УДК 004.41
ББК 30ц
И85

**Исабекова О.А. Разработка клиент-серверных мобильных приложений. Часть 1** [Электронный ресурс]: Учебное пособие / Исабекова О.А., Дворникова Е.М. — М.: МИРЭА – Российский технологический университет, 2025. — 1 электрон. опт. диск (CD-ROM)

Учебное пособие посвящено современным подходам к разработке клиент-серверных мобильных приложений. Приводятся необходимые теоретические сведения и практические задания для самостоятельного освоения материала и организации практических занятий.

Материал пособия необходим для освоения дисциплины «Разработка клиент-серверных мобильных приложений» и обеспечивает комплексное изучение ключевых аспектов создания современных мобильных приложений, включая проектирование пользовательского интерфейса, организацию сетевого взаимодействия, работу с данными и реализацию клиент-серверной архитектуры.

Предназначено для студентов вузов, обучающихся по направлению подготовки 09.03.04 «Программная инженерия», а также других направлений ИТ-профиля. Пособие может быть полезным для любых иных категорий читателей, интересующихся разработкой мобильных приложений, построенных на принципах клиент-серверной архитектуры.

Учебное пособие издается в авторской редакции.

Авторский коллектив: Исабекова Ольга Александровна, Дворникова Екатерина Михайловна

Рецензенты:

Гончаров Андрей Витальевич, к.т.н., доцент, заведующий кафедры «Системы автоматизированного управления» ФГБОУ ВО «Московский государственный университет технологий и управления имени К.Г. Разумовского (ПКУ)».

Семичевская Наталья Петровна, к.т.н., доцент, доцент кафедры «Информационные системы и цифровые технологии» ФГБОУ ВО «Московский государственный университет технологий и управления имени К.Г. Разумовского (ПКУ)».

Системные требования:
Наличие операционной системы Windows, поддерживаемой производителем.
Наличие свободного места в оперативной памяти не менее 128 Мб.
Наличие свободного места в памяти постоянного хранения (на жестком диске) не менее 30 Мб.
Наличие интерфейса ввода информации.
Дополнительные программные средства: программа для чтения pdf-файлов (Adobe Reader).
Подписано к использованию по решению Редакционно-издательского совета
МИРЭА — Российский технологический университет.
Обьем: 1.64 мб
Тираж: 10

**ISBN 978-5-7339-2723-7**





# ОГЛАВЛЕНИЕ







# ВВЕДЕНИЕ

Дисциплина «Разработка клиент-серверных мобильных приложений» направлена на формирование у студентов компетенции ПК-1.1: выполняет разработку и интеграцию программных модулей и компонент программных продуктов. В результате освоения дисциплины учащийся должен:

- знать основы разработки мобильных приложений, в том числе и для российского рынка, функционал интеграции мобильных модулей и компонентов приложений, возможность интеграции программных модулей и компонентов мобильных приложений, в том числе и для российского рынка;
- уметь разрабатывать мобильные приложения, в том числе для российского рынка, интегрировать мобильные модули и компоненты приложения;
- владеть навыками разработки мобильных приложений, в том числе и для российского рынка, навыками интеграции мобильных модулей и компонентов приложений, в том числе и для российского рынка.

В современном мире мобильные приложения прочно вошли во все сферы жизни, от развлечений и коммуникаций до бизнеса и образования. Значительная часть этих приложений построена по клиент-серверной архитектуре, что позволяет им эффективно взаимодействовать с удаленными серверами, обрабатывать большие объемы данных и предоставлять пользователям актуальную информацию.

Структура пособия охватывает все ключевые этапы разработки клиент-серверных мобильных приложений, начиная с создания пользовательского интерфейса (UI), где рассматриваются основные компоненты вёрстки и способы создания привлекательных и интуитивно понятных интерфейсов. Особое внимание уделяется работе со списками и использованию специализированных библиотек, позволяющих эффективно отображать и управлять большими объемами данных.

