the provided parameters. ◦ It then appends the field data string, the actual file data, and a newline separator to the body property, a Data object representing the entire request body. ◦ Finally, it returns the updated body object. 2. appendFormField(boundary:name:value:):

	uploading the new product. It performs data validation, constructs the request body with form fields and image data (if any), and sends a POST request to the server. The response from the server is handled asynchronously, and the relevant properties are updated based on the success or failure of the request. • appendFileField(boundary:name:data:filename:contentType:): A private method that appends the image data to the request body with the correct boundary and headers. • appendFormField(boundary:name:value:): A private method that appends a form field to the request body with the correct boundary.	◦ This function is responsible for appending a regular form field to the request body when uploading data using the multipart/form-data format. ◦ It is a generic function that accepts a LosslessStringConvertible type (T) for the value parameter. This means that the value can be of any type that can be converted to a string without losing any information (e.g., String, Int, Double). ◦ It takes the following parameters: ▪ boundary: A string that represents the boundary separator for multipart/form-data requests. ▪ name: The name of the form field. ▪ value: The value of the form field. ◦ Inside the function, it constructs the field data string with the necessary headers and separators using the provided parameters. ◦ It then appends the field data string to the body property, which is likely a Data object representing the entire request body. These two functions are used together in the uploadProducts() method to construct the request body for the multipart/form-data request. The appendFileField function is used to append the image data (if any), while the appendFormField function is used to append the other form fields, such as product name, product type, price, and tax.

🎯 Goals

1. Develop a screen that displays a list of products fetched from the provided API endpoint.
2. Implement search functionality to allow users to search for products by name.
3. Enable users to view all products.
4. Ensure the product list is scrollable to accommodate a large number of products.
5. Include a button or navigation link that allows users to navigate to the Add Product screen.
6. Display product images by loading them from the provided URLs, and use a default image if the URL is empty or invalid.
7. Display relevant product information, such as product name, product type, price, and tax, for each product in the list.

8. Provide a visual indicator, such as a progress bar or spinner, to show the loading progress while fetching data from the API.
9. Use the provided API endpoint and HTTP method (presumably GET) to retrieve the product data.
10. Implement a user interface that allows users to select the product type from a list of options (e.g., dropdown, picker, or radio buttons).
11. Provide text input fields for users to enter the product name, selling price, and tax rate.
12. Include functionality to allow users to optionally select one or more images in JPEG or PNG format
13. Implement input validation to ensure that the product type is selected, the product name is not empty, and the selling price and tax rate are valid decimal numbers.
14. Integrate with the provided API endpoint to submit the product data using the POST method.
15. Design a user-friendly interface for the Add Product Screen, following best practices for usability and accessibility.
16. Document the code thoroughly, including explanations of the functionality, data structures, and any external dependencies or libraries used.
17. Provide clear and concise feedback to the user upon successful or failed submission of the product data, such as success or error messages or alerts.

👣 Next steps

1. Implement the Filter based on availability and category in the Product Listing View
2. Implement share button
3. Implement Description and ratings in the API
4. The toggle button inside the product details view for availability
5. Unit Testing
6. UI Testing