Appendix C

Table of Contents

All Code

Krupas_FoodsApp.swift

```swift
import SwiftUI

@main
struct Krupas_FoodsApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .modelContainer(for: [Order.self, Product.self, Customer.self]) // This line
is used to initialize the SwiftData database with the required entities.
    }
}
```

ContentView.swift

```swift
import SwiftUI
import SwiftData

/// Primary ContentView of the app. This view facilitates the navigation functionality and provides a tabbed
interface for the app.
struct ContentView: View {
    @Query var products: [Product]

    @State private var product: Product?

    @State private var addingNewProduct = false
    @State private var managingProduct = false

    var body: some View {
        NavigationStack {
            TabView {
                if let product {
                    Group {
                        OrdersView(product: product)
                            .tabItem {
                                Label("Orders", systemImage: "shippingbox.fill")
                            }

                        if !product.isMadeToDelivery {
                            StockView(product: product)
                                .tabItem {
                                    Label("Stock", systemImage: "tray.2.fill")
                                }
                        }

                        CustomersView()
                            .tabItem {
                                Label("Customers", systemImage: "person.3.fill")
                            }

                        AnalyticsView(product: product)
                            .tabItem {
                                Label("Analytics", systemImage: "chart.bar.xaxis")
                            }
                    }
                } else {
                    // Content Unavailable view displayed when no product is
available. This is the first screen the user sees.
                    VStack {
                        ContentUnavailableView("No Product Available", systemImage:
"tag.slash.fill", description: Text("You haven't set up any products yet.\nClick
**Add Product** to get started."))
                        Button("Add Product") {
                            addingNewProduct = true
                        }
                        .buttonStyle(.borderedProminent)
                    }
                    .padding(20)
                }
            }
            .navigationBarTitleDisplayMode(.inline)
            .environment(\.horizontalSizeClass, .compact)
            .toolbarBackground(.hidden, for: .tabBar)
            .safeAreaInset(edge: .top, content: {
                // Product Switcher attached to the top. Visible only if products
are available.
                if !products.isEmpty {
                    HStack {
                        Picker("Product", selection: $product) {
                            ForEach(products) { product in
                                Group {
                                    Text("\(product.icon) \
(product.name)").foregroundStyle(.primary) + Text(Image(systemName:
"chevron.up.chevron.down")).font(.caption)
                                }
                                .tag(product as Product?)
                            }

                            Divider()

                            Label("Add New Product", systemImage: "plus")
                                .tag(nil as Product?)
                        }
                        .labelsHidden()
                        .background(.primary.opacity(0.1), in:
RoundedRectangle(cornerRadius: 10, style: .continuous))
                        .background(.ultraThickMaterial, in:
RoundedRectangle(cornerRadius: 10, style: .continuous))
                        .clipShape(.capsule)

                        Button("Edit", systemImage: "slider.horizontal.3") {
                            managingProduct = true
                        }
                        .labelStyle(.iconOnly)
                        .padding(9)
                        .background(.primary.opacity(0.1), in: .circle)
                        .background(.ultraThickMaterial, in: .circle)
                        .sheet(isPresented: $managingProduct) {
                            ManageProductsView()
                        }
                    }
                    #if targetEnvironment(macCatalyst)
```

```
                    .padding(.top)
                    #endif
                }
            })
            .onAppear {
                // Fetching the last selected product from UserDefaults. If not
found, selecting the first product.
                if let data = UserDefaults.standard.data(forKey: "currentProduct"),
let decodedID  = try? JSONDecoder().decode(UUID.self, from: data)  {
                    product = products.first { $0.id == decodedID }
                } else if let product = products.first {
                    let encodedID = try? JSONEncoder().encode(product.id)
                    UserDefaults.standard.setValue(encodedID, forKey:
"currentProduct")
                    self.product = product
                }
            }
            .onChange(of: product) { oldProduct, newProduct in
                if let newProduct {
                    // Stored to UserDefaults to persist across app launches.
                    let encodedID = try? JSONEncoder().encode(newProduct.id)
                    UserDefaults.standard.setValue(encodedID, forKey:
"currentProduct")
                } else {
                    // A value of 'nil' indicates that "Add New Product" has been
selected, so the addingNewProduct value is set to true to show the adding product
sheet.
                    addingNewProduct = true
                    self.product = oldProduct
                }
            }
            .sheet(isPresented: $addingNewProduct, onDismiss: {
                self.product = products.last
            }, content: AddProductView.init)
            .animation(.default, value: product)
        }
    }
}

#Preview {
    ContentView()
}
```

ManageProductsView.swift

```
import SwiftUI
import SwiftData
import SwipeActions

/// This view is used to manage the products, i.e., add, edit, and delete products.
struct ManageProductsView: View {
    @Environment(\.dismiss) var dismiss
    @Environment(\.modelContext) var modelContext
    @Query var products: [Product]

    @State private var addingNewProduct = false

    var body: some View {
        NavigationStack {
```

```swift
            // Scrollable list with the products
            ScrollView {
                LazyVStack {
                    ForEach(products, id: \.self) { product in
                        ProductItemView(product)
                            .transition(.asymmetric(insertion: .move(edge: .trailing
), removal: .move(edge: .leading).combined(with: .swipeDelete)))
                    }
                }
            }
            // Toolbar buttons to dismiss the view, or add a new product to the
list.
            .toolbar {
                ToolbarItem(placement: .navigationBarLeading) {
                    Button("Cancel", action: dismiss.callAsFunction)
                }

                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Add Product", systemImage: "plus.circle.fill") {
                        addingNewProduct = true
                    }
                    .sheet(isPresented: $addingNewProduct) {
                        AddProductView()
                    }
                }
            }
            .navigationTitle("Manage Products")
            .animation(.easeInOut.speed(1.75), value: products.count)
        }
    }
}
```

ProductItemView.swift

```swift
import SwiftUI
import SwipeActions

/// A view that represents a single product item in the ManageProductsView.
struct ProductItemView: View {
    @Environment(\.modelContext) var modelContext
    var product : Product

    init(_ product: Product) {
        self.product = product
    }

    @State private var isEditing = false
    @State private var showDeleteConfirmation = false

    var body: some View {
        SwipeView {
            HStack {
                Text(product.icon)
                    .font(.title)
                    .font(.largeTitle)

                VStack(alignment: .leading) {
                    Text(product.name)
                        .bold()
                }

                Spacer()
            }
            .padding(10)
            .background(.ultraThickMaterial, in: .rect(cornerRadius: 20,
style: .continuous))
        } trailingActions: { context in
            // Swipe actions for the product item.

            // Edit Button
            SwipeAction("Edit", systemImage: "pencil", backgroundColor: .blue) {
                context.state.wrappedValue = .closed
                isEditing = true
            }
            .sheet(isPresented: $isEditing) {
                // Passing the existing product into AddProductView allows for edit
functionality.
                AddProductView(product: product)
            }

            // Delete Button
            SwipeAction("Delete", systemImage: "trash", backgroundColor: .red) {
                showDeleteConfirmation = true // Show delete confirmation dialog
first instead of directly deleting the product.
            }
            .allowSwipeToTrigger()
            .foregroundStyle(.white)
            // Delete confirmation to prevent accidental deletions from swipes.
            .confirmationDialog("Confirm Deletion", isPresented:
$showDeleteConfirmation) {
                Button("Delete", role: .destructive) {
                    modelContext.delete(product)
                }

                Button("Cancel", role: .cancel) {
                    context.state.wrappedValue = .closed
                }
            } message: {
                Text("Are you sure you want to delete this product?")
            }
        }
        .swipeActionCornerRadius(20)
        .padding(.horizontal)
    }
}
```

ProductItemView.swift

```
import SwiftUI
import SwipeActions

/// A view that represents a single product item in the ManageProductsView.
struct ProductItemView: View {
    @Environment(\.modelContext) var modelContext
    var product : Product

    init(_ product: Product) {
        self.product = product
    }

    @State private var isEditing = false
    @State private var showDeleteConfirmation = false

    var body: some View {
        SwipeView {
            HStack {
                Text(product.icon)
                    .font(.title)
                    .font(.largeTitle)

                VStack(alignment: .leading) {
                    Text(product.name)
                        .bold()
                }

                Spacer()
            }
            .padding(10)
            .background(.ultraThickMaterial, in: .rect(cornerRadius: 20,
style: .continuous))
        } trailingActions: { context in
            // Swipe actions for the product item.

            // Edit Button
            SwipeAction("Edit", systemImage: "pencil", backgroundColor: .blue) {
                context.state.wrappedValue = .closed
                isEditing = true
            }
            .sheet(isPresented: $isEditing) {
                // Passing the existing product into AddProductView allows for edit
functionality.
                AddProductView(product: product)
            }

            // Delete Button
            SwipeAction("Delete", systemImage: "trash", backgroundColor: .red) {
                showDeleteConfirmation = true // Show delete confirmation dialog
first instead of directly deleting the product.
            }
            .allowSwipeToTrigger()
            .foregroundStyle(.white)
            // Delete confirmation to prevent accidental deletions from swipes.
            .confirmationDialog("Confirm Deletion", isPresented:
$showDeleteConfirmation) {
                Button("Delete", role: .destructive) {
                    modelContext.delete(product)
                }

                Button("Cancel", role: .cancel) {
                    context.state.wrappedValue = .closed
                }
            } message: {
                Text("Are you sure you want to delete this product?")
            }
        }
        .swipeActionCornerRadius(20)
        .padding(.horizontal)
    }
}
```

OrdersView.swift

```
import SwiftUI
import SwiftData

/// OrdersView is a view that displays all the orders placed for a specific product.
struct OrdersView: View {
    @Environment(\.modelContext) var modelContext
    @State private var showingNewOrderView: Bool = false

    @Query(sort: \Order.date, order: .reverse) var orders: [Order]

    var product: Product

    /// Initializes a new `OrdersView` with the specified product and fetches the orders that belong to the
specified product.
    /// - Parameter product: Pass the product for which the orders are to be displayed
    init(product: Product) {
        let id = product.id
        self._orders = Query(filter: #Predicate<Order> { order in
            return order.product?.id == id
        }, sort: \.date, order: .forward, animation: .default)

        self.product = product
    }

    /// Orders that have .isPending as true.
    var pendingOrders: [Order] {
        orders.filter { $0.isPending }
    }

    /// Orders that have .isCompleted as true.
    var completedOrders: [Order] {
        orders.filter { $0.isCompleted }
    }

    /// A namespace facilitates animations and transitions in the OrdersView.
    @Namespace var ordersSpace

    var body: some View {
        VStack {
            if orders.isEmpty {
                // Unavailability View in case no orders have been placed yet.
                ContentUnavailableView("No Orders Placed", systemImage: 
"shippingbox.fill", description: Text("Click \(Image(systemName: 
"plus.circle.fill")) to add your first order"))
                    .frame(maxHeight: .infinity, alignment: .center)
            } else {
                ScrollView {
                    LazyVStack {
                        // Shows a header with the current status of the number of 
pending and completed orders.
                        HStack {
                            VStack(spacing: 0) {
                                Text("\(pendingOrders.count)")
                                    .font(.title.bold())
                                Text("Pending")
                            }
                            .opacity(0.8)
                            .padding()
                            .frame(maxWidth: .infinity)
                            .background(.orange.gradient.opacity(0.2), 
in: .rect(cornerRadius: 20, style: .continuous))
                            .padding(2.5)

                            VStack(spacing: 0) {
                                Text("\(completedOrders.count)")
                                    .font(.title.bold())
                                Text("Completed")
                            }
                            .opacity(0.8)
                            .padding()
                            .frame(maxWidth: .infinity)
                            .background(.green.gradient.opacity(0.3), 
in: .rect(cornerRadius: 20, style: .continuous))
                            .padding(2.5)
                        }
                        .padding(.horizontal, 12.5)
                        .padding(.bottom, 2.5)

                        // Pending orders section
                        LazyVStack(pinnedViews: [.sectionHeaders]) {
                            Section {
                                ForEach(pendingOrders) { order in
                                    OrderListItem(order, namespace: ordersSpace)
                                }
                            } header: {
                                Text("Pending")
                                    .font(.title3.bold())
                                    .frame(maxWidth: .infinity, alignment: .leading)
                                    .padding(.horizontal, 12.5)
                                    .background {
                                        // VariableBlur backgrounds prevent the 
title from interfering with the orders content.
                                        VariableBlurView(maxBlurRadius: 20, 
direction: .blurredTopClearBottom)
                                            .padding(.top, -10)
                                            .frame(height: 30)
                                    }
                            }
                            .opacity(pendingOrders.isEmpty ? 0 : 1)
                            .padding(.bottom, 2.5)
                            // Completed orders section
                            Section {
                                ForEach(completedOrders) { order in
                                    OrderListItem(order, namespace: ordersSpace)
                                }
                            } header: {
                                Text("Completed")
                                    .font(.title3.bold())
                                    .frame(maxWidth: .infinity, alignment: .leading)
                                    .padding(.horizontal, 12.5)
                                    .background {
                                        // VariableBlur backgrounds prevent the
title from interfering with the orders content.
                                        VariableBlurView(maxBlurRadius: 20,
direction: .blurredTopClearBottom)
                                            .padding(.top, -10)
                                            .frame(height: 30)
                                    }
                            }
                            .opacity(completedOrders.isEmpty ? 0 : 1)
                        }
                    }
#if targetEnvironment(macCatalyst)
                    .padding(.top)
#endif
                }
                .scrollIndicators(.visible)
            }
        }
        .safeAreaInset(edge: .top, content: {
            // Title and Toolbar at the top with the Tab Title and "Add Order"
button.
            HStack {
                Text("Orders")
                    .font(.largeTitle.bold())

                Spacer()

                Button("Add Order", systemImage: "plus.circle.fill") {
                    showingNewOrderView = true
                }
                .labelStyle(.iconOnly)
                .imageScale(.large)
            }
            .padding([.horizontal, .bottom])
            .padding(.top, 40)
            .background(.bar)
        })
        .sheet(isPresented: $showingNewOrderView) {
            AddOrderView(product: product)
        }
        .animation(.easeInOut.speed(1.75), value: orders.count)
    }
}
```

