Лекция 5 - Управление состоянием приложения и работа с сетью

Тлеубаева А.О.

Введение

Разработка мобильных приложений — это сложный и многоаспектный процесс, требующий глубокого понимания различных технических аспектов, среди которых управление состоянием и работа с сетью занимают ключевые позиции. Эти компоненты критически важны для создания надежных, эффективных и удобных в использовании мобильных приложений.

Значение управления состоянием

Состояние приложения включает в себя все данные, которые приложение может хранить и изменять во время своей работы: от данных пользователя до информации о текущем интерфейсе и настройках. Управление состоянием — это процесс контроля за этими данными: их хранением, обновлением и передачей между компонентами приложения.

Эффективное управление состоянием позволяет приложению работать плавно и интуитивно понятно для пользователя. Оно обеспечивает сохранность данных при смене экранов, обновлении контента, восстановлении после ошибок и при перезапусках приложения.

Роль работы с сетью

Практически любое современное мобильное приложение в той или иной степени взаимодействует с сетью: загружает данные, отправляет информацию на сервер, синхронизирует настройки и так далее. Работа с сетью обеспечивает динамичное обновление контента, возможность работы с пользовательскими данными в реальном времени и поддержку сетевых сервисов и API.

Однако сетевое взаимодействие вносит дополнительные сложности в разработку и тестирование приложений, такие как управление состоянием подключения, обработка ошибок сети, оптимизация загрузки данных для сохранения трафика и батареи устройства. Поэтому разработчикам необходимо тщательно продумывать архитектуру сетевых запросов и стратегии кэширования данных.

Взаимосвязь управления состоянием и работы с сетью

Управление состоянием и работа с сетью тесно связаны. Состояние приложения может зависеть от данных, полученных из сети, а изменения состояния могут требовать отправки данных на сервер. Например, приложение для социальной сети может отображать список сообщений (состояние), который регулярно обновляется из сети. При этом отправка нового сообщения изменяет как локальное состояние приложения, так и данные на сервере.

Эффективная разработка мобильных приложений требует от разработчиков глубокого понимания этих аспектов и умения грамотно их применять для создания качественных продуктов, способных удовлетворить запросы и ожидания пользователей в условиях постоянно меняющегося цифрового мира.

Управление состоянием приложения

- Управление состоянием приложения играет критическую роль в разработке мобильных приложений, обеспечивая гладкую и интуитивно понятную работу пользовательского интерфейса, а также надёжное сохранение и обработку данных пользователя. Понимание различных видов состояний и методов их управления позволяет создавать эффективные и удобные приложения.

Определение и виды состояний приложения

- **Состояние приложения** — это любая информация, которую приложение сохраняет в памяти или на диске для последующего использования. Это может включать:
- **Локальное состояние**: Состояние, специфичное для отдельного компонента или экрана (например, текст, введённый в текстовое поле).
- **Глобальное состояние**: Состояние, доступное для всего приложения (например, профиль пользователя, настройки).
- **Сетевое состояние**: Информация, связанная с сетевыми запросами и ответами (например, данные, загруженные с сервера).
- **Сессионное состояние**: Данные, связанные с текущей сессией пользователя (например, идентификатор сессии, токены аутентификации).

Методы и инструменты управления состоянием

Разработчики могут выбирать из множества подходов и инструментов для управления состоянием, в зависимости от требований и сложности приложения:

- **SharedPreferences**: Легковесное хранилище ключ-значение для сохранения простых данных, таких как настройки пользователя.
- **Базы данных**: SQLite, Room (для Android) предоставляют мощные средства для хранения структурированных данных, таких как пользовательские записи, списки и т.д.
- **Файловое хранилище**: Для сохранения файлов или сериализованных объектов напрямую в файловую систему устройства.
- **State Management Libraries**: Библиотеки, такие как Redux, MobX, Provider (для Flutter), Vuex (для Vue Native), позволяют централизованно управлять состоянием в больших и сложных приложениях.