Значительный раздел посвящен организации сетевого взаимодействия, включая изучение протоколов обмена данными, принципов построения REST API и способов обеспечения безопасности передаваемой информации. Рассматриваются различные подходы к хранению данных на клиентской стороне, от простых локальных хранилищ до сложных баз данных.

Пособие знакомит с продвинутыми UI-компонентами, позволяющими создавать сложные и кастомизированные интерфейсы. Многопоточность в





Android рассматривается как необходимый инструмент для обеспечения отзывчивости приложения при выполнении длительных операций. Внимание уделяется вопросам архитектуры мобильных приложений, позволяющим создавать масштабируемые и поддерживаемые проекты.

Для разработки серверной части рассматриваются серверные платформы Kotlin и фреймворк Ktor, предлагающий мощные инструменты для создания REST API. Подробно освещаются вопросы unit-тестирования и Test Driven Development (TDD), позволяющие обеспечить высокое качество кода и надежность разрабатываемых приложений.

Таким образом, данное учебное пособие обеспечивает комплексное изучение разработки клиент-серверных мобильных приложений, охватывая все необходимые этапы от проектирования UI до тестирования и развертывания серверной части. Материал, представленный в пособии, позволит студентам приобрести не только теоретические знания, но и практические навыки, необходимые для успешной работы в сфере разработки мобильных приложений.

В первой части учебного пособия рассматриваются следующие темы:

- Введение в UI.
- Базовые компоненты Android.
- Работа со списками.
- Основы сетевого взаимодействия.





# ГЛАВА 1. ВВЕДЕНИЕ В UI

## 1.1 Основы верстки интерфейса пользователя с использованием XML

В Android-разработке, XML играет ключевую роль в создании пользовательского интерфейса (UI). В отличие от императивного подхода, когда UI создается программно, XML позволяет декларативно описывать структуру и внешний вид элементов интерфейса. Это делает код более читаемым, облегчает поддержку и позволяет отделить логику приложения от его представления.

XML-файл состоит из элементов, каждый из которых определяется тэгами, атрибутами и значениями:

- Тэги (Tags): Тэгами в XML называют специальные текстовые метки, которые обозначают некоторые объекты. Тэги используются для определения типа элемента интерфейса. Каждый тег — отдельный элемент пользовательского интерфейса. В зависимости от типа элемента он имеет разные атрибуты. Например, <TextView> представляет текстовое поле, <Button> - кнопку, <ImageView> - изображение. Тэги бывают открывающими (<TextView>) и закрывающими (</TextView>). Элемент может быть пустым и иметь сокращенную запись: <ImageView ... />. Параметров внутри каждого тега может быть неограниченное количество. Но они не должны повторяться.
- Атрибуты (Attributes): Атрибуты предоставляют дополнительную информацию об элементе. Они задаются в виде attribute_name="attribute_value" внутри открывающего тэга. Например, android:text="Привет, мир!" устанавливает текст для TextView.
- Значения (Values): Значения атрибутов определяют конкретные свойства элемента. Значения могут быть строками, числами, ссылками на ресурсы (например, @string/app_name для строки из файла ресурсов) или константами (например, match_parent).

Рассмотрим пример:

```
<TextView
    android:id="@+id/my_MIREA"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Привет, МИРЭА!"
    android:textSize="30sp"
```

```
    android:textColor="#FF0000" />
```

В этом примере:

- TextView — это тэг, определяющий текстовое поле.
- android:id, android:layout_width, android:layout_height, android:text, android:textSize, android:textColor - это атрибуты.
- @+id/my_MIREA, wrap_content, wrap_content, "Привет, МИРЭА!", 30sp, #FF0000 - это значения атрибутов.