CustomersView.swift

```
import SwiftUI
import SwiftData

/// A view that displays all the customers ever obtained by the business, and their order count.
struct CustomersView: View {
    @Query var customers: [Customer]
    @Environment(\.modelContext) var modelContext

    @State private var presentCannotDeleteAlert = false

    var body: some View {
        NavigationStack {
            // List with all customers ever acquired by the business.
            List {
                ForEach(customers, id: \.self) { customer in
                    // Links to AddCustomerView with the customer passed in,
allowing for the editing of customer details.
                    NavigationLink(destination: AddCustomerView(existingCustomer:
customer)) {
                        CustomerItem(customer: customer)
                    }
                }
                .onDelete(perform: deleteCustomer)
            }
            #if targetEnvironment(macCatalyst)
            .padding(.top, 65)
            #else
            .padding(.top, 50)
            #endif
            .navigationTitle("Customers")
            // Alert if the customer has existing orders and therefore cannot delete
the the customer record to prevent orphaned records.
            .alert("This customer cannot be deleted.", isPresented:
$presentCannotDeleteAlert) {
                Button("OK", role: .cancel) {
                    presentCannotDeleteAlert = false
                }
            } message: {
                Text("This customer has previously placed orders. Delete associated
orders to delete this customer.")
            }

        }
    }


    /// Check if the customer can be deleted if they have zero orders. If orders are detected, an alert
indicating the customer cannot be deleted is presented.
    /// - Parameter offsets: The offsets of the customers to be deleted from the customers array.
    private func deleteCustomer(at offsets: IndexSet) {
        for index in offsets {
            let customer = customers[index]
            if customer.wrappedOrderHistory.isEmpty {
                modelContext.delete(customer)
            } else {
                presentCannotDeleteAlert = true
            }
        }

        try? modelContext.save()
    }
}

#Preview {
    CustomersView()
}
```

OrderListItem.swift

```
import SwiftUI
import SwiftData
import SwipeActions

/// A view that represents a single order in OrdersView
struct OrderListItem: View {
    @Environment(\.modelContext) var modelContext

    var order: Order

    /// A view that represents a single order in OrdersView
    /// - Parameters:
    ///   - order: Pass in a order to display its details
    ///   - namespace: Pass in the namespace of the parent view to enable a matched geometry effect
animation.
    init(_ order: Order, namespace: Namespace.ID) {
        self.order = order
        self.namespace = namespace
        self._paymentStatus = State(initialValue: order.paymentStatus)
        self._deliveryStatus = State(initialValue: order.deliveryStatus)
    }

    @State private var showDeleteConfirmation = false
    @State private var showOrderEditView = false

    @State private var paymentStatus = Order.Status.pending
    @State private var deliveryStatus = Order.Status.pending
    @State private var showStatusChanger = false

    @State private var showBillView = false

    var namespace: Namespace.ID

    var body: some View {
        SwipeView {
            Button {
                // Tapping this view expands it to show more order details than
visible at the surface.
                withAnimation(.bouncy) {
                    showStatusChanger.toggle()
                }
            } label: {
                VStack {
                    HStack {
                        // Shows the emoji icon associated with the product for
which the order has been placed.
                        Text(order.wrappedProduct.icon)
                            .font(.largeTitle)

                        // Basic customer details are shown on the left.
                        VStack(alignment: .leading) {
                            Text(order.wrappedCustomer.name)
                                .bold()
                            Text(order.wrappedCustomer.address.line1)
                                .foregroundStyle(.secondary)
                        }

                        Spacer()

                        // If the order has any notes, a small icon is displayed to 
indicate that.
                        if !(order.notes ?? "").isEmpty {
                            Image(systemName: "text.alignright")
                                .foregroundStyle(.secondary)
                                .padding(.trailing, 5)
                        }

                        // The order count and amount paid is shown on the right.
                        VStack(alignment: .leading) {
                            // Automatic Grammar inflection is used to pluralize the 
measurement unit.
                            Text("^[\(order.quantity.formatted()) \
(order.wrappedProduct.measurementUnit.rawValue.capitalized)](inflect: true)")
                            Text(order.amountPaid, format: .currency(code: "INR"))
                                .foregroundStyle(.secondary)
                        }
                        .frame(width: 85, alignment: .leading)
                    }
                    .contentShape(Rectangle())

                    // If the user has expanded the view, more details about the 
order are shown.
                    if showStatusChanger {
                        Group {
                            Divider()

                            // Shows the payment method that the customer has 
chosen.
                            HStack {
                                Text("Payment Method:")
                                    .bold()

                                Spacer()

                                Text(order.paymentMethod.rawValue)
                            }
                            .padding([.top, .trailing], 5)

                            Divider()

                            // If order notes are not empty, they are displayed.
                            if let notes = order.notes, !notes.isEmpty {
                                Text("Notes:")
                                    .bold()
                                    .frame(maxWidth: .infinity, alignment: .leading)
                                    .padding(.top, 5)


                                Text(notes)
```

```
                                      .frame(maxWidth: .infinity, alignment: .leading)

                                  Divider()
                              }

                              // The payment and delivery statuses can be changed by 
the user.
                              LabeledContent("Payment Status") {
                                  EnumPicker(title: "Payment Status", selection: 
$paymentStatus)
                              }

                              LabeledContent("Delivery Status") {
                                  EnumPicker(title: "Delivery Status", selection: 
$deliveryStatus)
                              }
                          }
                          .padding(.leading, 10)
                          .transition(.move(edge: .top).combined(with: .blurReplace))
                          .onAppear {
                              paymentStatus = order.paymentStatus
                              deliveryStatus = order.deliveryStatus
                          }
                      }
                  }
              }
              .buttonStyle(.plain)
              .padding(10)
              .background {
                  RoundedRectangle(cornerRadius: 20, style: .continuous)
                      .fill(.ultraThickMaterial)
                      .shadow(color: .black.opacity(showStatusChanger ? 0.15 : 0), 
radius: 5, x: 0, y: 0)

              }
          } leadingActions: { context in
              // Leading Swipe action to generate an invoice/bill. Only available if 
the order has been paid for.
              if order.paymentStatus == .completed {
                  SwipeAction("Bill", systemImage: "doc.text") {
                      showBillView = true
                  }
              }
          } trailingActions: { context in
              // Trailing Swipe actions to edit or delete the order.

              // Edit Button
              SwipeAction("Edit", systemImage: "pencil") {
                  context.state.wrappedValue = .closed
                  showOrderEditView = true
              }

              // Delete Button
              SwipeAction("Delete", systemImage: "trash", backgroundColor: .red) {
                  showDeleteConfirmation = true
              }
              .allowSwipeToTrigger()
              .foregroundStyle(.white)
```

```
            .confirmationDialog("Confirm Deletion", isPresented:
$showDeleteConfirmation) {
                Button("Delete", role: .destructive) {
                    modelContext.delete(order)
                }

                Button("Cancel", role: .cancel) {
                    context.state.wrappedValue = .closed
                }
            } message: {
                Text("Are you sure you want to delete this order?")
            }
        }
        .swipeActionCornerRadius(20)
        .matchedGeometryEffect(id: order.id, in: namespace)
        .padding(.horizontal)
        .padding(.vertical, showStatusChanger ? 7.5 : 2.5)
        .transition(.asymmetric(insertion: .move(edge: .trailing),
removal: .move(edge: .leading).combined(with: .swipeDelete)))
        .sheet(isPresented: $showOrderEditView) {
            // Passing an existing order to the AddOrderView allows for it to be
edited.
            AddOrderView(order: order)
        }
        .sheet(isPresented: $showBillView) {
            BillView(order: order)
        }
        .onChange(of: paymentStatus) {
            withAnimation {
                self.order.paymentStatus = paymentStatus
            }
        }
        .onChange(of: deliveryStatus) {
            withAnimation {
                self.order.deliveryStatus = deliveryStatus
            }
        }
        .onChange(of: paymentStatus == .completed && deliveryStatus == .completed) {
            withAnimation(.bouncy) {
                showStatusChanger = false
            }
        }
    }
}
```

StockView.swift

```
import SwiftUI
import SwiftData

/// A view that displays the inventory of a product.
struct StockView: View {
    @Query var pendingStock: [PendingStock]
    @Query var stock: [Stock]

    @State private var showingAddStockView = false

    var product: Product
    /// Initializes the stock view with the given product.
    /// - Parameter product: The product whose stock is to be displayed
    init(product: Product) {
        let id = product.id
        // Fetches the stock of the product
        self._stock = Query(filter: #Predicate<Stock> { stock in
            return stock.product?.id == id
        }, sort: \.date, order: .reverse, animation: .default)

        // Fetches the backordered stock of the product
        self._pendingStock = Query(filter: #Predicate<PendingStock> { pendingStock
in
            if pendingStock.fulfilledBy != nil {
                return false
            } else if let product = pendingStock.product {
                return product.persistentModelID == product.persistentModelID
            } else {
                return false
            }
        }, sort: \.date, order: .forward)

        self.product = product
    }

    var body: some View {
        Group {
            if stock.isEmpty {
                // Displays the content unavailable view when the stock is empty.
THe backordering alert is still presented.
                VStack {
                    if !pendingStock.isEmpty {
                        pendingStockAlert()
                    }

                    ContentUnavailableView("No Available Stock", systemImage:
"tray.2.fill", description: Text("Click \(Image(systemName: "plus.circle.fill")) to
add update your inventory"))
                        .frame(maxHeight: .infinity, alignment: .center)
                }
            } else {
                ScrollView {
                    // If any backorders are pending to be fulfilled, an alert is
shown with the details.
                    if !pendingStock.isEmpty {
                        pendingStockAlert()
                    }

                    // Displays the stock items in a list.
                    LazyVStack {
                        ForEach(stock) { stockOrder in
                            StockItemView(stockOrder)
                                .padding(.horizontal)
                                .padding(.vertical, 2.5)
                        }
                    }
#if targetEnvironment(macCatalyst)
                    .padding(.top)
#endif
                }
            }
        }
        .safeAreaInset(edge: .top, content: {
            // // Title and Toolbar at the top with the Tab Title and "Add Stock"
button.
            HStack {
                Text("Stock")
                    .font(.largeTitle.bold())

                Spacer()

                Button("Add Stock", systemImage: "plus.circle.fill") {
                    showingAddStockView = true
                }
                .labelStyle(.iconOnly)
                .imageScale(.large)
            }
            .padding([.horizontal, .bottom])
            .padding(.top, 40)
            .background(.bar)
        })
        .sheet(isPresented: $showingAddStockView) {
            AddStockView(product: product)
        }
        .badge(Int(pendingStock.reduce(0) { $0 + $1.quantityToBePurchased})) // This
badge shows the total quantity of stock that is pending to be restocked, and is
presented inside the tab bar.
    }

    /// The backordering alert is shown when the stock is empty and there are pending stock with the ability
to add stock and details about pending stock.
    /// - Returns: A view containing the alert and buttons
    func pendingStockAlert() -> some View {
        VStack(alignment: .leading) {
            Text("Out of stock!")
                .bold()
                .padding(10)
                .frame(maxWidth: .infinity, alignment: .leading)
                .background(Color.red.opacity(0.2), in: Rectangle())

            VStack(alignment: .leading) {
                // Automatic grammar inflection is used to pluralize the measurement
unit.
                Text("You have ^[\(pendingStock.reduce(0) { $0 +
$1.quantityToBePurchased }.formatted()) \(product.measurementUnit.title)](inflect:
true) pending restocking for recent orders to be fulfilled.")
                    .foregroundStyle(.secondary)
                Divider()
                Button("Add Stock") {
                    showingAddStockView = true
                }
            }
            .padding([.bottom, .horizontal], 10)
        }
        .background(Color.red.opacity(0.2), in: RoundedRectangle(cornerRadius: 20))
        .clipShape(RoundedRectangle(cornerRadius: 20))
        .padding()
    }
}
```

