Лекция 8 - Управление состоянием приложения, работа с сетью

Сегодняшняя наша тема - управление состоянием приложения и работа с сетью, включая использование REST API и библиотек для сетевого взаимодействия. Эти аспекты являются ключевыми для создания современных интерактивных приложений, будь то веб-сайты или мобильные приложения.

Управление состоянием приложения

Управление состоянием - это процесс отслеживания изменений, которые происходят с данными в вашем приложении. Это может включать пользовательский ввод, данные, полученные от сервера, пользовательские предпочтения и так далее.

Зачем нужно управлять состоянием?

- Повышение производительности: Корректное управление избегает ненужных обновлений и перерисовок.
- Легкость отладки: Состояние приложения можно инспектировать и изменять на лету.
- Предсказуемость поведения: Зная текущее состояние, можно точно предсказать реакцию приложения на действия пользователя.
- Масштабируемость: Правильное управление состоянием упрощает добавление новых функций и возможностей.

Паттерны управления состоянием:

- Global State Management (Redux, Vuex, NgRx)
- Local State Management (useState в React, data в Vue)
- Context API / Provide & inject механизмы для избегания "prop drilling".
- State Management Libraries (MobX, Recoil)

Работа с сетью

REST API

REST (Representational State Transfer) - это архитектурный стиль взаимодействия компонентов распределенного приложения в сети. RESTful API - это API, который следует принципам REST.

В мобильной разработке важная часть работы приложения часто связана с сетевым взаимодействием. Приложения могут взаимодействовать с сервером для получения данных, отправки данных пользователя или выполнения различных операций на сервере. RESTful API является одним из самых популярных способов для такого взаимодействия.

Компоненты REST:

- Ресурсы: Основная абстракция информации в REST, обычно представляется в виде URI.
- HTTP методы: GET, POST, PUT, DELETE и др. определяют действия над ресурсами.
- Статусы ответов HTTP: Коды (200, 404, 500 и т.д.), указывающие на результат операции.

Особенности работы с REST API в мобильной разработке:

1. Асинхронность:

В мобильных приложениях сетевые запросы должны выполняться асинхронно, чтобы избежать блокировки главного потока (UI thread), который отвечает за взаимодействие с пользователем.

2. Обработка ответов:

Каждый сетевой запрос возвращает ответ, который может быть успешным или содержать ошибку. Разработчики должны правильно обрабатывать эти ответы и предоставлять соответствующий UI для каждого случая.

3. Поддержка оффлайн-режима:

Хорошее мобильное приложение способно корректно функционировать при ограниченном или отсутствующем интернет-соединении, что подразумевает кэширование данных и их синхронизацию.

4. Безопасность:

Обеспечение безопасной передачи данных между клиентом и сервером, используя HTTPS и другие механизмы безопасности, такие как аутентификация и авторизация с помощью токенов.

Как работать с REST API в Flutter:

Flutter предоставляет несколько способов для выполнения сетевых запросов. Вот пример использования пакета http для выполнения HTTP запросов:

```
import 'package:http/http.dart' as http;
import 'dart:convert';

Future<void> fetchPosts() async {
  final response = await
http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));

  if (response.statusCode == 200) {
    List<dynamic> posts = jsonDecode(response.body);
    // Обработка полученных данных
  } else {
    // Ошибка сервера
    throw Exception('Failed to load posts');
  }
}
```

Обработка различных HTTP методов:

GET для получения данных.
POST для создания новых записей.
PUT для обновления существующих записей.
DELETE для удаления записей.

Каждый из этих методов соответствует стандартному CRUD (Create, Read, Update, Delete) интерфейсу.

Обработка статусов ответов HTTP:

200 OK - успешный запрос.
201 Created - успешно создана запись.
204 No Content - успешно обработан запрос, но нет контента для ответа.

400 Bad Request - ошибка в запросе от клиента.
401 Unauthorized - ошибка аутентификации.
403 Forbidden - доступ запрещён.
404 Not Found - ресурс не найден.
500 Internal Server Error - ошибка на стороне сервера.
Разработчики должны предусмотреть обработку всех возможных ответов сервера для корректного функционирования приложения.

Библиотеки для работы с REST API в Flutter:
http:
Простая и удобная библиотека для отправки HTTP запросов.
Dio:
Мощная библиотека для работы с HTTP запросами, которая предоставляет расширенные возможности по сравнению с http.
Retrofit:
Популярная библиотека в мире Android, портированная для Flutter, которая позволяет работать с REST API, используя аннотации.
Chopper:
Библиотека, вдохновленная Retrofit, которая генерирует код для работы с HTTP запросами.
Пример использования Dio для выполнения GET запроса:

```
import 'package:dio/dio.dart';

Future<void> fetchUserPosts() async {
  var dio = Dio();
  final response = await dio.get('https://jsonplaceholder.typicode.com/posts');

  if (response.statusCode == 200) {
    List<dynamic> posts = response.data;
    // Обработка данных
  } else {
    // Ошибка сервера
    throw Exception('Failed to load posts');
  }
}
```

Ключевым аспектом работы с сетью является умение правильно интегрировать сетевые запросы в жизненный цикл приложения, обеспечивать обработку исключений и ошибок, а также предоставлять пользователю понятный интерфейс в случае непредвиденных ситуаций, таких как отсутствие сети или ошибки сервера.