Примеры использования локального хранения данных

1. **Сохранение настроек пользователя**:
    1. Использование SharedPreferences для сохранения простых настроек, таких как предпочтения в уведомлениях или темной теме интерфейса.
2. **Хранение сложных данных**:
    1. Применение SQLite или Room для хранения и управления сложными пользовательскими данными, например, историей заказов в приложении для электронной коммерции.
3. **Кэширование сетевых запросов**:
    1. Использование файлового хранилища или баз данных для кэширования данных, полученных из сетевых запросов, что позволяет приложению функционировать в офлайн-режиме.
4. **Управление состоянием формы**:
    1. Применение библиотек управления состоянием для сложных форм, позволяющих сохранять введённые данные, управлять валидацией и состоянием отправки формы.

- Каждый из этих методов и инструментов имеет свои преимущества и недостатки, и выбор зависит от специфики задачи, требований к производительности и предпочтений разработчика. Грамотное управление состоянием приложения позволяет создать основу для разработки надёжных и удобных мобильных приложений.

Работа с сетью в разработке мобильных приложений

- Взаимодействие с сетью является фундаментальной частью большинства мобильных приложений, позволяя им обмениваться данными с сервером, получать актуальную информацию и предоставлять пользователю динамический контент. Одним из наиболее распространенных подходов к организации сетевого обмена является использование REST API.

Знакомство с REST API

- **REST (Representational State Transfer)** — это архитектурный стиль взаимодействия компонентов распределенного приложения в сети. REST предполагает обращение к ресурсам (данным) через простые HTTP-запросы, используя стандартные методы: GET для получения данных, POST для создания, PUT для обновления и DELETE для удаления данных.
- REST API использует URL для идентификации ресурсов и HTTP-статусы для индикации результатов операций, что делает его легко используемым и интегрируемым с любым типом клиентских приложений, включая мобильные.

Популярные библиотеки для сетевого взаимодействия

1.Retrofit:

1. Одна из наиболее популярных библиотек для Android, разработанная Square.
2. Позволяет легко интегрировать REST API, автоматически сериализуя данные в JSON или XML и обратно.
3. Поддерживает синхронные и асинхронные запросы, а также мощную систему обработки ошибок.

2.OkHttp:

1. Также разработана Square, часто используется в связке с Retrofit как HTTP-клиент.
2. Отличается высокой производительностью и эффективностью благодаря встроенным механизмам кэширования и повтора запросов.

3.Volley:

1. Библиотека от Google, оптимизированная для выполнения сетевых запросов и обработки изображений.
2. Хорошо подходит для отправки данных в формате JSON, обработки большого количества маленьких запросов.

Примеры создания сетевых запросов и обработки ответов

- **Пример с Retrofit**:

```java
// Определение интерфейса для сервиса API. Этот интерфейс описывает конечные точки API,
// которые вы хотите использовать в вашем приложении.
public interface MyApiService {
    // Аннотация @GET указывает на тип HTTP-запроса (GET) и путь к ресурсу.
    // В данном случае "users/{user}/repos" будет динамически заменен на имя пользователя
    // для получения списка его репозиториев.
    @GET("users/{user}/repos")
    Call<List<Repo>> listRepos(@Path("user") String user);
}

// Создание экземпляра Retrofit, который будет использоваться для выполнения сетевых запросов.
Retrofit retrofit = new Retrofit.Builder()
    // Установка базового URL для всех запросов. Все относительные URL-адреса будут добавляться к этому базовому адресу.
    .baseUrl("https://api.github.com/")
    // Добавление конвертера, который будет использоваться для сериализации данных.
    // GsonConverterFactory автоматически конвертирует JSON-ответы от сервера в Java-объекты.
    .addConverterFactory(GsonConverterFactory.create())
    // Построение объекта Retrofit.
    .build();

// Создание реализации интерфейса API, определенного выше, с использованием Retrofit.
// Этот объект будет использоваться для отправки запросов к API.
MyApiService service = retrofit.create(MyApiService.class);

// Вызов метода интерфейса API для получения списка репозиториев пользователя "octocat".
// Возвращается объект Call, который можно использовать для асинхронного или синхронного выполнения запроса.
Call<List<Repo>> repos = service.listRepos("octocat");