StockItemView.swift

```swift
import SwiftUI
import SwipeActions

/// A view that represents an individual stock order in StockView
struct StockItemView: View {
    @Environment(\.modelContext) var modelContext
    var stockOrder: Stock

    /// Initialize the StockItemView with a stock order
    /// - Parameter stockOrder: The stock order which is to be displayed.
    init(_ stockOrder: Stock) {
        self.stockOrder = stockOrder
    }

    @State private var showDeleteConfirmation = false

    @State private var showingDetails = false

    /// The customer orders who have consumed stock from this stock order.
    var associatedOrders: [Order] {
        if let fulfilledStock = stockOrder.fulfillingStock {
            return stockOrder.wrappedUsedBy + fulfilledStock.filter { !
stockOrder.wrappedUsedBy.contains($0.order!) }.compactMap { $0.order }
        } else {
            return stockOrder.wrappedUsedBy
        }
    }

    var body: some View {
        SwipeView {
            VStack(alignment: .leading) {
                HStack {
                    Image(systemName: "shippingbox.fill")
                        .foregroundStyle(.yellow.gradient)
                        .font(.largeTitle)

                    VStack(alignment: .leading) {
                        // Shows the quantity of stock left and the date of the
stock order. Automatic grammar inflection is used to show the correct plural form of
the measurement unit.
                        Text("^[\(stockOrder.quantityLeft.formatted())/\
(stockOrder.quantityPurchased.formatted()) \
(stockOrder.wrappedProduct.measurementUnit.title)](inflect: true) remaining")
                            .bold()

                        // A secondary label showing the date of the stock order.
                        Text(stockOrder.date.formatted(date: .abbreviated,
time: .omitted))
                            .foregroundStyle(.secondary)
                    }

                    Spacer()
                    // The chevron indicates that this view can be expanded to show 
more details. Tapping it changes the chevron's rotation to 90 degrees to indicate 
that the view is expanded.
                    Image(systemName: "chevron.right")
                        .labelStyle(.iconOnly)
                        .foregroundStyle(.secondary)
                        .rotationEffect(.degrees(showingDetails ? 90 : 0))
                        .accessibilityHint("\(showingDetails ? "Hide" : "Show") 
Details for Stock Order on \(stockOrder.date.formatted())")
                }
                // Reduce opacity and add a strikethrough to the stock order if it 
has been fully consumed and the details are not being shown.
                .strikethrough(stockOrder.quantityLeft == 0 && !showingDetails)
                .opacity((stockOrder.quantityLeft == 0 && showingDetails) ? 0.6 : 1)

                // If the view is tapped, the further details are shown.
                if showingDetails {
                    Group {
                        Divider()

                        // The exact amount that was paid for the stock order is 
shown with the INR symbol.
                        LabeledContent("Amount Paid", value: "\
(INRFormatter.string(from: NSNumber(value: stockOrder.amountPaid)) ?? "")")

                        // If the stock order has been tied to orders placed by 
customers, the customer and order details are shown here.
                        if !stockOrder.wrappedUsedBy.isEmpty {
                            Divider()

                            Text("Associated Orders:")
                                .bold()
                                .font(.title3)

                            ForEach(stockOrder.wrappedUsedBy) { order in
                                HStack {
                                    VStack(alignment: .leading) {
                                        Text(order.wrappedCustomer.name)
                                            .bold()
                                        Text(order.amountPaid, 
format: .currency(code: "INR"))
                                            .foregroundStyle(.secondary)
                                    }

                                    Spacer()

                                    Text("\(order.quantity.formatted())")
                                }
                            }
                            .padding(5)
                        }
                    }
                    .transition(.move(edge: .top).combined(with: .blurReplace))
                }
            }
            .padding(10)
            .background(.ultraThickMaterial, in: .rect(cornerRadius: 20, 
style: .continuous))
            .clipped()
```

```swift
            .transition(.asymmetric(insertion: .move(edge: .trailing),
removal: .move(edge: .leading).combined(with: .swipeDelete)))
            .onTapGesture {
                // Tapping to reveal more details about the stock order.
                showingDetails.toggle()
            }
        } trailingActions: { context in
            // Trailing swipe action to delete the stock order.

            SwipeAction("Delete", systemImage: "trash", backgroundColor: .red) {
                showingDetails = false
                showDeleteConfirmation = true
            }
            .allowSwipeToTrigger()
            .confirmationDialog("Confirm Deletion", isPresented:
$showDeleteConfirmation) {
                Button("Delete", role: .destructive) {
                    modelContext.delete(stockOrder)
                }

                Button("Cancel", role: .cancel) {
                    showingDetails = false
                    context.state.wrappedValue = .closed
                }
            } message: {
                Text("Are you sure you want to delete this stock order?")
            }
        }
        .swipeActionCornerRadius(20)
        .animation(.bouncy, value: showingDetails)
    }
}
```

AnalyticsView.swift

```swift
import SwiftUI
import Charts
import SwiftData

/// This view displays the analytics for a specific product. It shows the revenue and profits over a period of time.
struct AnalyticsView: View {
    var product: Product
    @Query var orders: [Order]

    /// Initializes the analytics view with a specific product.
    /// - Parameter product: The product for which analytics are to be displayed.
    init(product: Product) {
        self.product = product
        let id = product.id
        self._orders = Query(
            filter: #Predicate<Order> { order in
                order.product?.id == id
            },
            sort: \.date,
            order: .forward,
            animation: .default
        )
    }
    var body: some View {
        Form {
            // Revenue section.
            Section("Revenue") {
                ChartView(orders: orders, chartType: .revenue)
            }

            // Show Profits only if data about inventory is available.
            if !product.isMadeToDelivery {
                Section("Profits") {
                    ChartView(orders: orders, chartType: .profit)
                }
            }
        }
        #if targetEnvironment(macCatalyst)
        .padding(.top, 65)
        #else
        .padding(.top, 50)
        #endif
    }
}

extension Date {
    // Function for checking if a specified date is the same as the another date.
    func isSameDay(as otherDate: Date) -> Bool {
        let calendar = Calendar.current
        return calendar.isDate(self, inSameDayAs: otherDate)
    }

    // Function for formatting the day in the MMM d format.
    var formattedMonthDay: String {
        let formatter = DateFormatter()
        formatter.dateFormat = "MMM d" // "Nov 24" format
        return formatter.string(from: self)
    }
}

struct ChartView: View {
    /// Use this enum to choose between a revenue or profit chart for ChartView.
    enum ChartType {
        case revenue
        case profit
    }

    // Parameters to be passed to the ChartView from the parent view.
    let orders: [Order]
    let chartType: ChartType

    // Date range parameters for analytics
    @State private var selectedTimeFrame: TimeFrame = .lastWeek
    @State private var startDate: Date = Date.now.addingTimeInterval(-86400 * 7)
    @State private var endDate: Date = Date.now
    @State private var currentHoverDate: Date? = nil

    /// TimeFrame enum for selecting the time frame for analytics
    enum TimeFrame: Hashable {
        case lastWeek, lastMonth, custom

        var title: String {
            switch self {
            case .lastWeek: return "7 Days"
            case .lastMonth: return "30 Days"
            case .custom: return "Custom"
            }
        }
    }

    /// Computer property for date range based on "selectedTimeFrame"
    private var dateRange: (start: Date, end: Date) {
        switch selectedTimeFrame {
        case .lastWeek:
            let start = Calendar.current.date(byAdding: .day, value: -7, to: Date())
?? Date.now
            return (start, Date.now)
        case .lastMonth:
            let start = Calendar.current.date(byAdding: .day, value: -30, to:
Date()) ?? Date.now
            return (start, Date.now)
        case .custom:
            return (startDate, endDate)
        }
    }

    // Computer property for date range array based on "dateRange"
    private var dateRangeArray: [Date] {
        let (start, end) = dateRange
        guard start <= end else { return [] }

        var dates: [Date] = []
        var current = start
        let calendar = Calendar.current

        while current <= end {
            dates.append(current)
            guard let next = calendar.date(byAdding: .day, value: 1, to: current)
else { break }
            current = next
        }
        return dates
    }

    /// Computes the total (revenue or profit) over the selected time frame.
    private var totalForTimePeriod: Double {
        orders
            .filter { $0.date >= dateRange.start && $0.date <= dateRange.end }
            .reduce(0.0) { partialResult, order in
                partialResult + (chartType == .revenue
                                   ? order.amountPaid
                                   : (order.amountPaid - order.totalCost))
            }
    }

    /// Computes an array of daily totals for use with the chart’s Y-axis.
    private var dailyValues: [Double] {
        dateRangeArray.map { date in
            let dailyOrders = orders.filter { $0.date.isSameDay(as: date) }
            return dailyOrders.reduce(0.0) { result, order in
                result + (chartType == .revenue
                            ? order.amountPaid
                            : (order.amountPaid - order.totalCost))
            }
        }
    }

    /// Determines the Y-axis domain based on the chart type.
    private var yDomain: ClosedRange<Double> {
        if chartType == .revenue {
            let maxVal = dailyValues.max() ?? 0
            return 0...(maxVal + 1000)
        } else {
            let minVal = dailyValues.min() ?? 0
            let maxVal = dailyValues.max() ?? 0
            return (minVal - 100)...(maxVal + 100)
        }
    }

    // The number of seconds remaining in the current day.
    private var secondsRemaining: Double {
        86400 - Date.now.timeIntervalSince(Calendar.current.startOfDay(for:
Date.now))
    }

    // The number of seconds that have passed in the current day.
    private var secondsPast: Double {
        Date.now.timeIntervalSince(Calendar.current.startOfDay(for: Date.now))
    }

    /// Determines the X-axis domain based on the date range.
    private var xDomain: ClosedRange<Date> {
        dateRange.start.advanced(by: -secondsPast)...dateRange.end.advanced(by:
secondsRemaining)
    }

    var body: some View {
        VStack {
            header

            GeometryReader { geo in
                Chart(dateRangeArray, id: \.self) { date in
                    // Get orders for the day.
                    let dailyOrders = orders.filter { $0.date.isSameDay(as: date) }
                    let grouped = Dictionary(grouping: dailyOrders, by:
\.paymentStatus)

                    let confirmedTotal = (grouped[.completed] ?? []).reduce(0.0)
{ result, order in
                        result + (chartType == .revenue
                                  ? order.amountPaid
                                  : (order.amountPaid - order.totalCost))
                    }

                    let unconfirmedTotal = (grouped[.pending] ?? []).reduce(0.0)
{ result, order in
                        result + (chartType == .revenue
                                  ? order.amountPaid
                                  : (order.amountPaid - order.totalCost))
                    }
                    // First bar: confirmed orders.
                    BarMark(
                        x: .value("Day", date, unit: .day),
                        y: .value(chartType == .revenue ? "Revenue" : "Profits",
confirmedTotal)
                    )
                    .foregroundStyle(chartType == .revenue
                                        ? Color.yellow.gradient
                                        : (confirmedTotal >= 0 ?
Color.green.gradient : Color.red.gradient))

                    // Second overlayed bar: unconfirmed (pending) orders.
                    BarMark(
                        x: .value("Day", date, unit: .day),
                        y: .value(chartType == .revenue ? "Revenue" : "Profits",
unconfirmedTotal)
                    )
                    .foregroundStyle(.gray.gradient)

                    // Show a hover annotation if the day matches.
                    if let currentHoverDate, currentHoverDate.isSameDay(as: date) {
                        let hoverOffset = self.hoverOffset(for: currentHoverDate,
in: xDomain, chartCount: dateRangeArray.count)

                        RuleMark(x: .value("Day", currentHoverDate, unit: .day))
                            .foregroundStyle(.gray)
                            .lineStyle(.init(lineWidth: 2, dash: [2], dashPhase: 5))
                            .annotation(position: .top) {
                                VStack(alignment: .leading) {
                                    if !(grouped[.pending] ?? []).isEmpty {
                                        HStack {
                                            Image(systemName: "circle.fill")
                                                .foregroundStyle(chartType
== .revenue
                                                                      ?
Color.yellow.gradient
                                                                      :
(confirmedTotal >= 0 ? Color.green.gradient : Color.red.gradient))
                                            Text("₹\(confirmedTotal.formatted()) \
(chartType == .revenue ? "Paid" : "Proceeds")")
                                                .bold()
                                        }
                                        HStack {
                                            Image(systemName: "circle.fill")
                                                .foregroundStyle(.gray.gradient)
                                            Text("₹\(unconfirmedTotal.formatted())
Pending")
                                                .bold()
                                        }
                                        .minimumScaleFactor(0.5)
                                    } else {
                                        Text(confirmedTotal, format: .currency(code:
"INR"))
                                            .bold()
                                    }
                                    Divider()
                                    // Automatic grammar inflection is used to
pluralize the word “Orders”.
                                    Text("^[\(dailyOrders.count) Orders](inflect:
true)")
                                        .foregroundColor(.secondary)
                                    Text(date.formatted(date: .abbreviated,
time: .omitted))
                                        .foregroundColor(.secondary)
                                }
                                .padding(10)
                                .background {
                                    RoundedRectangle(cornerRadius: 15)
                                        .foregroundColor(.invertedPrimary.opacity(0.
8))
                                        .background(.ultraThinMaterial, in:
RoundedRectangle(cornerRadius: 15))
                                        .shadow(color: .black.opacity(0.125),
radius: 2)
                                }
                                .offset(x: hoverOffset, y: unconfirmedTotal > 0 ? 80
: 60)
                            }
                    }
                }
                .chartXAxis {
                    // Grid line marks for the Chart
                    AxisMarks(values: .stride(by: .day)) { _ in
                        AxisGridLine()
                    }

                    // Stride by day for the X-axis labels.
                    AxisMarks(
                        values: .stride(
                            by: .day,
                            count: Int(ceil(Double(dateRangeArray.count) /
(geo.size.width / 80)))
                        )
                    ) { value in
                        AxisValueLabel(format: .dateTime.month().day())
                    }
                }
                .chartXScale(domain: xDomain)
                .chartYScale(domain: yDomain)
                .chartOverlay { proxy in
                    GeometryReader { _ in
                        Rectangle()
                            .fill(.clear)
                            .contentShape(Rectangle())
                            .gesture(
                                // Drag Gesture allows for hover annotation to
display specific details for a selected day.
                                DragGesture()
                                    .onChanged { value in
                                        if let day = proxy.value(atX:
value.location.x, as: Date.self) {
                                            currentHoverDate = day
                                        }
                                    }
                                    .onEnded { _ in
```

```swift
                                            currentHoverDate = nil
                                        }
                                )
                        }
                    }
                }
                .frame(height: 200)
            }
        }

    var header: some View {
        VStack {
            // Top row: total label and time frame picker.
            HStack {
                Text(totalForTimePeriod, format: .currency(code: "INR"))
                    .font(.system(.title, design: .rounded))
                    .bold()

                Spacer()

                Picker("Time Frame", selection: $selectedTimeFrame) {
                    Text("Last 7 Days").tag(TimeFrame.lastWeek)
                    Text("Last 30 Days").tag(TimeFrame.lastMonth)
                    Text("Custom").tag(TimeFrame.custom)
                }
                .labelsHidden()
            }

            // Custom date pickers appear only if "Custom" is selected.
            if selectedTimeFrame == .custom {
                HStack {
                    DatePicker("Start Date", selection: $startDate,
                               in: (orders.first?.date ??
Date.distantPast)...endDate,
                               displayedComponents: [.date])
                        .labelsHidden()

                    Spacer()

                    DatePicker("End Date", selection: $endDate,
                               in: startDate...(orders.last?.date ?? Date()),
                               displayedComponents: [.date])
                        .labelsHidden()
                }
            }
        }
    }

    /// Computes an x-offset for the hover annotation if the date is near the edges.
    private func hoverOffset(for date: Date, in domain: ClosedRange<Date>,
chartCount: Int) -> CGFloat {
        let totalDays = domain.upperBound.timeIntervalSince(domain.lowerBound) /
86400
        let thresholdDays = ceil(totalDays * 0.1)
        let timeFromStart = ceil(date.timeIntervalSince(domain.lowerBound) / 86400)
        let timeToEnd = ceil(domain.upperBound.timeIntervalSince(date) / 86400)
        let maxOffset: CGFloat = 45

        if timeFromStart <= thresholdDays {
            return maxOffset
        } else if timeToEnd <= thresholdDays {
            return -maxOffset
        } else {
            return 0
        }
    }
}
```

