**Введение в разработку мобильных приложений и обзор TCA**

Разработка мобильных приложений для iOS — это захватывающая область, которая позволяет создавать функциональные и удобные приложения для миллионов пользователей по всему миру. В этой лекции мы познакомимся с архитектурой **TCA (The Composable Architecture)**, которая помогает разрабатывать современные приложения с акцентом на управление состоянием и модульность.

**Что такое TCA и зачем она нужна?**

The Composable Architecture (TCA) — это библиотека и архитектурный подход для построения приложений с использованием одностороннего потока данных.

**Почему стоит использовать TCA?**

• Централизованное управление состоянием упрощает отладку и тестирование.

• Приложения становятся модульными и легко расширяемыми.

• TCA отлично интегрируется с декларативным подходом SwiftUI.

**Основные компоненты TCA:**

1. **State:** отвечает за текущее состояние приложения.

2. **Action:** представляет пользовательские или системные действия.

3. **Reducer:** функция, которая изменяет состояние на основе действия.

4. **Store:** хранит состояние и связывает его с интерфейсом.

5. **Environment:** управляет внешними зависимостями, такими как API или базы данных.

**Основные архитектуры в iOS**

В iOS-разработке существует несколько популярных архитектур. Каждая из них имеет свои преимущества и недостатки:

1. **MVC (Model-View-Controller):**

• Простая реализация.

• Код сложно поддерживать в крупных проектах из-за тесной связи компонентов.

2. **MVVM (Model-View-ViewModel):**

• Улучшает разделение логики и интерфейса.

• Подходит для использования с SwiftUI благодаря привязке данных.

3. **VIPER:**

• Разделяет приложение на слои.

• Подходит для сложных приложений, но добавляет избыточность.

4. **TCA:**

• Модульная, тестируемая архитектура.

• Идеальна для SwiftUI и работы с состоянием.

**Пример структуры приложения на TCA**

**1. State**

struct CounterState: Equatable {

var count: Int = 0

}

CounterState определяет, что приложение хранит только одно значение — текущий счётчик.

**2. Action**

enum CounterAction: Equatable {

    case increment

    case decrement

}

Действия описывают возможные изменения состояния: увеличение или уменьшение счётчика.

**3. Reducer**

let counterReducer = Reducer<CounterState, CounterAction, Void> { state, action, \_ in

    switch action {

    case .increment:

        state.count += 1

        return .none

    case .decrement:

        state.count -= 1

        return .none

    }

}

Reducer принимает текущее состояние и действие, а затем возвращает новое состояние.

**4. Store**

let store = Store(

    initialState: CounterState(),

    reducer: counterReducer,

    environment: ()

)

Store объединяет состояние, редюсер и зависимости, которые используются в приложении.