```

- **MyApiService:** Это интерфейс, в котором используются аннотации Retrofit для определения HTTP-запросов. В данном случае определен один метод listRepos, который выполняет GET-запрос к GitHub API для получения списка репозиториев указанного пользователя.
- **Аннотация @GET:** Указывает на тип HTTP-запроса (GET) и путь к ресурсу. Путь содержит переменную {user}, значение которой будет подставлено во время выполнения запроса.
- **Аннотация @Path:** Используется для указания того, что параметр метода String user должен быть использован для замены соответствующей переменной в URL-адресе запроса.
- **Retrofit.Builder:** Используется для настройки и создания экземпляра Retrofit. Здесь задается базовый URL для всех запросов, а также добавляется фабрика конвертеров GsonConverterFactory для автоматической обработки данных JSON.
- **retrofit.create(MyApiService.class):** Создает реализацию интерфейса MyApiService, которую можно использовать для отправки сетевых запросов.
- **service.listRepos("octocat"):** Вызывает метод listRepos с аргументом "octocat", что инициирует создание и отправку HTTP-запроса к API GitHub для получения списка репозиториев пользователя "octocat".

Пример с OkHttp:

```java
// Создание экземпляра OkHttpClient, который будет использоваться для отправки запросов.
OkHttpClient client = new OkHttpClient();

// Построение объекта запроса, включая URL-адрес, по которому будет выполнен запрос.
Request request = new Request.Builder()
    .url("https://api.github.com/users/octocat/repos")
    .build();

// Отправка асинхронного запроса с использованием созданного клиента и запроса.
client.newCall(request).enqueue(new Callback() {
    // Метод onResponse вызывается, когда получен ответ от сервера.
    @Override
    public void onResponse(Call call, Response response) throws IOException {
        // Проверка, является ли ответ успешным (код состояния 200-299).
        if (response.isSuccessful()) {
            // Здесь можно обработать ответ сервера, например, прочитать тело ответа.
            // Пример: String responseData = response.body().string();
        }
    }

    // Метод onFailure вызывается, когда запрос не удалось выполнить из-за отмены,
    // сетевой проблемы или истекшего времени ожидания.
    @Override
    public void onFailure(Call call, IOException e) {
        // Обработка ситуации с ошибкой, например, можно вывести сообщение об ошибке.
    }
});

```

- **OkHttpClient:** Это клиент для отправки и получения HTTP-запросов и ответов. Один экземпляр OkHttpClient является полностью функциональным и может быть использован для отправки множества запросов.
- **Request.Builder:** Используется для создания объекта Request, который инкапсулирует все необходимые данные о запросе, включая URL.
- **enqueue(Callback callback):** Этот метод асинхронно отправляет запрос и уведомляет ваш Callback об ответе или если возникла ошибка. Это позволяет выполнять сетевые запросы без блокирования основного потока приложения.
- **onResponse и onFailure:** Это переопределенные методы интерфейса Callback, предоставляемые OkHttp для обработки ответов на запросы. onResponse вызывается, если запрос был успешно выполнен и получен ответ от сервера, в то время как onFailure вызывается в случае ошибки при выполнении запроса.

Этот код является базовым примером использования OkHttp для асинхронных HTTP-запросов, демонстрируя его простоту и мощь в обработке сетевых операций.

Пример с Volley:

```java
// Инициализация очереди запросов Volley. Context 'this' указывает на текущий контекст,
// например, Activity или Application, в зависимости от того, где этот код выполняется.
RequestQueue queue = Volley.newRequestQueue(this);

// URL-адрес, по которому будет отправлен запрос.
String url ="https://api.github.com/users/octocat/repos";

// Создание StringRequest для отправки GET-запроса. StringRequest ожидает в ответе строку.
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
        new Response.Listener<String>() {
            // Метод, вызываемый при успешном получении ответа от сервера.
            @Override
            public void onResponse(String response) {
                // Здесь можно обработать ответ сервера, преобразованный в строку.
                // Например, можно парсить JSON-ответ и использовать полученные данные.
            }
        }, new Response.ErrorListener() {
            // Метод, вызываемый при возникновении ошибки при выполнении запроса.
            @Override
            public void onErrorResponse(VolleyError error) {
                // Здесь можно обработать ошибку, например, вывести сообщение об ошибке.
            }
        });

// Добавление созданного запроса в очередь запросов для его выполнения.
queue.add(stringRequest);
```