AddProductView.swift

```
import SwiftUI
import MCEmojiPicker

/// A view to create or edit a product entity.
struct AddProductView: View {
    @Environment(\.modelContext) var modelContext
    @Environment(\.dismiss) var dismiss

    @State private var icon: String = ""
    @State private var name: String = ""
    @State private var measurementUnit: Product.Unit = .piece
    @State private var isMadeToDelivery = false

    @State private var showEmojiPicker = false

    var product: Product?


    /// Standard initializer for using the AddProductView in Create Mode.
    init() {}

    /// Overloaded initializer for using the AddProductView in Edit Mode.
    /// - Parameter product: Pass in an existing product to edit its details.
    init(product: Product? = nil) {
        self.product = product

        if let product {
            self._icon = State(initialValue: product.icon)
            self._name = State(initialValue: product.name)
            self._measurementUnit = State(initialValue: product.measurementUnit)
            self._isMadeToDelivery = State(initialValue: product.isMadeToDelivery)
        }
    }

    var body: some View {
        NavigationStack {
            Form {
                // A section to add the product icon and name.
                Section("Details") {
                    Group {
                        if !icon.isEmpty  {
                            Text(icon)
                        } else {
                            Image(systemName: "plus.circle.dashed")
                                .foregroundStyle(.secondary)
                        }
                    }
                    .padding()
                    .font(.system(size: 100))
                    .frame(maxWidth: .infinity, maxHeight: 250, alignment: .center)
                    .onTapGesture {
                        showEmojiPicker = true
                    }
                    // Use the MCEmojiPicker to select an emoji for the product.
                    .emojiPicker(isPresented: $showEmojiPicker, selectedEmoji:
$icon)

                    TextField("Title", text: $name)
                }

                // A toggle to set if the product is made to delivery, to determine
whether to manage inventory for the product.
                Section {
                    Toggle("Made to delivery", isOn: $isMadeToDelivery)
                } footer: {
                    Text("If a product is made to delivery, you will not be able to
manage inventory.")
                }

                // A picker to select the unit of measurement for the product.
                Section {
                    Picker("Unit", selection: $measurementUnit) {
                        ForEach(Product.Unit.allCases, id: \.self) { unit in
                            Text(unit.title)
                                .tag(unit)
                        }
                    }
                } header: {
                    Text("Measurement")
                } footer: {
                    Text("This will be the unit of measurement that will be used
when placing orders for this product.")
                }
            }
            .navigationTitle(name.isEmpty ? "New Product" : name)
            .toolbar {
                // Toolbar items for the top bar.

                // Cancel button to dismiss the view.
                ToolbarItem(placement: .topBarLeading) {
                    Button("Cancel", action: dismiss.callAsFunction)
                }

                ToolbarItem(placement: .topBarTrailing) {
                    Group {
                        // If the product is being edited, show the Save button,
else show the Add button.
                        if let product {
                            Button("Save") {
                                product.name = name
                                product.icon = icon
                                product.measurementUnit = measurementUnit
                                product.isMadeToDelivery = isMadeToDelivery
                                dismiss()
                            }
                        } else {
```

```swift
                            Button("Add") {
                                let product = Product(name: name, icon: icon,
measurementUnit: measurementUnit, isMadeToDelivery: isMadeToDelivery)
                                modelContext.insert(product)
                                dismiss()
                            }
                        }
                    }
                    .bold()
                    .disabled(name.isEmpty || icon.isEmpty)
                }
            }
        }

    }
}
```

AddOrderView.swift

```swift
import SwiftUI
import SwiftData

/// A view to create or edit an order entity.
struct AddOrderView: View {
    @Query var orders: [Order]
    @Environment(\.dismiss) var dismiss
    @Environment(\.modelContext) var modelContext

    @State private var notes: String = ""

    @Query var customers: [Customer]
    @Query var stock: [Stock]

    @State private var customer: Customer?
    @State private var paymentMethod: Order.PaymentMethod = .UPI
    @State private var quantity: Double = 0.0
    @State private var amountPaid: Double = 0.0
    @State private var paymentStatus = Order.Status.pending
    @State private var deliveryStatus = Order.Status.pending

    @State private var showCustomerPicker = false
    @State private var showAddCustomerView = false
    @State private var showingSmartOrderInference = false

    @State private var showingLossAlert = false

    var toBeEditedOrder: Order? = nil

    var product: Product


    /// Standard initializer to add a new order for the specified product.
    /// - Parameter product: Pass in the product for which the order is to be placed.
    init(product: Product) {
        let id = product.id
        self._stock = Query(filter: #Predicate<Stock> { stock in
            return stock.product?.id == id
        }, sort: \.date, order: .forward, animation: .default)
        self.product = product
    }

    /// Overloaded initializer to edit an existing order.
    /// - Parameter order: Pass in the existing order to be edited.
    init(order: Order) {
        self.product = order.wrappedProduct
        self.toBeEditedOrder = order
        self._customer = State(initialValue: order.wrappedCustomer)
        self._paymentMethod = State(initialValue: order.paymentMethod)
        self._quantity = State(initialValue: order.quantity)
        self._amountPaid = State(initialValue: order.amountPaid)
        self._paymentStatus = State(initialValue: order.paymentStatus)
        self._deliveryStatus = State(initialValue: order.deliveryStatus)
        self._notes = State(initialValue: order.notes ?? "")
    }

    /// A computed property that returns the stock that will be consumed by this order.
    var usedStock: [Stock] {
        var usedStock: [Stock] = []
        var quantity = quantity
        while quantity != 0 {
            if let stockToUse = stock.first(where: { $0.quantityLeft > 0 }) {
                usedStock.append(stockToUse)
                if stockToUse.quantityLeft >= quantity {
                    break
                } else {
                    quantity -= stockToUse.quantityLeft
                }
            } else {
                break
            }
        }

        return usedStock
    }

    var body: some View {
        NavigationStack {
            Form {
                Section("Customer Information") {
                    if let customer {
                        // If a customer is selected, their details are displayed.
                        HStack {
                            VStack(alignment: .leading) {
                                Text(customer.name)
                                    .bold()
                                Text(customer.phoneNumber)
                                    .foregroundStyle(.secondary)
                                Text("^[\(customer.wrappedOrderHistory.count)
Orders](inflect: true)")
                                    .foregroundStyle(.secondary)
                            }

                            Spacer()

                            // A button to change the customer.
                            Menu {
                                menuOptions()
                            } label: {
                                Label("Change Customer", systemImage: 
"arrow.2.circlepath")
                                    .bold()
                                    .labelStyle(.iconOnly)
                                    .imageScale(.large)
                                    .padding(5)
                                    .background(.ultraThinMaterial, in: .circle)
                            }
                        }
                    } else {
                        // If no customer is selected, the user is prompted to 
choose or add a new customer.
                        Menu("Choose Customer") {
                            menuOptions()
                        }
                    }

                }

                // Section to input the quantity and amount to be paid
                Section {
                    TextField("Amount to be paid", value: $amountPaid, formatter: 
INRFormatter)
                        .keyboardType(.numberPad)

                    Stepper(value: $quantity, in: 0.0...(.infinity), step: 
product.stepAmount, format: .number) {
                        Text("\(quantity.formatted()) \
(product.measurementUnit.title)")
                    }
                } footer: {
                    // Footer to display warnings if the quantity exceeds available 
stock.
                    if product.availableStock == 0.0 {
                        Text("\(Image(systemName: "exclamationmark.triangle")) You 
do not have any stock left. You will be prompted to add stock.")
                            .foregroundStyle(.yellow)
                    } else if quantity > product.availableStock {
                        Text("\(Image(systemName: "exclamationmark.triangle")) You 
do not have enough stock left. You will be prompted to add stock.")
                            .foregroundStyle(.yellow)
                    }
                }

                // An option to add notes to the order.
                Section("Order Notes") {
                    TextField("Notes", text: $notes, axis: .vertical)
                        .lineLimit(5, reservesSpace: true)
                }

                // If the order is not being sent out for free, the payment details 
status pickers are displayed.
                if amountPaid != 0 {
                    Section("Payment Details") {
                        EnumPicker(title: "Payment Method", selection: 
$paymentMethod)
                            .transition(.opacity)
                        EnumPicker(title: "Payment Status", selection:
$paymentStatus)
                            .transition(.opacity)
                    }
                }

                // Delivery Status Details
                Section("Status") {
                    EnumPicker(title: "Delivery Status", selection: $deliveryStatus)
                }
            }
            .navigationTitle("\(toBeEditedOrder == nil ? "New" : "Edit") Order")
            .toolbar {
                // A toolbar with a cancel and save button.

                // Cancel button
                ToolbarItem(placement: .topBarLeading) {
                    Button("Cancel", action: dismiss.callAsFunction)
                }

                ToolbarItem(placement: .topBarTrailing) {
                    HStack {
                        // If the order is not being edited, it means this is a new
order being placed. In that case, the smart AI add button is displayed.
                        if toBeEditedOrder == nil {
                            Button("Smart Add", systemImage: "sparkles") {
                                showingSmartOrderInference = true
                            }
                        }

                        // Calculates the unit cost price for the stock being used
by this product.
                        let unitCostPrice = (usedStock.map(\.averageCost).max() ??
0)

                        // Determine based on whether is being edited or not the
button label
                        Button(toBeEditedOrder == nil ? "Add" : "Save") {
                            // If the user is selling at a loss, an alert is shown
before the order is placed.
                            if amountPaid/quantity < unitCostPrice {
                                showingLossAlert = true
                            } else {
                                completionAction()
                            }
                        }
                        .bold()
                        .disabled(customer == nil || quantity == 0.0)
                        .alert(isPresented: $showingLossAlert) {
                            // Alert to show the user that they are selling at a
loss with calculated cost price, loss, and break-even price. This is allowed because
sometimes the client might want to sell at a loss or send a free sample to a
customer.
                            Alert(
                                title: Text("Are you sure you want to sell at a
loss?"),
                                message: Text("""
                                Your cost price is ₹\(unitCostPrice.formatted())/\
(product.measurementUnit.title)
```

```
                                You are incurring a loss of ₹\(((unitCostPrice -
amountPaid/quantity)*quantity).formatted()) on this order.

                                You must sell at least ₹\
((unitCostPrice*quantity).formatted()) to break even.
                                """),
                                primaryButton: .cancel(),
                                secondaryButton: .destructive(Text("Yes, Continue"),
action: completionAction)
                            )
                        }
                    }
                }
            }
            .customerPicker(isPresented: $showCustomerPicker, selection: $customer)
            .sheet(isPresented: $showAddCustomerView) {
                AddCustomerView {
                    // Completion handler to assign the customer once a new customer
has been added.
                    self.customer = $0
                }
            }
            .sheet(isPresented: $showingSmartOrderInference) {
                // A sheet to show the smart order inference view.
                SmartOrderInfererenceView(product: product) { response, customer in
                    // Completion handler to assign the customer and the inferred
data to the form.

                    self.customer = customer
                    self.quantity = response.wrappedQuantity
                    self.amountPaid = response.wrappedPriceToBePaid
                    self.paymentMethod = response.wrappedPaymentMethod
                }
            }
            .animation(.default, value: amountPaid == 0)
        }
    }

    /// A function to complete the action of adding or editing an order.
    func completionAction() {
        if let toBeEditedOrder {
            // Saves changes to the original order.
            toBeEditedOrder.customer = customer
            toBeEditedOrder.paymentMethod = paymentMethod
            toBeEditedOrder.quantity = quantity
            toBeEditedOrder.amountPaid = amountPaid
            toBeEditedOrder.paymentStatus = paymentStatus
            toBeEditedOrder.deliveryStatus = deliveryStatus
            toBeEditedOrder.notes = notes

            // If the amount is 0, the payment status is marked as complete.
            if toBeEditedOrder.amountPaid == 0 {
                toBeEditedOrder.paymentStatus = .completed
            }
        } else {
            var pendingStock: PendingStock? = nil

            // Adds a backorder if the quantity exceeds the available stock.
```

```swift
            if quantity > product.availableStock {
                pendingStock = PendingStock(quantityToBePurchased: quantity - 
product.availableStock, product: product)
                modelContext.insert(pendingStock!)
            }

            // Calculates the order number for the new order
            let orderNumber = orders.reduce(0) { max($0, $1.orderNumber ?? 0) } + 1

            // Creates a new order with the given details.
            let order = Order(orderNumber: orderNumber, for: product, customer: 
customer!, paymentMethod: paymentMethod, quantity: quantity, stock: [], amountPaid: 
amountPaid, date: Date.now, paymentStatus: paymentStatus, deliveryStatus: 
deliveryStatus, notes: notes)
            modelContext.insert(order)
            pendingStock?.order = order

            // Calculates the stock that is consumed by this order.
            var usedStock: [Stock] = []
            var quantity = order.quantity
            while quantity != 0 {
                if let stockToUse = stock.first(where: { $0.quantityLeft > 0 }) {
                    usedStock.append(stockToUse)

                    stockToUse.usedBy?.append(order)
                    if stockToUse.quantityLeft >= quantity {
                        break
                    } else {
                        quantity -= stockToUse.quantityLeft
                    }
                } else {
                    break
                }
            }

            order.stock = usedStock

            if order.amountPaid == 0 {
                order.paymentStatus = .completed
            }
        }


        dismiss()
    }

    func menuOptions() -> some View {
        Group {
            Button("Choose from existing", systemImage: "person.fill.badge.plus") {
                showCustomerPicker = true
            }

            Button("Add new", systemImage: "plus") {
                showAddCustomerView = true
            }
        }
    }
}
```