Управление состоянием в мобильных приложениях

Управление состоянием в мобильной разработке имеет свои особенности, связанные с ограниченными ресурсами устройства и необходимостью сохранять отзывчивость интерфейса.
Примеры подходов к управлению состоянием:
SharedPreferences/NSUserDefaults — для хранения небольших данных, таких как настройки пользователя.

SQLite/CoreData/Room — для организации сложного хранения данных в виде баз данных.

In-memory caches — кэши в памяти, чтобы быстро доступаться к часто используемой информации.

State management libraries:
Flutter/Dart: Provider, Bloc, Redux, Riverpod.
React Native/JavaScript: Redux, MobX, Context API, Recoil.
Android/Kotlin: LiveData, ViewModel, StateFlow, Redux.
iOS/Swift: Redux, Combine, RxSwift.

Пример управления состоянием с ViewModel и LiveData в Android:

Kotlin:

```
// ViewModel
class UserViewModel : ViewModel() {
    private val _userData = MutableLiveData<User>()
    val userData: LiveData<User> = _userData

    fun loadUser(userId: String) {
        // Загрузка данных пользователя, например, из сети или базы данных
        _userData.value = repository.getUser(userId)
    }
}

// Activity
class UserActivity : AppCompatActivity() {

    private lateinit var viewModel: UserViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel = ViewModelProvider(this).get(UserViewModel::class.java)

        viewModel.userData.observe(this, Observer { user ->
            // Обновляем UI с данными пользователя
        })

        // Загружаем данные пользователя
        viewModel.loadUser("userId")
    }
}
```

Работа с сетью в МП

Мобильные приложения часто взаимодействуют с сервером для получения и отправки данных. Для этих целей используется REST API, который позволяет выполнить CRUD операции над ресурсами сервера.

HTTP клиенты для мобильной разработки:

- **Android:** Retrofit, OkHttp, Volley.
- **iOS:** Alamofire, URLSession, AFNetworking.
- **Cross-platform:** HttpClient в Flutter, Axios в React Native.

Пример сетевого запроса с использованием Retrofit в Android:

```
interface ApiService {
    @GET("users/{user}")
    suspend fun getUser(@Path("user") userId: String): Response<User>
}

// Retrofit Builder
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

// Использование в коде
val apiService = retrofit.create(ApiService::class.java)
val response = apiService.getUser("userId")

if (response.isSuccessful) {
    // Делаем что-то с данными пользователя
}
```

Пример сетевого запроса с использованием URLSession в iOS:

```
let url = URL(string: "https://api.example.com/users/userId")!
let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
    if let error = error {
        // Обработка ошибки
        return
    }
    guard let httpResponse = response as? HTTPURLResponse,
          (200...299).contains(httpResponse.statusCode) else {
        // Обработка ответа сервера как ошибки
        return
    }
    if let data = data,
       let user = try? JSONDecoder().decode(User.self, from: data) {
        // Использование данных пользователя
    }
}
task.resume()
```

В мобильной разработке управление состоянием и работа с сетью имеют ряд специфик, вызванных ограниченными ресурсами и необходимостью обеспечить отзывчивый интерфейс. Понимание этих аспектов и умение использовать

соответствующие инструменты и библиотеки — ключ к созданию качественных мобильных приложений.

Flutter

Состояние приложения в Flutter можно управлять различными способами, и здесь мы рассмотрим несколько популярных библиотек и паттернов управления состоянием: Provider, Bloc, Redux и Riverpod.

Provider — это библиотека рекомендованная Flutter командой для упрощения управления состоянием. Она работает на основе прослушивания изменений в объектах и перестроения интерфейса при их изменении.

Пример использования Provider:

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => Counter(),
      child: MaterialApp(
        home: HomePage(),
      ),
    );
  }
}

class Counter with ChangeNotifier {
  int _count = 0;

  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counter = Provider.of<Counter>(context);

    return Scaffold(
      appBar: AppBar(title: Text('Example Provider')),
      body: Center(
        child: Text('Value: ${counter.count}'),
      ),
```

```
      floatingActionButton: FloatingActionButton(
        onPressed: counter.increment,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Bloc

Bloc — это паттерн, который помогает разделить бизнес-логику от представления с помощью потоков (Streams) и Синков (Sinks). Он предлагает более предсказуемый способ управления состоянием через события и состояния.

Пример использования Bloc:

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:counter_bloc/counter_bloc.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) => CounterBloc(),
      child: MaterialApp(
        home: CounterPage(),
      ),
    );
  }
}

class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Получаем Bloc из контекста
    final CounterBloc counterBloc = BlocProvider.of<CounterBloc>(context);

    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: BlocBuilder<CounterBloc, int>(
        builder: (context, count) {
          return Center(child: Text('$count'));
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          // Добавляем событие
          counterBloc.add(CounterEvent.increment);
        },
```

```
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Redux

Redux — это библиотека, которая использует концепцию глобального состояния приложения, которое управляется с помощью действий и редюсеров.