- **Ключевые моменты:**
- **RequestQueue**: Компонент Volley, управляющий потоком сетевых запросов. Он отвечает за планирование выполнения запросов, их отправку и обработку ответов.
- **StringRequest**: Один из типов запросов, предоставляемых Volley, предназначенный для получения ответов в виде строки. В конструкторе StringRequest указывается метод запроса (GET), URL-адрес, слушатель для обработки ответа и слушатель для обработки ошибок.
- **Response.Listener и Response.ErrorListener**: Интерфейсы слушателей, предоставляемые Volley для обработки успешных ответов и ошибок соответственно. onResponse вызывается, когда сервер успешно возвращает ответ, а onErrorResponse - при возникновении ошибки.
- **queue.add(stringRequest)**: Добавление запроса в RequestQueue инициирует его выполнение. Volley автоматически управляет выполнением запроса в фоновом режиме, обеспечивая асинхронную обработку без блокировки основного потока приложения.
- Использование Volley упрощает выполнение сетевых запросов и обработку ответов, предоставляя мощный и гибкий инструментарий для работы с сетью в Android-приложениях.

Онлайн и офлайн режимы работы приложения

- Современные мобильные приложения часто требуют поддержки работы как в онлайн, так и в офлайн режимах. Это обеспечивает пользователям возможность доступа к функционалу приложения даже при отсутствии сетевого соединения, повышая удобство использования и надежность приложения.

- Принципы построения приложений с поддержкой обоих режимов

1. Разделение данных на локальные и удаленные:

1. Приложение должно четко разделять данные, которые могут быть доступны локально, и данные, для доступа к которым необходимо сетевое соединение.

2. Локальное хранение данных:

1. Все данные, необходимые для работы приложения в офлайн режиме, должны кэшироваться на устройстве пользователя.

3. Умное кэширование и синхронизация:

1. Приложение должно определять, когда и какие данные нужно обновлять или синхронизировать с сервером, чтобы минимизировать использование сети и обеспечить актуальность данных.

4. Уведомление пользователя о режиме работы:

1. Пользователь должен быть информирован о текущем сетевом состоянии и о том, в каком режиме работает приложение (онлайн или офлайн).

- Механизмы кэширования данных и синхронизации

1.Кэширование данных:

1. Использование баз данных (например, SQLite, Realm) или легковесных хранилищ (SharedPreferences) для сохранения данных на устройстве.
2. Применение стратегий кэширования, таких как "сначала кэш", "только кэш" и "сеть в приоритете", в зависимости от типа данных и требований приложения.

2.Синхронизация данных:

1. Разработка механизмов для автоматической синхронизации локальных данных с сервером при восстановлении сетевого соединения.
2. Использование фоновых сервисов и задач для выполнения синхронизации без вмешательства пользователя.

- Тестирование и отладка

1.Тестирование сценариев работы в офлайн и онлайн:

1. Проведение тестирования поведения приложения при переходе из одного режима в другой, а также при различных условиях сетевого соединения.

2.Отладка процессов кэширования и синхронизации:

1. Использование инструментов отладки и логирования для проверки корректности механизмов кэширования и синхронизации данных.

3.Производительность и оптимизация:

1. Анализ производительности приложения в офлайн режиме, оптимизация времени загрузки и объема используемой памяти.

- Создание приложений, поддерживающих работу в онлайн и офлайн режимах, требует тщательного планирования и реализации. Основное внимание следует уделять механизмам кэширования и синхронизации данных, а также обеспечению бесперебойной работы приложения в различных условиях сетевого соединения.

Практические рекомендации по управлению состоянием и сетевыми запросами

- Управление состоянием и выполнение сетевых запросов — критические аспекты разработки мобильных приложений, требующие внимательного подхода для обеспечения высокой производительности, безопасности и хорошего пользовательского опыта. Вот несколько практических рекомендаций:
- Управление состоянием

1. **Используйте подходящий механизм хранения**: Выбирайте механизмы хранения состояния (например, SharedPreferences, базы данных, файлы настройки) в зависимости от типа и размера данных. Для сложных структур данных предпочтительнее использовать базы данных.
2. **Минимизируйте глобальное состояние**: Старайтесь ограничивать использование глобального состояния, чтобы упростить управление данными и избежать ошибок совместного доступа.
3. **Разделите состояние на модули**: Для сложных приложений используйте модульное разделение состояния, что упрощает его управление и тестирование.