GeminiHandler.swift

```swift
import Foundation
import GoogleGenerativeAI
import SwiftUI

class GeminiHandler: ObservableObject {
    enum APIKey {
      // Fetch the API key from `GenerativeAI-Info.plist`
      static var `default`: String {
          guard let filePath = Bundle.main.path(forResource: "GenerativeAI-Info",
ofType: "plist")
          else {
            fatalError("Couldn't find file 'GenerativeAI-Info.plist'.")
          }
          let plist = NSDictionary(contentsOfFile: filePath)
          guard let value = plist?.object(forKey: "API_KEY") as? String else {
            fatalError("Couldn't find key 'API_KEY' in 'GenerativeAI-Info.plist'.")
          }
          if value.starts(with: "_") {
            fatalError(
              "Follow the instructions at https://ai.google.dev/tutorials/setup to
get an API key."
            )
          }
          return value
      }
    }

    struct Response: Codable, Identifiable, Equatable {
        var id: UUID = UUID()
        var quantity: Double?
        var priceToBePaid: Double?
        var paymentMethod: Order.PaymentMethod?
        var addressLine1: String?
        var addressLine2: String?
        var city: String?
        var pincode: String?
        var customerName: String?
        var phoneNumber: String?

        /// Coding keys for decoding the JSON response.
        init(from decoder: Decoder) throws {
            let container = try decoder.container(keyedBy: CodingKeys.self)
            quantity = try container.decodeIfPresent(Double.self, forKey: .quantity)
            priceToBePaid = try container.decodeIfPresent(Double.self,
forKey: .priceToBePaid)
            paymentMethod = try container.decodeIfPresent(Order.PaymentMethod.self,
forKey: .paymentMethod)
            addressLine1 = try container.decodeIfPresent(String.self,
forKey: .addressLine1)
            addressLine2 = try container.decodeIfPresent(String.self,
forKey: .addressLine2)
            city = try container.decodeIfPresent(String.self, forKey: .city)
            pincode = try container.decodeIfPresent(String.self, forKey: .pincode)
            customerName = try container.decodeIfPresent(String.self,
forKey: .customerName)
            phoneNumber = try container.decodeIfPresent(String.self,
forKey: .phoneNumber)
```

```
            id = UUID()
        }

        var wrappedQuantity: Double {
            quantity ?? 0.0
        }

        var wrappedPriceToBePaid: Double {
            priceToBePaid ?? 0.0
        }

        var wrappedPaymentMethod: Order.PaymentMethod {
            paymentMethod ?? .UPI
        }

        var wrappedAddressLine1: String {
            addressLine1 ?? ""
        }

        var wrappedAddressLine2: String {
            addressLine2 ?? ""
        }

        var wrappedCity: String {
            city ?? ""
        }

        var wrappedPincode: String {
            pincode ?? ""
        }

        var wrappedCustomerName: String {
            customerName ?? ""
        }

        var wrappedPhoneNumber: String {
            phoneNumber ?? ""
        }
    }

    /// Infers the order details from a screenshot of a chat.
    /// - Parameters:
    ///    - product: The product for which the order is being placed. The measurement unit of the
product will be used to infer the quantity.
    ///    - image: A screenshot of the chat containing the order details.
    /// - Returns: A `Response` object containing the inferred order details from the image.
    func inferOrderDetails(for product: Product, from image: UIImage) async throws
-> Response {
        let generativeModel = GenerativeModel(
                                    name: "gemini-1.5-flash",
                                    apiKey: APIKey.default
                                )

        let prompt = """
        This is a screenshot of a chat. Contextually figure out the following
details about the chat.
        If a particular detail is not present in the chat, you can leave it empty.
        - Quantity being ordered (the number should be in \
(product.measurementUnit.title))
```

```
        - Address of the Customer
        - Price to be paid by the customer
        - The Full Name and Phone Number of the Customer
        - Mode of Payment (UPI, Cash, or Other - No other value to be here)

        And then, format it into the JSON format with the following Keys. All keys
are optional:
        - quantity (Double)
        - priceToBePaid (Double)
        - paymentMethod (UPI, Cash, or Other - No other value to be here)
        - addressLine1 (String)
        - addressLine2 (String)
        - city (String)
        - pincode (String)
        - customerName (String)
        - phoneNumber (String)

        Your output should strictly adhere to the provided JSON schema, and nothing
else should be included in your reply.
        """

        guard let jsonResponse = try await generativeModel.generateContent(prompt,
image).text else {
            throw CancellationError()
        }

        let pattern = "\\{(?:[^{}]|\\{[^{}]*\\})*\\}"

        // Extract the JSON response from the generated text using Regular
Expressions.
        if let regex = try? NSRegularExpression(pattern: pattern, options: []) {
            let range = NSRange(jsonResponse.startIndex..<jsonResponse.endIndex, in:
jsonResponse)
            if let match = regex.firstMatch(in: jsonResponse, options: [], range:
range) {
                if let matchRange = Range(match.range, in: jsonResponse) {
                    let jsonString = String(jsonResponse[matchRange])
                    print("Extracted JSON: \(jsonString)")
                    guard let data = jsonString.data(using: .utf8) else {
                        throw CancellationError()
                    }

                    do {
                        /// Decode the JSON and turn it into a `Response` object.
                        let geminiResponse = try JSONDecoder().decode(Response.self,
from: data)

                        return geminiResponse
                    } catch {
                        throw error
                    }
                }
            }
        }

        throw CancellationError()
    }
}
```

ScreenshotInferenceView.swift

```swift
import SwiftUI
import PhotosUI

/// A view for inferring order details from a screenshot.
struct ScreenshotInferenceView: View {
    @Environment(\.dismiss) var dismiss
    var product: Product
    @Binding var response: GeminiHandler.Response?

    @StateObject var geminiHandler = GeminiHandler()

    @State private var selectedItem: PhotosPickerItem? = nil
    @State private var selectedImageData: Data? = nil

    var selectedImage: UIImage? {
        if let selectedImageData, let image = UIImage(data: selectedImageData) {
            return image
        }

        return nil
    }

    @State private var isProcessing = false
    @State private var showErrorAlert = false
    @State private var isShowingPhotoPicker = false

    var body: some View {
        NavigationStack {
            VStack {
                // A large button that allows the user to select an image from the
photo library, and previews a previously selected image.
                Button {
                    isShowingPhotoPicker.toggle()
                } label: {
                    if let selectedImage {
                        Image(uiImage: selectedImage)
                            .resizable()
                            .scaledToFit()
                            .clipShape(.rect(cornerRadius: 15, style: .continuous))
                            .transition(.blurReplace)
                            .shadow(radius: 10)
                            .shimmering(active: isProcessing, bandSize: 1)
                    } else {
                        VStack {
                            RoundedRectangle(cornerRadius: 15, style: .continuous)
                                .fill(Gradient(colors:
[.purple, .blue]).opacity(0.2))
                                .frame(maxWidth: .infinity, maxHeight: .infinity)
                                .shadow(radius: 10)
                                .overlay {
                                    Image(systemName: "photo.badge.plus")
                                        .font(.title)
                                        .bold()
                                        .foregroundStyle(.gray.opacity(0.5))
                                        .foregroundStyle(.ultraThickMaterial)
                                }
                                .transition(.blurReplace)
                        }
                    }
                }
                .photosPicker(isPresented: $isShowingPhotoPicker, selection:
$selectedItem, photoLibrary: .shared())
                .disabled(isProcessing)
                .padding()
                .onChange(of: selectedItem) {
                    Task {
                        if let data = try? await
selectedItem?.loadTransferable(type: Data.self) {
                            withAnimation(.default.speed(0.5)) {
                                selectedImageData = data
                            }
                        }
                    }
                }

                if selectedImage != nil {
                    // A button to start or stop processing the selected image.
Shown only when an image is selected.
                    HStack(spacing: 20) {
                        Button("\(isProcessing ? "Stop" : "Start") Processing") {
                            if isProcessing {
                                stopProcessing()
                            } else {
                                startProcessing()
                            }
                        }
                        .foregroundStyle(.white)
                        .padding(10)
                        .background(.blue.gradient, in: .capsule)
                        .transition(.blurReplace)

                        if !isProcessing {
                            Button("Change Image", systemImage:
"arrow.2.circlepath") {
                                isShowingPhotoPicker.toggle()
                            }
                            .labelStyle(.iconOnly)
                            .font(.title2)
                            .transition(.blurReplace)
                        }
                    }
                    .padding(.bottom)
                }

                // A disclaimer about the processing of the image.
                Text("\(Image(systemName: "sparkles")) Powered by Google Gemini")
                    .bold()
                Text("Do not share any private or sensitive conversations.
Screenshots will be sent to Google Servers for processing.")
                    .font(.footnote)
                    .foregroundStyle(.secondary)
                    .multilineTextAlignment(.center)
                    .padding()
            }
            // Error handling
            .alert("An Error Ocurred", isPresented: $showErrorAlert, actions: {
```

```swift
                Button("Cancel") {
                    dismiss()
                }
            }, message: {Text("Please try again later.")})
        }
    }

    func stopProcessing() {
        isProcessing = false
    }

    /// Sending the selected image to the `GeminiHandler` for processing.
    func startProcessing() {
        isProcessing = true

        Task {
            do {
                self.response = try await geminiHandler.inferOrderDetails(for:
product, from: selectedImage!)
            } catch {
                showErrorAlert = true
                isProcessing = false
            }
        }
    }
}
```

SmartOrderInfererenceView.swift

```swift
import SwiftUI
import PhotosUI
import Shimmer

// This is the view that facilitates the smart order inference process.
struct SmartOrderInfererenceView: View {
    @Environment(\.dismiss) var dismiss
    var product: Product

    @StateObject var geminiHandler = GeminiHandler()
    @State private var response: GeminiHandler.Response? = nil

    @State private var selectedItem: PhotosPickerItem? = nil
    @State private var selectedImageData: Data? = nil

    @State private var customer: Customer?

    var selectedImage: UIImage? {
        if let selectedImageData, let image = UIImage(data: selectedImageData) {
            return image
        }

        return nil
    }

    @State private var isProcessing = false
    @State private var showErrorAlert = false
    @State private var isShowingPhotoPicker = false

    @State private var selection = 0
    var completion: (GeminiHandler.Response, Customer) -> Void

    var body: some View {
        NavigationStack {
            Group {
                if response == nil {
                    // If a response is not available, show the screenshot picker.
                    ScreenshotInferenceView(product: product, response: $response)
                        .tag(0)
                } else if let response {
                    // Once a response is available, show the customer picker.
                    SmartCustomerPicker(product: product, response: response)
{ customer in
                        self.customer = customer
                        completion(response, customer)
                        dismiss()
                    }
                    .tag(1)
                }
            }
            .toolbar {
                ToolbarItem(placement: .topBarLeading) {
                    Button("Cancel", action: dismiss.callAsFunction)
                }
            }
        }
        .onChange(of: response) {
            // Once a response is available, navigate to the customer picker.
            if response != nil {
                DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
                    selection += 1
                }
            }
        }
        .animation(.default, value: response)
        .tabViewStyle(.page)
    }
}

#Preview {
//    SmartOrderInfererenceView(product: Product())
}
```

SmartCustomerPicker.swift

```
import SwiftUI
import SwiftData

/// A view that allows the user to pick a customer from a list of customers when in Smart Inference Mode based
on the details inferred from the screenshot.
struct SmartCustomerPicker: View {
    @Environment(\.dismiss) var dismiss
    @State private var customer: Customer?

    var product: Product
    var response: GeminiHandler.Response?

    @State private var predicate: Predicate<Customer> = #Predicate { _ in true }
    @Query var customers: [Customer]

    var filteredCustomers: [Customer] {
        return (try? customers.filter(predicate)) ?? []
    }

    var completion: (Customer) -> Void

    /// Initializes the SmartCustomerPicker to display customers relevant to the inferred details.
    /// - Parameters:
    ///   - product: The product for which the order is being placed.
    ///   - response: The response from the Gemini API with the inferred details.
    ///   - completion: A completion handler that returns the selected customer.
    init(product: Product, response: GeminiHandler.Response?, completion: @escaping
(Customer) -> Void) {
        self.product = product
        self.response = response
        self.completion = completion
        if let response {
            let addressLine1 = response.wrappedAddressLine1
            let addressLine2 = response.wrappedAddressLine2
            let city = response.wrappedCity
            let customerName = response.wrappedCustomerName
            let phoneNumber = response.wrappedPhoneNumber
            let pincode = response.wrappedPincode

            // Create a predicate that filters customers based on the inferred
details.
            self._predicate = State(initialValue: #Predicate<Customer> { customer in
                return (
                    (customer.address.line1.localizedStandardContains(addressLine1)
|| customer.address.line2.localizedStandardContains(addressLine2) ||
customer.address.city.localizedStandardContains(city) ||
customer.address.pincode.localizedStandardContains(pincode) ||
customer.name.localizedStandardContains(customerName) ||
customer.phoneNumber.localizedStandardContains(phoneNumber))
                )
            })
        }
    }

    var body: some View {
        Group {
            Form {
                if filteredCustomers.isEmpty, let response {
                    // If no customers are found based on the inferred details,
allow the user to add a new customer.

                    AddCustomerView(name: response.wrappedCustomerName, phoneNumber:
response.wrappedPhoneNumber, addressLine1: response.wrappedAddressLine1,
addressLine2: response.wrappedAddressLine2, city: response.wrappedCity, pincode:
response.wrappedPincode) { customer in
                        if let customer {
                            completion(customer)
                        }
                    }
                } else {
                    // If customers are found based on the inferred details, allow
the user to choose from existing customers.

                    Picker("Choose Customer", selection: $customer) {
                        CustomersList(customer: $customer, filter: predicate)
                    }
                    .pickerStyle(.inline)
                    .navigationTitle("Choose Customer")
                }
            }
            .toolbar {
                ToolbarItem(placement: .topBarTrailing) {
                    Button("Continue") {
                        completion(customer!)
                    }
                    .disabled(customer == nil)
                }
            }
        }
    }
}
```