Пример использования Redux в Flutter:

```
import 'package:flutter/material.dart';
import 'package:flutter_redux/flutter_redux.dart';
import 'package:redux/redux.dart';

void main() {
  final store = Store<int>(counterReducer, initialState: 0);
  runApp(MyApp(store: store));
}

class MyApp extends StatelessWidget {
  final Store<int> store;

  MyApp({required this.store});

  @override
  Widget build(BuildContext context) {
    return StoreProvider<int>(
      store: store,
      child: MaterialApp(
        home: CounterPage(),
      ),
    );
  }
}

class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: StoreConnector<int, String>(
        converter: (store) => store.state.toString(),
        builder: (context, count) {
          return Center(child: Text(count));
        },
      ),
      floatingActionButton: StoreConnector<int, VoidCallback>(
        converter: (store) {
          return () => store.dispatch(IncrementAction());
        },
```

```
        builder: (context, callback) {
          return FloatingActionButton(
            onPressed: callback,
            child: Icon(Icons.add),
          );
        },
      ),
    );
  }
}

// Action
class IncrementAction {}

// Reducer
int counterReducer(int state, dynamic action) {
  if (action is IncrementAction) {
    return state + 1;
  }
  return state;
}
```

Riverpod

Riverpod — это более новая и гибкая замена для Provider, которая позволяет более удобно управлять состоянием, делая его более тестируемым и масштабируемым.

Пример использования Riverpod:

```
import 'package:flutter/material.dart';
import 'package:flutter_riverpod/flutter_riverpod.dart';

final counterProvider = StateProvider((ref) => 0);

void main() => runApp(
      ProviderScope(child: MyApp()),
    );

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Riverpod Counter Example')),
        body: Consumer(
          builder: (context, watch, child) {
            final count = watch(counterProvider).state;
            return Center(child: Text(count.toString()));
          },
        ),
        floatingActionButton: Consumer(
          builder: (context, watch, child) {
```

```
            return FloatingActionButton(
              onPressed: () => context.read(counterProvider).state++,
              child: Icon(Icons.add),
            );
          },
        ),
      ),
    );
  }
}
```

Каждый из этих методов управления состоянием имеет свои преимущества и может использоваться в различных сценариях. Важно выбрать подход, который лучше всего подходит для размера и сложности вашего приложения, а также для команды, которая над ним работает.

JSON

JSON (JavaScript Object Notation) - это текстовый формат обмена данными, который легко читается как человеками, так и машинами.

Библиотеки для сетевого взаимодействия:

- Fetch API: Встроенный в браузеры инструмент для работы с HTTP-запросами.
- Axios: Популярная библиотека JavaScript, которая упрощает выполнение HTTP-запросов.
- Retrofit для Android: Типичный выбор для взаимодействия с REST API на Android.
- Alamofire для iOS: Мощная библиотека для работы с HTTP-запросами на Swift.

Примеры кода

Давайте рассмотрим простой пример с использованием Axios для отправки GET-запроса:

Javascript:

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Ошибка при выполнении запроса:', error);
  });
```

А теперь пример с использованием Fetch API:

```
fetch('https://api.example.com/data')
  .then(response => {
```

```
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
})
.then(data => {
  console.log(data);
})
.catch(error => {
  console.error('Ошибка при выполнении запроса:', error);
});
```

Пример работы с REST API с использованием async/await в JavaScript

```
async function getUserData(userId) {
  try {
    const response = await fetch(`https://api.example.com/users/${userId}`);
    if (!response.ok) {
      throw new Error(`Ошибка: ${response.status}`);
    }
    const data = await response.json();
    console.log('Полученные данные пользователя:', data);
  } catch (error) {
    console.error('Произошла ошибка при получении данных пользователя:', error);
  }
}

getUserData(1);
```

Пример POST-запроса с Axios

```
import axios from 'axios';

async function createUser(userData) {
  try {
    const response = await axios.post('https://api.example.com/users', userData);
    console.log('Пользователь создан:', response.data);
  } catch (error) {
    console.error('Ошибка при создании пользователя:', error);
  }
}

const newUser = {
  name: 'John Doe',
  email: 'john.doe@example.com',
};

createUser(newUser);
```

Эти примеры демонстрируют основные концепции управления состоянием и взаимодействия с сетью в современных JavaScript-приложениях. Примеры кода представлены в сокращенной форме и предполагают базовое понимание соответствующих фреймворков и библиотек.

Управление состоянием и работа с сетью - это фундаментальные аспекты современной разработки приложений. Использование REST API и библиотек для сетевого взаимодействия позволяет приложениям быть интерактивными и динамичными, обмениваясь данными с сервером и предоставляя пользователям актуальную информацию. Важно понимать принципы работы с состоянием и сетевыми запросами для создания эффективных и масштабируемых приложений. Спасибо за внимание, теперь перейдем к вопросам и практическим заданиям.