- Сетевые запросы

1. **Эффективное использование кэширования**: Применяйте стратегии кэширования ответов от сервера, чтобы уменьшить количество сетевых запросов и ускорить загрузку данных.
2. **Обработка ошибок и тайм-аутов**: Реализуйте гибкую обработку ошибок и тайм-аутов для сетевых запросов, чтобы обеспечить надежность приложения в условиях нестабильного соединения.
3. **Оптимизация данных**: Используйте сжатие данных и выбирайте оптимальные форматы передачи данных (например, JSON вместо XML), чтобы уменьшить объем передаваемых данных.

- Безопасность данных

1. **Шифрование данных**: Шифруйте чувствительные данные, хранящиеся локально или передаваемые через сеть, используя стандартные алгоритмы шифрования.
2. **Использование HTTPS**: Всегда используйте HTTPS для сетевых запросов, чтобы обеспечить безопасную передачу данных.
3. **Обновление и поддержка**: Регулярно обновляйте библиотеки и фреймворки до последних версий, чтобы использовать актуальные механизмы безопасности.

- Управление производительностью

1. **Асинхронная загрузка данных**: Выполняйте сетевые запросы и обработку данных асинхронно, чтобы избежать блокировки основного потока приложения.
2. **Профилирование и оптимизация**: Используйте инструменты профилирования для идентификации и оптимизации узких мест в производительности приложения.
3. **Ленивая загрузка данных**: Используйте подходы ленивой загрузки данных (lazy loading), чтобы загружать данные по мере необходимости, а не все сразу.

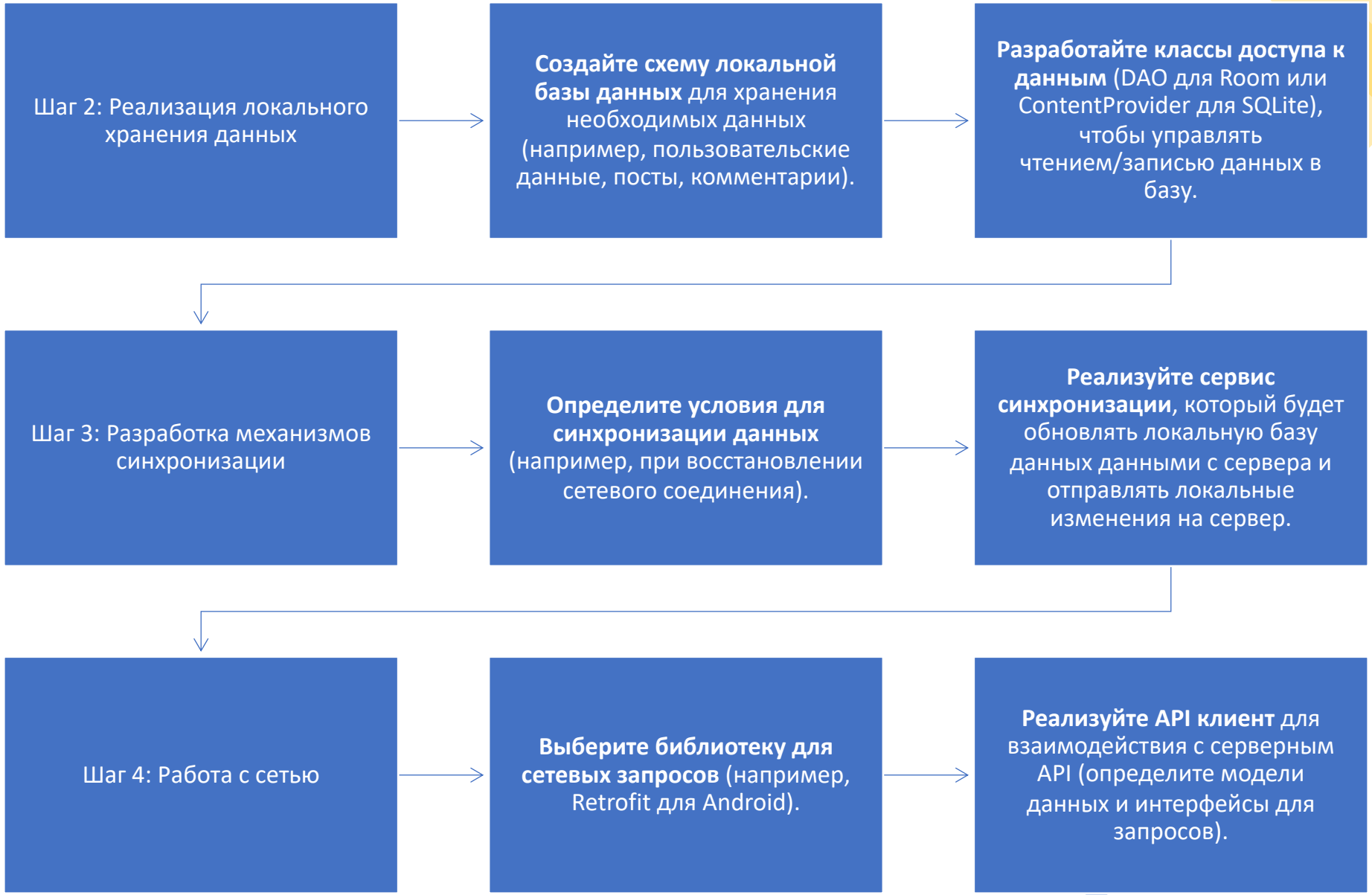
- Инструменты и библиотеки
- **Retrofit, OkHttp, Volley**: Мощные библиотеки для сетевых запросов в Android-приложениях.
- **Room, SQLite**: Библиотеки для работы с локальными базами данных.
- **SharedPreferences**: Для хранения легковесных данных настроек.
- **Jetpack Security Library**: Для шифрования файлов и SharedPreferences.
- **ProGuard/R8**: Инструменты для обфускации кода и оптимизации для улучшения безопасности и производительности.