AddStockView.swift

```
import SwiftUI
import SwiftData

// A view for adding stock to a product.
struct AddStockView: View {
    @Environment(\.dismiss) var dismiss
    @Environment(\.modelContext) var modelContext
    @Query(sort: \PendingStock.date, order: .forward) var pendingStocks:
[PendingStock]

    @State private var amountPaid: Double = 0.0
    @State private var quantityPurchased: Double = 0.0
    @State private var quantityLeft: Double = 0.0
    @State private var date: Date = .now

    @State private var hasConsumed: Bool = false

    var product: Product

    @State private var detent: PresentationDetent = .medium

    /// Initializes the stock creation view for a product.
    /// - Parameter product: The product for which the stock is to be added.
    init(product: Product) {
        self.product = product
        self._pendingStocks = Query(filter: #Predicate<PendingStock> { pendingStock
in
            if pendingStock.fulfilledBy != nil {
                return false
            } else if let product = pendingStock.product {
                return product.persistentModelID == product.persistentModelID
            } else {
                return false
            }
        }, sort: \.date, order: .forward)
    }

    var body: some View {
        NavigationStack {
            Form {
                Section("Purchase Details") {
                    TextField("Amount Paid", value: $amountPaid, formatter: 
INRFormatter)
                        .keyboardType(.numberPad)
                    DatePicker("Purchase Date", selection: $date)
                        .datePickerStyle(.compact)
                        .frame(maxHeight: 400)
                }

                Section("Quantity Purchased") {
                    Stepper(value: $quantityPurchased, in: 0.0...(.infinity), step: 
product.stepAmount, format: .number) {
                        Text("\(quantityPurchased.formatted()) \
(product.measurementUnit.title)")
                    }

                    if !hasConsumed {
                        Toggle("Set Remaining Quantity", isOn: 
$hasConsumed.animation())
                    }
                }
                // Adjust the quantity left linearly when the quantity purchased is 
changed.
                .onChange(of: quantityPurchased) { oldValue, newValue in
                    if !hasConsumed || (quantityLeft > quantityPurchased) {
                        quantityLeft = quantityPurchased
                    } else {
                        if (newValue - oldValue) > 0 {
                            quantityLeft += newValue - oldValue
                        }
                    }
                }

                // Allow the user to set the quantity left if the stock has been 
consumed.
                if hasConsumed {
                    Section("Quantity Left") {
                        Stepper(value: $quantityLeft, in: 0.0...quantityPurchased, 
step: product.stepAmount, format: .number) {
                            Text("\(quantityLeft.formatted()) \
(product.measurementUnit.title)")
                        }
                    }
                    .onAppear {
                        detent = .large
                    }
                }

            }
            .navigationTitle("\(product.icon) Add Stock")
            .toolbar {
                // Cancel button to dismiss the view
                ToolbarItem(placement: .topBarLeading) {
```

```swift
                    Button("Cancel", action: dismiss.callAsFunction)
                }

                // Add button to save the stock.
                ToolbarItem(placement: .topBarTrailing) {
                    Button("Add") {
                        let stock = Stock(amountPaid: amountPaid, quantityPurchased: 
quantityPurchased, quantityLeft: (hasConsumed  ? quantityLeft : quantityPurchased), 
for: product)
                        modelContext.insert(stock)

                        // Fulfill pending stocks if any.
                        if !pendingStocks.isEmpty {
                            var quantityRemaining = self.quantityLeft
                            for pendingStock in pendingStocks {
                                if quantityRemaining > 0 && 
(pendingStocks.reduce(0.0) { $0 + $1.quantityToBePurchased } > 0) {
                                    if pendingStock.quantityToBePurchased > 
quantityRemaining {
                                        pendingStock.quantityToBePurchased -= 
quantityRemaining
                                        quantityRemaining = 0
                                    } else {
                                        quantityRemaining -= 
pendingStock.quantityToBePurchased
                                        pendingStock.fulfilledBy = stock
                                    }
                                }
                            }
                        }

                        dismiss()
                    }
                    .bold()
                    .disabled(quantityPurchased == 0.0)
                }
            }
        }
        .presentationDetents([.medium, .large], selection: $detent)
    }
}
```

AddCustomerView.swift

```swift
import SwiftUI
import Contacts

/// A view for adding or editing a new customer.
struct AddCustomerView: View {
    @Environment(\.modelContext) var modelContext
    @Environment(\.dismiss) var dismiss

    @State private var name: String = ""
    @State private var phoneNumber = ""

    @State private var addressLine1 = ""
    @State private var addressLine2 = ""
    @State private var pincode: String = ""
    @State private var city = ""
    @State var contact: CNContact?
    @State private var existingCustomer: Customer? = nil

    /// Code for controlling the currently focused field programmatically.
    @FocusState private var focusedField: Field?
    enum Field: Int, Hashable {
        case name = 1
        case number = 2
        case addressLine1 = 3
        case addressLine2 = 4
        case city = 5
        case pincode = 6
    }

    /// A completion handler that returns the newly created or edited customer.
    var completion: (Customer?) -> Void

    /// Standard initializer for using the AddCustomerView in Create Mode.
    init(completion: @escaping (Customer?) -> Void) {
        self.completion = completion
    }

    /// Overloaded initializer for using the AddCustomerView in Smart Inference Mode with Pre-filled Details.
    init (name: String, phoneNumber: String, addressLine1: String, addressLine2: 
String, city: String, pincode: String, completion: @escaping (Customer?) -> Void) {
        self._name = State(initialValue: name)
        self._phoneNumber = State(initialValue: phoneNumber)
        self._addressLine1 = State(initialValue: addressLine1)
        self._addressLine2 = State(initialValue: addressLine2)
        self._city = State(initialValue: city)
        self._pincode = State(initialValue: pincode)
        self.completion = completion
    }

    /// Overloaded initializer for using the AddCustomerView in Edit Mode.
    init(existingCustomer: Customer, completion: @escaping (Customer?) -> Void = { _ 
in }) {
        self._name = State(initialValue: existingCustomer.name)
        self._phoneNumber = State(initialValue: existingCustomer.phoneNumber)
        self._addressLine1 = State(initialValue: existingCustomer.address.line1)
        self._addressLine2 = State(initialValue: existingCustomer.address.line2)
        self._city = State(initialValue: existingCustomer.address.city)
        self._pincode = State(initialValue: existingCustomer.address.pincode)
        self.existingCustomer = existingCustomer
        self.completion = completion
    }

    @State private var showingExistingCustomerPicker = false

    var body: some View {
        NavigationStack {
            Form {
                if existingCustomer == nil {
                    Section {
                        // A button to import details from the user's system 
contacts.
                        ContactPickerButton(contact: $contact) {
                            Label("Import Details from Contacts", systemImage:
"book.closed.fill")
                                .frame(maxWidth: .infinity, alignment: .leading)
                        }

                        // A button to choose an existing customer stored in the
app.
                        Button {
                            showingExistingCustomerPicker = true
                        } label: {
                            HStack {
                                Image(systemName: "person.2.fill")
                                Text("Choose from Existing Customers")
                            }
                        }
                        .navigationDestination(isPresented:
$showingExistingCustomerPicker) {
                            ExistingCustomerPicker(
                                customer: Binding(
                                    get: { nil },
                                    set: {
                                        showingExistingCustomerPicker = false
                                        completion($0)
                                        dismiss()
                                    }
                                ),
                                style: .navigation
                            )
                        }
                    }
                }

                Section(header: Text("Customer Details")) {
                    TextField("Name", text: $name)
                        .submitLabel(.next)
                        .focused($focusedField, equals: .name)
                        .onSubmit(submitAction)

                    TextField("Phone Number", text: $phoneNumber)
                        .submitLabel(.next)
                        .keyboardType(.phonePad)
                        .focused($focusedField, equals: .number)
                        .onSubmit(submitAction)
                }

                Section(header: Text("Address")) {
                    TextField("Line 1", text: $addressLine1)
                        .submitLabel(.next)
                        .focused($focusedField, equals: .addressLine1)
                        .onSubmit(submitAction)

                    TextField("Line 2", text: $addressLine2)
                        .submitLabel(.next)
                        .focused($focusedField, equals: .addressLine2)
                        .onSubmit(submitAction)

                    TextField("City", text: $city)
                        .submitLabel(.next)
                        .focused($focusedField, equals: .city)
                        .onSubmit(submitAction)

                    TextField("Pincode", text: $pincode)
                        .keyboardType(.numberPad)
                        .submitLabel(.done)
                        .focused($focusedField, equals: .pincode)
                        .onSubmit(submitAction)
                }
            }
            // Determine the title of the navigation bar based on whether the
customer is new or existing.
            .navigationTitle("\(existingCustomer == nil ? "New" : "Edit") Customer")
            .toolbar {
                ToolbarItemGroup(placement: .topBarLeading) {
                    Button("Cancel", action: dismiss.callAsFunction)
                }

                ToolbarItemGroup(placement: .topBarTrailing) {
                    Group {
                        // If the customer is being edited, show the Save button,
else show the Add button.
                        if let existingCustomer {
                            Button("Save") {
                                existingCustomer.name = name
                                existingCustomer.phoneNumber = phoneNumber
                                existingCustomer.address.line1 = addressLine1
                                existingCustomer.address.line2 = addressLine2
                                existingCustomer.address.city = city
                                existingCustomer.address.pincode = pincode

                                completion(existingCustomer)
                                dismiss()
                            }
                        } else {
                            Button("Add") {
                                let address = Address(line1: addressLine1, line2:
addressLine2, city: city, pincode: pincode)
                                let customer = Customer(name: name, phoneNumber:
phoneNumber, address: address)

                                modelContext.insert(customer)
                                completion(customer)
                                dismiss()
                            }
                        }
                    }
                    // Disable if any of the required fields are empty.
                    .disabled(name.trimmingCharacters(in: .whitespacesAndNewlines).i
sEmpty || phoneNumber.trimmingCharacters(in: .whitespacesAndNewlines).count < 9 ||
addressLine1.isEmpty || city.isEmpty)
                    .bold()
                }

            }
        }
        .onChange(of: contact) {
            // If a system contact is selected, fill the details from the contact
            if let contact {
                importDetails(from: contact)
```

```swift
            }
        }
    }

    /// Function to handle the submission of the form fields and switching of focused fields.
    func submitAction() {
        if focusedField == .pincode {
            focusedField = nil
        } else if let field = focusedField {
            focusedField = Field(rawValue: field.rawValue + 1)
        }
    }

    /// Function to import details from a system `CNContact` object.
    /// - Parameter contact: System `CNContact` object to import details
    func importDetails(from contact: CNContact) {
        self.name = (contact.givenName + " " + contact.familyName)
                        .trimmingCharacters(in: .whitespacesAndNewlines)

        if let phoneNumber = contact.phoneNumbers.first?.value.stringValue {
            self.phoneNumber = phoneNumber
        } else {
            focusedField = .number
        }

        if let address = contact.postalAddresses.first?.value {
            self.addressLine1 = address.street
            self.addressLine2 = address.subLocality
            self.pincode = address.postalCode
            self.city = address.city
        } else if !self.phoneNumber.isEmpty {
            focusedField = .addressLine1
        }
    }
}

#Preview {
    AddCustomerView() { _ in

    }
}
```

ExistingCustomerPicker.swift

```swift
import SwiftUI
import SwiftData

/// A view that allows the user to pick from existing customers.
struct ExistingCustomerPicker: View {
    @Environment(\.dismiss) var dismiss

    @Query var customers: [Customer]

    @Binding var customer: Customer?

    @State private var searchTerm: String = ""

    /// The style of the picker. This adapts the UI to match the style of the parent view.
    enum Style {
        case navigation
        case sheet
    }

    var style: Style = .sheet

    var body: some View {
        NavigationStack {
            Group {
                if customers.isEmpty {
                    // Content Unavailable View if no customers are available.
                    VStack {
                        Spacer()

                        ContentUnavailableView("No Customers", systemImage: 
"person.2.slash.fill", description: Text("You don't have any customers, yet."))

                        Button("Done", action: dismiss.callAsFunction)
                            .buttonStyle(.borderedProminent)

                        Spacer()
                    }
                } else {
                    // Form with picker to choose from existing customers.
                    Form {
                        Picker("Choose Customer", selection: $customer) {
                            CustomersList(customer: $customer, searchTerm: 
searchTerm)
                        }
                        .pickerStyle(.inline)
                    }
                }
            }
            // Searchable to select for a partocular customer.
            .searchable(text: $searchTerm, prompt: "Search for an existing 
customer...")
            .navigationTitle("Choose Customer")
            .navigationBarTitleDisplayMode(.inline)
            .toolbar {
                if style == .sheet {
                    ToolbarItem(placement: .topBarLeading) {
                        Button("Cancel", action: dismiss.callAsFunction)
                    }
                }
            }
            .onChange(of: customer, dismiss.callAsFunction)
        }
    }
}

struct CustomersList: View {
    @Binding var customer: Customer?

    @Query var customers: [Customer]

    /// List of customers that match the search term.
    /// - Parameters:
    ///   - customer: The customer that is to be selected.
    ///   - searchTerm: The search term to filter the customers by.
    init(customer: Binding<Customer?>, searchTerm: String) {
        self._customer = customer
        self._customers = Query(filter: #Predicate {
                if searchTerm.isEmpty {
                    return true
                } else {
                    // Finds any similarity between the search term and the
customer's name, phone number, address line 1, address line 2, city, or pincode.
                    return $0.name.localizedStandardContains(searchTerm) ||
$0.phoneNumber.localizedStandardContains(searchTerm) ||
$0.address.line1.localizedStandardContains(searchTerm) ||
$0.address.line2.localizedStandardContains(searchTerm) ||
$0.address.city.localizedStandardContains(searchTerm) ||
$0.address.pincode.localizedStandardContains(searchTerm)
                }
            }
        )
    }

    /// List of customers with a predicate.
    /// - Parameters:
    ///   - customer: The customer that is to be selected.
    ///   - filter: The predicate to filter the customers by.
    init(customer: Binding<Customer?>, filter: Predicate<Customer>) {
        self._customer = customer
        self._customers = Query(filter: filter)
    }

    var body: some View {
        ForEach(customers, id: \.self) {
            CustomerItem(customer: $0)
        }
    }
}

/// A view that displays a an individual customers details in the CustomersList
struct CustomerItem: View {
    var customer: Customer

    var body: some View {
        HStack {
            VStack(alignment: .leading) {
                Text(customer.name)
                    .bold()

                ViewThatFits {
                    Text([customer.address.line1, customer.address.line2,
customer.address.city, customer.address.pincode]
                        .compactMap { $0 }
                        .joined(separator: ", "))
                        .lineLimit(1)

                    Text([customer.address.line1, customer.address.line2,
customer.address.city]
                        .compactMap { $0 }
                        .joined(separator: ", "))
                        .lineLimit(1)
                    Text([customer.address.line1, customer.address.line2]
                        .compactMap { $0 }
                        .joined(separator: ", "))
                        .lineLimit(1)

                    Text(customer.address.line1)
                        .lineLimit(1)
                }
            }

            Spacer()

            Text("^[\(customer.wrappedOrderHistory.count) Orders](inflect: true)")
                .foregroundStyle(.secondary)
        }
        .tag(customer as Customer?)
    }
}

#Preview {
    ExistingCustomerPicker(customer: .constant(nil))
}

extension View {
    /// An extension to present the ExistingCustomerPicker as a sheet. This is a convenience method to
present the picker as a sheet.
    /// - Parameters:
    ///   - isPresented: A binding to a Boolean value that determines whether to present the customer
picker sheet.
    ///   - selection: A binding to the selected customer.
    func customerPicker(isPresented: Binding<Bool>, selection: Binding<Customer?>)
-> some View {
        self
            .sheet(isPresented: isPresented) {
                ExistingCustomerPicker(customer: selection)
            }
    }
}
```

Product.swift

```
import Foundation
import SwiftData

@Model
class Product {
    var id: UUID = UUID()
    var name: String = ""
    var icon: String = ""
    var measurementUnit: Unit = Unit.piece
    @Relationship(deleteRule: .cascade, inverse: \Order.product) var orders:
[Order]? = []
    @Relationship(deleteRule: .cascade, inverse: \Stock.product) var stock: [Stock]?
= []
    @Relationship(deleteRule: .cascade, inverse: \PendingStock.product) var
pendingStock: [PendingStock]? = []
    var isMadeToDelivery: Bool = false
    var stepAmount: Double = 1.0
```

```swift
    var wrappedOrders: [Order] {
        orders ?? []
    }

    var wrappedStock: [Stock] {
        stock ?? []
    }
    /// A computed property that returns the total available stock by summing the quantity left in each stock
entry.
    var availableStock: Double {
        return wrappedStock.reduce(0.0) { totalStock, item in
            totalStock + item.quantityLeft
        }
    }

    /// Enum representing the unit of measurement for the product.
    enum Unit: String, CaseIterable, Codable {
        case kg, g, dozen, box, piece

        /// A computed property that returns a display-friendly title for the unit.
        var title: String {
            switch(self) {
            case .kg, .g:
                return self.rawValue
            default:
                return self.rawValue.capitalized
            }
        }
    }

    /// Initializes a new `Product` instance.
    ///
    /// - Parameters:
    ///   - id: A unique identifier for the product. Defaults to a new UUID if not provided.
    ///   - name: The name of the product.
    ///   - icon: The icon or image representation of the product.
    ///   - measurementUnit: The unit of measurement used for this product (e.g., kg, dozen, piece).
    ///   - stepAmount: The step amount to adjust product quantities, with a default of 1.0.
    ///   - orders: The list of `Order` objects related to this product. Defaults to an empty array.
    ///   - stock: The list of `Stock` entries for this product. Defaults to an empty array.
    ///   - isMadeToDelivery: A Boolean flag indicating if the product is made specifically for delivery.
    init(id: UUID = UUID(), name: String, icon: String, measurementUnit: Unit,
         stepAmount: Double = 1.0, orders: [Order] = [], stock: [Stock] = [],
         isMadeToDelivery: Bool) {
        self.id = id
        self.name = name
        self.icon = icon
        self.measurementUnit = measurementUnit
        self.stepAmount = stepAmount
        self.orders = orders
        self.stock = stock
        self.isMadeToDelivery = isMadeToDelivery
    }
}
```

Order.swift

```swift
import SwiftData
import Foundation

@Model
class Order {
    var id: UUID = UUID()
    var orderNumber: Int?
    var product: Product?
    var customer: Customer?
    var paymentMethod: PaymentMethod = PaymentMethod.UPI
    var quantity: Double = 0.0
    var amountPaid: Double = 0.0
    var date: Date = Date.now
    var paymentStatus: Status = Status.pending
    var deliveryStatus: Status = Status.pending
    var notes: String?
    var stock: [Stock]? = []
    @Relationship(deleteRule: .cascade, inverse: \PendingStock.order) var
pendingStock: PendingStock?

    /// Computed property that returns the stock associated with the order, or an empty array if not found
    var wrappedStock: [Stock] {
        stock ?? []
    }

    /// Computed property that returns the product associated with the order, or a default product if not
    var wrappedProduct: Product {
        product ?? Product(name: "Unknown Product", icon: "❓",
measurementUnit: .piece, isMadeToDelivery: false)
    }

    /// Computed property that returns the customer associated with the order, or a default customer if not
found
    var wrappedCustomer: Customer {
        customer ?? Customer(name: "Unknown Customer", phoneNumber: "Unknown Phone
Number", address: Address())
    }

    /// A computed property that returns `true` if either the delivery or payment status is pending.
    var isPending: Bool {
        self.deliveryStatus == .pending || self.paymentStatus == .pending
    }

    /// A computed property that returns `true` if both the delivery and payment statuses are completed.
    var isCompleted: Bool {
        self.deliveryStatus == .completed && self.paymentStatus == .completed
    }

    /// A computed property that returns the total cost of the order based on the average cost of the stock
and the quantity ordered.
    var totalCost: Double {
        if let stock {
            return stock.reduce(0.0) { $0 + ($1.averageCost*($1.wrappedUsedBy.first{
$0 == self }?.quantity ?? 0))}
        } else {
            return 0
        }
    }
    /// Represents the status of an order or payment, either pending or completed.
    enum Status: String, CaseIterable, Codable {
        case pending = "Pending"
        case completed = "Completed"
    }

    /// Enum to represent the various payment methods for an order.
    enum PaymentMethod: String, CaseIterable, Codable {
        case cash = "Cash"
        case UPI = "UPI"
        case other = "Other"
    }

    /// Initializes a new `Order` instance.
    ///
    /// - Parameters:
    ///   - id: A unique identifier for the order. Defaults to a new UUID if not provided.
    ///   - product: The `Product` associated with the order.
    ///   - customer: The `Customer` who placed the order. This is an optional value.
    ///   - paymentMethod: The method of payment used for the order (e.g., Cash, UPI).
    ///   - quantity: The quantity of the product ordered.
    ///   - stock: The list of `Stock` entries related to this order.
    ///   - amountPaid: The total amount paid for the order.
    ///   - date: The date the order was placed. Defaults to the current date if not provided.
    ///   - paymentStatus: The current status of the payment (either pending or completed).
    ///   - deliveryStatus: The current status of the delivery (either pending or completed).
    init(id: UUID = UUID(), orderNumber: Int, for product: Product, customer:
Customer,
        paymentMethod: PaymentMethod, quantity: Double, stock: [Stock],
        amountPaid: Double, date: Date = .now,
         paymentStatus: Status, deliveryStatus: Status, notes: String?) {
        self.orderNumber = orderNumber
        self.id = id
        self.product = product
        self.customer = customer
        self.paymentMethod = paymentMethod
        self.quantity = quantity
        self.amountPaid = amountPaid
        self.date = date
        self.paymentStatus = paymentStatus
        self.deliveryStatus = deliveryStatus
        self.stock = stock
        self.notes = notes
    }
}
```

Stock.swift

```
import Foundation
import SwiftData

@Model
class Stock {
    var id: UUID = UUID()
    var amountPaid: Double = 0.0
    var quantityPurchased: Double = 0.0
    var manuallyConsumedQuantity: Double = 0.0
```

```swift
    var date: Date = Date.now
    var product: Product?
    @Relationship(inverse: \Order.stock) var usedBy: [Order]? = []
    @Relationship(deleteRule: .nullify,inverse: \PendingStock.fulfilledBy) var
fulfillingStock: [PendingStock]? = []

    /// Computed property that returns the product associated with the stock, or a default product if not
found
    var wrappedProduct: Product {
        product ?? Product(name: "Unknown Product", icon: "❓",
measurementUnit: .piece, isMadeToDelivery: false)
    }

    /// Computed property that returns the orders that have used this stock, or an empty array if not
    var wrappedUsedBy: [Order] {
        usedBy ?? []
    }

    /// Computed property that returns the average cost of the stock based on the total amount paid and
quantity purchased.
    var averageCost: Double {
        if quantityPurchased == 0 {
            return 0
        } else {
            return amountPaid / quantityPurchased
        }
    }

    /// Computed property that returns the remaining quantity of the stock after accounting for sales or usage.
    var quantityLeft: Double {
        let subtractedQuantity = quantityPurchased
            - self.manuallyConsumedQuantity
            - self.wrappedUsedBy.reduce(0.0) { total, order in
                total + order.quantity
            }
            - (self.fulfillingStock?.filter { pendingStock in
                !self.wrappedUsedBy.contains { $0.persistentModelID ==
pendingStock.order?.persistentModelID }
            } ?? []).reduce(0.0) { total, pendingStock in
                total + pendingStock.quantityToBePurchased
            }


        if subtractedQuantity < 0 {
            return 0
        } else {
            return subtractedQuantity
        }
    }

    /// Initializes a new `Stock` instance with specified values for all properties.
    ///
    /// - Parameters:
    ///   - id: A unique identifier for the stock. Defaults to a new UUID if not provided.
    ///   - amountPaid: The total amount paid for the purchased stock.
    ///   - quantityPurchased: The total quantity of the product that was purchased.
    ///   - quantityLeft: The remaining quantity of the product in stock after sales or usage.
    ///     - date: The date the stock was purchased or recorded. Defaults to the current date if not
provided.
    ///     - product: The `Product` this stock entry is associated with.
    ///
    /// This initializer allows you to specify all properties, including `quantityLeft` which may be different
from `quantityPurchased` if some stock has already been used or sold.
    init(id: UUID = UUID(), amountPaid: Double, quantityPurchased: Double,
quantityLeft: Double, date: Date = Date.now, for product: Product) {
        self.id = id
        self.amountPaid = amountPaid
        self.quantityPurchased = quantityPurchased
        self.manuallyConsumedQuantity = quantityPurchased - quantityLeft
        self.date = date
        self.product = product
        self.usedBy = []
    }

    /// Initializes a new `Stock` instance with `quantityLeft` automatically set to the value of
`quantityPurchased`.
    ///
    /// - Parameters:
    ///     - id: A unique identifier for the stock. Defaults to a new UUID if not provided.
    ///     - amountPaid: The total amount paid for the purchased stock.
    ///     - quantityPurchased: The total quantity of the product that was purchased.
    ///     - date: The date the stock was purchased or recorded. Defaults to the current date if not
provided.
    ///     - product: The `Product` this stock entry is associated with.
    ///
    /// This initializer automatically sets `quantityLeft` to the same value as `quantityPurchased`,
assuming no stock has been used or sold at the time of initialization.
    init(id: UUID = UUID(), amountPaid: Double, quantityPurchased: Double, date:
Date = Date.now, for product: Product) {
        self.id = id
        self.amountPaid = amountPaid
        self.quantityPurchased = quantityPurchased
        self.manuallyConsumedQuantity = 0
        self.date = date
        self.product = product
        self.usedBy = []
    }
}
```

PendingStock.swift

```swift
import Foundation
import SwiftData


@Model
class PendingStock: Identifiable {
    var id: UUID = UUID()
    var quantityToBePurchased: Double = 0.0
    var date: Date = Date.now
    var product: Product?
    var order: Order?
    var fulfilledBy: Stock?
    /// Computed property that returns the product associated with the pending stock, or a default product if
not
    var wrappedProduct: Product {
        product ?? Product(name: "Unknown Product", icon: "❓",
measurementUnit: .piece, isMadeToDelivery: false)
    }

    ///
    /// - Parameters:
    ///   - id: A unique identifier for the backorder. Defaults to a new UUID if not provided.
    ///   - quantityToBePurchased: The quantity of the product that needs to be purchased.
    ///   - date: The date when the backorder was created. Defaults to the current date and time.
    ///   - product: The product for which the backorder is placed.
    ///   - order: The order for which the backorder is placed.
    init(id: UUID = UUID(), quantityToBePurchased: Double, date: Date = Date.now,
product: Product? = nil, order: Order? = nil) {
        self.id = id
        self.quantityToBePurchased = quantityToBePurchased
        self.date = date
        self.product = product
        self.order = order
    }
}
```

Customer.swift

```swift
import Foundation
import SwiftData

@Model
class Customer {
    var id: UUID = UUID()
    var name: String = ""
    var phoneNumber: String = ""
    var address: Address = Address()
    @Relationship(inverse: \Order.customer) var orderHistory: [Order]? = []

    /// Computed property that returns the customer's order history, or an empty array if not found
    var wrappedOrderHistory: [Order] {
        orderHistory ?? []
    }

    /// Initializes a new `Customer` instance.
    ///
    /// - Parameters:
    ///   - id: A unique identifier for the customer. Defaults to a new UUID if not provided.
    ///   - name: The name of the customer.
    ///   - phoneNumber: The customer's contact phone number.
    ///   - address: The `Address` where the customer resides or is located.
    ///   - orderHistory: A list of `Order` objects representing the customer's previous orders.
Defaults to an empty array.
    init(id: UUID = UUID(), name: String, phoneNumber: String, address: Address,
orderHistory: [Order] = []) {
        self.id = id
        self.name = name
        self.phoneNumber = phoneNumber
        self.address = address
        self.orderHistory = orderHistory
    }
}
```