Соблюдение этих рекомендаций поможет создать более надежные, безопасные и производительные мобильные приложения.

Пошаговое задание для студентов: Создание приложения с функциональностью онлайн и офлайн работы

- **Цель задания**: Разработать мобильное приложение, которое обеспечивает эффективное переключение между онлайн и офлайн режимами и поддерживает синхронизацию данных между локальной базой данных и сервером.

- Шаг 1: Планирование архитектуры приложения
- **Определите ключевые функции приложения**, которые должны быть доступны в офлайн режиме.
- **Выберите подход к архитектуре приложения** (например, MVC, MVP, MVVM), который лучше всего подходит для управления состоянием и сетевыми запросами.
- **Определите механизмы хранения данных** для локальной базы данных (например, SQLite, Room для Android).


Шаг 2: Реализация локального хранения данных
Создайте схему локальной базы данных для хранения необходимых данных (например, пользовательские данные, посты, комментарии).
Разработайте классы доступа к данным (DAO для Room или ContentProvider для SQLite), чтобы управлять чтением/записью данных в базу.
Шаг 3: Разработка механизмов синхронизации
Определите условия для синхронизации данных (например, при восстановлении сетевого соединения).
Реализуйте сервис синхронизации, который будет обновлять локальную базу данных данными с сервера и отправлять локальные изменения на сервер.
Шаг 4: Работа с сетью
Выберите библиотеку для сетевых запросов (например, Retrofit для Android).
Реализуйте API клиент для взаимодействия с серверным API (определите модели данных и интерфейсы для запросов).

- Шаг 5: Обработка режимов работы приложения

1. **Реализуйте механизм определения состояния сети** для переключения между режимами работы.
2. **Добавьте обработку переключения режимов**, обеспечивая корректное отображение данных из локальной базы в офлайн режиме и обновление данных в онлайн режиме.

- Шаг 6: Тестирование приложения

1. **Протестируйте функциональность в офлайн режиме**, убедитесь, что приложение корректно отображает данные из локальной базы.
2. **Проверьте синхронизацию данных** между локальной базой и сервером при восстановлении сетевого соединения.
3. **Проведите тестирование производительности** для оценки скорости работы приложения и времени синхронизации данных.

- Шаг 7: Отладка и оптимизация

1. **Используйте инструменты профилирования** для выявления и устранения узких мест производительности.
2. **Оптимизируйте процессы синхронизации** и хранения данных для улучшения производительности приложения.