Address.swift

```swift
import Foundation

/// A struct representing a customer's address.
struct Address: Codable {
    var line1: String
    var line2: String
    var city: String
    var pincode: String

    /// Initializes a new `Address` instance.
    ///
    /// - Parameters:
    ///   - line1: The first line of the address (e.g., street name and number).
    ///   - line2: The second line of the address (e.g., apartment or suite number).
    ///   - city: The city where the address is located.
    ///   - pincode: The postal code or ZIP code of the address.
    init(line1: String = "", line2: String = "", city: String = "", pincode: String
= "") {
        self.line1 = line1
        self.line2 = line2
        self.city = city
        self.pincode = pincode
    }
}
```

ContactPickerButton.swift

```swift
//  Code from: https://gist.github.com/seanwoodward/e35e3fb29b5a69a37860beb50c22f5fc
Accessed 11/08/24.

import Foundation
import SwiftUI
import Contacts
import ContactsUI

struct ContactPickerButton<Label: View>: UIViewControllerRepresentable {
    class Coordinator: NSObject, CNContactPickerDelegate {
        var onCancel: () -> Void
        var viewController: UIViewController = .init()
        var picker = CNContactPickerViewController()

        @Binding var contact: CNContact?

        // Possible take a binding
        init<Label: View>(contact: Binding<CNContact?>, onCancel: @escaping () ->
Void, @ViewBuilder content: @escaping () -> Label) {
            self._contact = contact
            self.onCancel = onCancel
            super.init()
            let button = Button<Label>(action: showContactPicker, label: content)
            let hostingController: UIHostingController<Button<Label>> =
UIHostingController(rootView: button)

            hostingController.view?.backgroundColor = .clear
            hostingController.view?.sizeToFit()

            (hostingController.view?.frame).map {
                hostingController.view!.widthAnchor.constraint(equalToConstant:
$0.width).isActive = true
                hostingController.view!.heightAnchor.constraint(equalToConstant:
$0.height).isActive = true
                viewController.preferredContentSize = $0.size
            }

            hostingController.willMove(toParent: viewController)
            viewController.addChild(hostingController)
            viewController.view.addSubview(hostingController.view)

            hostingController.view.anchor(to: viewController.view)

            picker.delegate = self

        }

        func showContactPicker() {
            viewController.present(picker, animated: true)
        }

        func contactPickerDidCancel(_ picker: CNContactPickerViewController) {
            onCancel()
        }

        func contactPicker(_ picker: CNContactPickerViewController, didSelect
contact: CNContact) {
            self.contact = contact
        }

        func makeUIViewController() -> UIViewController {
            return viewController
        }

        func updateUIViewController(_ uiViewController: UIViewController, context:
UIViewControllerRepresentableContext<ContactPickerButton>) {
        }
    }

    @Binding var contact: CNContact?

    @ViewBuilder
    var content: () -> Label

    var onCancel: () -> Void

    init(contact: Binding<CNContact?>, onCancel: @escaping () -> () = {},
@ViewBuilder content: @escaping () -> Label) {
        self._contact = contact
        self.onCancel = onCancel
        self.content = content
    }
    func makeCoordinator() -> Coordinator {
        .init(contact: $contact, onCancel: onCancel, content: content)
    }

    func makeUIViewController(context: Context) -> UIViewController {
        context.coordinator.makeUIViewController()
    }

    func updateUIViewController(_ uiViewController: UIViewController, context:
Context) {
        context.coordinator.updateUIViewController(uiViewController, context:
context)
    }

}

fileprivate extension UIView {
    func anchor(to other: UIView) {
        self.translatesAutoresizingMaskIntoConstraints = false

        self.topAnchor.constraint(equalTo: other.topAnchor).isActive = true
        self.bottomAnchor.constraint(equalTo: other.bottomAnchor).isActive = true
        self.leadingAnchor.constraint(equalTo: other.leadingAnchor).isActive = true
        self.trailingAnchor.constraint(equalTo: other.trailingAnchor).isActive =
true
    }
}
```

EnumPicker.swift

```
import Foundation
import SwiftUI

/// Custom Picker for Enumerations. The Enum must conform to `RawRepresentable`, `CaseIterable`,
`Codable` and `Hashable`. This allows for selection of Enum values in a Picker based on all values of the
Enum.
struct EnumPicker<T: RawRepresentable & CaseIterable & Codable & Hashable>: View
where T.AllCases: RandomAccessCollection, T.RawValue == String {
    let title: String
    @Binding var selection: T

    var body: some View {
        Picker(title, selection: $selection) {
            ForEach(T.allCases, id: \.self) { option in
                Text(option.rawValue)
                    .tag(option)
            }
        }
    }
}
```

INRFormatter.swift

```
import SwiftUI

/// A number formatter that formats numbers as Indian Rupees.
```

```swift
var INRFormatter: NumberFormatter {
    let formatter = NumberFormatter()
    formatter.numberStyle = .currency
    formatter.currencyCode = "INR"
    formatter.maximumFractionDigits = 0
    formatter.locale = Locale(identifier: "en_IN")
    return formatter
}

import SwiftUI

/// A view that displays and generates the bill/invoice for a given order.
struct BillView: View {
    @Environment(\.dismiss) var dismiss

    /// The order for which the bill is to be generated.
    var order: Order

    var bill: some View {
        VStack(alignment: .leading, spacing: 16) {
            // Header section with the brand icon and details.
            VStack(alignment: .center, spacing: 2) {
                Image("BrandIcon")
                    .resizable()
                    .scaledToFit()
                    .frame(height: 55)

                Text("Krupa's Foods")
                    .font(.title)
                    .bold()

                Text("XYZ Street, ABC City, 123456")
                    .font(.subheadline)

                Text("+91 12345 67890")
                    .font(.subheadline)
            }
            .frame(maxWidth: .infinity)

            Divider()

            // Invoice details section.
            HStack {
                VStack(alignment: .leading, spacing: 4) {
                    Text("INVOICE")
                        .font(.headline)

                    if let orderNumber = order.orderNumber {
                        Text("Invoice No.: \(String(format: "%03d", orderNumber))")
                            .font(.subheadline)
                    }

                    Text("Date: \(order.date.formatted(date: .abbreviated,
time: .omitted))")
                        .font(.subheadline)
                }

                Spacer()
            }
            // Customer details
            VStack(alignment: .leading, spacing: 8) {
                Text("Billed To:")
                    .font(.headline)

                Text(order.wrappedCustomer.name)
                    .bold()

                let addressComponents = [
                    order.wrappedCustomer.address.line1,
                    order.wrappedCustomer.address.line2,
                    order.wrappedCustomer.address.city,
                    order.wrappedCustomer.address.pincode
                ]
                .filter { !$0.isEmpty }
                .joined(separator: ", ")

                Text(addressComponents)
                    .font(.subheadline)
            }

            Divider()

            // Purchase details
            VStack(alignment: .leading, spacing: 8) {
                HStack {
                    Text("Description")
                        .font(.subheadline)
                        .bold()
                        .frame(maxWidth: .infinity, alignment: .leading)

                    Text("Quantity")
                        .font(.subheadline)
                        .bold()
                        .frame(width: 70, alignment: .trailing)

                    Text("Price")
                        .font(.subheadline)
                        .bold()
                        .frame(width: 90, alignment: .trailing)
                }

                Divider()

                HStack {
                    Text(order.wrappedProduct.name)
                        .frame(maxWidth: .infinity, alignment: .leading)

                    Text("\(order.quantity.formatted()) \
(order.wrappedProduct.measurementUnit.rawValue.capitalized)")
                        .frame(width: 70, alignment: .trailing)

                    Text("\(order.amountPaid, format: .currency(code: "INR"))")
                        .frame(width: 90, alignment: .trailing)
                }

                Divider()
            }
            // Purchase Total
            HStack {
                Spacer()
                VStack(alignment: .trailing, spacing: 4) {
                    Text("Total:")
                        .font(.subheadline)
                        .bold()

                    Text("\(order.amountPaid, format: .currency(code: "INR"))")
                        .font(.body)
                }
            }

        }
        .padding()
    }

    var body: some View {
        NavigationStack {
            bill
                .frame(maxHeight: .infinity, alignment: .center)
                .toolbar {
                    ToolbarItem(placement: .cancellationAction) {
                        Button {
                            dismiss()
                        } label: {
                            Label("Return", systemImage: "chevron.left")
                        }
                    }
                    // Bottom toolbar item to share the bill.
                    ToolbarItemGroup(placement: .bottomBar) {
                        // Renders the Bill SwiftUI View as an image, with a forced
light mode to ensure the invoice is printable.
                        let renderer = ImageRenderer(content: bill
                            .background(.white)
                            .environment(\.colorScheme, .light))
                        if let image = renderer.uiImage {
                            let swiftUIImage = Image(uiImage: image)

                            // ShareLink is used to share the generated invoice via
the system share sheet.
                            ShareLink(item: swiftUIImage, preview:
SharePreview("Bill", image: swiftUIImage))
                        }
                    }
                }
                .navigationTitle("Generated Invoice")
                .navigationBarTitleDisplayMode(.inline)
        }
    }
}

// Source: https://github.com/aheze/VariableBlurView Accessed 10/03/25

import SwiftUI
#if canImport(UIKit)
import UIKit
#endif
```

```
import CoreImage.CIFilterBuiltins
import QuartzCore

public enum VariableBlurDirection {
    case blurredTopClearBottom
    case blurredBottomClearTop
}


public struct VariableBlurView: UIViewRepresentable {

    public var maxBlurRadius: CGFloat = 20

    public var direction: VariableBlurDirection = .blurredTopClearBottom

    /// By default, variable blur starts from 0 blur radius and linearly increases to `maxBlurRadius`. Setting 
`startOffset` to a small negative coefficient (e.g. -0.1) will start blur from larger radius value which might look 
better in some cases.
    public var startOffset: CGFloat = 0

    public func makeUIView(context: Context) -> VariableBlurUIView {
        VariableBlurUIView(maxBlurRadius: maxBlurRadius, direction: direction, 
startOffset: startOffset)
    }

    public func updateUIView(_ uiView: VariableBlurUIView, context: Context) {
    }
}


/// credit https://github.com/jtrivedi/VariableBlurView
open class VariableBlurUIView: UIVisualEffectView {

    public init(maxBlurRadius: CGFloat = 20, direction: VariableBlurDirection 
= .blurredTopClearBottom, startOffset: CGFloat = 0) {
        super.init(effect: UIBlurEffect(style: .regular))

        // `CAFilter` is a private QuartzCore class that dynamically create using 
Objective-C runtime.
        guard let CAFilter = NSClassFromString("CAFilter")! as? NSObject.Type else {
            print("[VariableBlur] Error: Can't find CAFilter class")
            return
        }
        guard let variableBlur = 
CAFilter.self.perform(NSSelectorFromString("filterWithType:"), with: 
"variableBlur").takeUnretainedValue() as? NSObject else {
            print("[VariableBlur] Error: CAFilter can't create filterWithType: 
variableBlur")
            return
        }

        // The blur radius at each pixel depends on the alpha value of the 
corresponding pixel in the gradient mask.
        // An alpha of 1 results in the max blur radius, while an alpha of 0 is 
completely unblurred.
        let gradientImage = makeGradientImage(startOffset: startOffset, direction: 
direction)

        variableBlur.setValue(maxBlurRadius, forKey: "inputRadius")
        variableBlur.setValue(gradientImage, forKey: "inputMaskImage")
        variableBlur.setValue(true, forKey: "inputNormalizeEdges")

        // We use a `UIVisualEffectView` here purely to get access to its 
`CABackdropLayer`,
        // which is able to apply various, real-time CAFilters onto the views 
underneath.
        let backdropLayer = subviews.first?.layer

        // Replace the standard filters (i.e. `gaussianBlur`, `colorSaturate`, etc.) 
with only the variableBlur.
        backdropLayer?.filters = [variableBlur]

        // Get rid of the visual effect view's dimming/tint view, so we don't see a 
hard line.
        for subview in subviews.dropFirst() {
            subview.alpha = 0
        }
    }

    required public init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    open override func didMoveToWindow() {
        // fixes visible pixelization at unblurred edge (https://github.com/nikstar/
VariableBlur/issues/1)
        guard let window, let backdropLayer = subviews.first?.layer else { return }
        backdropLayer.setValue(window.screen.scale, forKey: "scale")
    }

    open override func traitCollectionDidChange(_ previousTraitCollection: 
UITraitCollection?) {
        // `super.traitCollectionDidChange(previousTraitCollection)` crashes the app
    }

    private func makeGradientImage(width: CGFloat = 100, height: CGFloat = 100, 
startOffset: CGFloat, direction: VariableBlurDirection) -> CGImage { // much lower 
resolution might be acceptable
      //  let ciGradientFilter =  CIFilter.linearGradient()
        let ciGradientFilter =  CIFilter.smoothLinearGradient()
        ciGradientFilter.color0 = CIColor.black
        ciGradientFilter.color1 = CIColor.clear
        ciGradientFilter.point0 = CGPoint(x: 0, y: height)
        ciGradientFilter.point1 = CGPoint(x: 0, y: startOffset * height) // small 
negative value looks better with vertical lines
        if case .blurredBottomClearTop = direction {
            ciGradientFilter.point0.y = 0 - 10
            ciGradientFilter.point1.y = height - ciGradientFilter.point1.y + 10
        }
        return CIContext().createCGImage(ciGradientFilter.outputImage!, from: 
CGRect(x: 0, y: 0, width: width, height: height))!
    }
}
```