Шаг 1: Планирование архитектуры приложения
Определите ключевые функции приложения, которые должны быть доступны в офлайн режиме.
Выберите подход к архитектуре приложения (например, MVC, MVP, MVVM), который лучше всего подходит для управления состоянием и сетевыми запросами.
Определите механизмы хранения данных для локальной базы данных (например, SQLite, Room для Android).
Шаг 2: Реализация локального хранения данных
Создайте схему локальной базы данных для хранения необходимых данных (например, пользовательские данные, посты, комментарии).
Разработайте классы доступа к данным (DAO для Room или ContentProvider для SQLite), чтобы управлять чтением/записью данных в базу.
Шаг 3: Разработка механизмов синхронизации
Определите условия для синхронизации данных (например, при восстановлении сетевого соединения).
Реализуйте сервис синхронизации, который будет обновлять локальную базу данных данными с сервера и отправлять локальные изменения на сервер.
Шаг 4: Работа с сетью
Выберите библиотеку для сетевых запросов (например, Retrofit для Android).
Реализуйте API клиент для взаимодействия с серверным API (определите модели данных и интерфейсы для запросов).
Шаг 5: Обработка режимов работы приложения
Реализуйте механизм определения состояния сети для переключения между режимами работы.
Добавьте обработку переключения режимов, обеспечивая корректное отображение данных из локальной базы в офлайн режиме и обновление данных в онлайн режиме.
Шаг 6: Тестирование приложения
Протестируйте функциональность в офлайн режиме, убедитесь, что приложение корректно отображает данные из локальной базы.
Проверьте синхронизацию данных между локальной базой и сервером при восстановлении сетевого соединения.
Проведите тестирование производительности для оценки скорости работы приложения и времени синхронизации данных.
Шаг 7: Отладка и оптимизация
Используйте инструменты профилирования для выявления и устранения узких мест производительности.
Оптимизируйте процессы синхронизации и хранения данных для улучшения производительности приложения.

- Давайте разработаем концепцию простого мобильного приложения "Заметки", которое позволяет пользователям создавать, просматривать и редактировать заметки как в онлайн, так и в офлайн режимах, а также синхронизировать изменения с сервером, когда соединение становится доступным.
- **Шаг 1: Планирование архитектуры приложения**
- **Ключевые функции**: Создание, просмотр и редактирование заметок.
- **Архитектура приложения**: Используем MVVM для разделения логики UI и бизнес-логики.
- **Механизмы хранения данных**: Room Database для локального хранения заметок.
- **Шаг 2: Реализация локального хранения данных**
- **Схема базы данных**:
  - Таблица Notes с полями id, title, content, lastModified.
- **Классы доступа к данным (DAO)**:
- Kotlin
- @Dao interface NoteDao { @Query("SELECT * FROM Notes") fun getAllNotes(): LiveData<List<Note>> @Insert(onConflict = OnConflictStrategy.REPLACE) fun insertNote(note: Note) @Delete fun deleteNote(note: Note) }
- **Шаг 3: Разработка механизмов синхронизации**
- **Сервис синхронизации**:
  - Создаем SyncService, который проверяет наличие интернет-соединения и синхронизирует локальные заметки с сервером.

- **Шаг 4: Работа с сетью**
- **API клиент (Retrofit)**:kotlin
- interface NotesApi { @GET("notes") fun getAllNotes(): Call<List<Note>> @POST("notes") fun createOrUpdateNote(@Body note: Note): Call<Note> }
- **Шаг 5: Обработка режимов работы приложения**
- **Определение состояния сети**:
  - Используем ConnectivityManager для определения наличия интернет-соединения.
- **Переключение режимов**:
  - UI адаптируется для отображения локальных данных в офлайн режиме и обновления данных при восстановлении соединения.
- **Шаг 6: Тестирование приложения**
- **Тестирование в офлайн режиме**:
  - Проверка функциональности создания и просмотра заметок без интернета.
- **Тестирование синхронизации**:
  - Проверка корректности синхронизации данных между локальной базой и сервером.
- **Шаг 7: Отладка и оптимизация**
- **Профилирование**:
  - Использование Android Studio Profiler для определения и устранения узких мест в производительности.
- **Оптимизация процессов синхронизации**:
  - Минимизация количества сетевых запросов и объема передаваемых данных.
- Этот пример дает представление о пошаговом процессе разработки приложения с поддержкой онлайн и офлайн режимов работы, начиная от планирования архитектуры и заканчивая тестированием и оптимизацией.