Элементы интерфейса могут быть вложены друг в друга. Вложенность позволяет создавать сложные иерархические структуры UI. Родительский элемент называется контейнером (Layout), а вложенные элементы - дочерними элементами. Контейнеры отвечают за расположение и упорядочивание дочерних элементов.

В основе каждого элемента, который вы видите на экране мобильного приложения Android, лежит класс View. **View** – это базовый класс для всех компонентов UI, и именно он определяет, как элемент будет отображаться и взаимодействовать с пользователем. Каждый элемент пользовательского интерфейса, будь то кнопка, текстовое поле или изображение, является наследником класса View. Этот класс содержит основные компоненты:

- Логика отрисовки элемента на экране: View определяет, как элемент будет нарисован на экране. Это включает в себя отрисовку текста, фона, границ и других визуальных элементов.
- Базовые параметры: View содержит базовые параметры, которые должны быть у каждого UI-элемента:
    - android:layout_width (ширина): определяет ширину элемента.
    - android:layout_height (высота): определяет высоту элемента.
    - android:visibility (видимость): определяет, видим ли элемент (visible), не видим, но занимает место (invisible) или полностью скрыт и не занимает место (gone).
    - android:padding (отступы): определяет отступы вокруг содержимого элемента.
- Логика анимаций и трансформаций: View предоставляет механизмы для анимации и трансформации элемента во время работы приложения. Можно изменять положение, размер, прозрачность и другие свойства элемента.

ViewGroup — это абстрактный класс, который также наследуется от View. Его основная задача – служить контейнером для других View. ViewGroup

определяет, как его дочерние View будут расположены и упорядочены на экране. Другими словами, это основа для создания макетов (layouts).

ViewGroup отвечает за компоновку видимых элементов экрана. Он определяет правила, по которым дочерние View будут размещены внутри него. ViewGroup может содержать другие ViewGroup, создавая иерархическую структуру UI. Например, представьте себе комнату. Комната – это ViewGroup (контейнер), а мебель (стулья, столы, диваны) – это View. ViewGroup (комната) определяет, как мебель будет расположена в пространстве.

Одним из простейших наследников ViewGroup является **FrameLayout**. Он достаточно прост в использовании, но менее гибок, чем другие Layouts. Так, FrameLayout размещает все дочерние View в левом верхнем углу один над другим. Последний добавленный элемент будет отображаться поверх предыдущих. Обычно он используется, когда нужно отобразить один элемент, или когда элементы должны перекрывать друг друга.

Рассмотрим пример кода:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/image22" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="РТУ МИРЭА"
        android:textColor="#FFFFFF"
        android:layout_gravity="center" />
</FrameLayout>
```

Давайте разберем код. В данном случае ImageView будет отображаться под TextView. TextView с текстом "РТУ МИРЭА" будет отображаться поверх изображения и будет выровнен по центру. Атрибут layout_gravity используется для позиционирования TextView внутри FrameLayout.

TextView – это наследник класса View, который отвечает за отображение текста на экране пользователя. В отличие от FrameLayout, TextView сам по себе не является контейнером (т.е. не может содержать другие View) и наследником ViewGroup.

Параметры TextView:

- android:text: определяет текст для отображения.
- android:textColor: устанавливает цвет текста.
- android:textSize: устанавливает размер текста.

Рассмотрим пример:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="РТУ МИРЭА"
    android:textColor="#FF000000"
    android:textSize="20sp" />
```

В примере TextView отобразит черным цветом текст "РТУ МИРЭА" размером 20sp.

LinearLayout располагает все вложенные элементы один за другим, либо вертикально, либо горизонтально. Это очень распространенный и полезный Layout.

Параметры LinearLayout:

- android:orientation: определяет направление расположения элементов.
- android:orientation="vertical": элементы располагаются один под другим (вертикально).
- android:orientation="horizontal": элементы располагаются рядом друг с другом (горизонтально). Это значение по умолчанию.

Таким образом, если не указывать ориентацию, то по умолчанию будет принята горизонтальная ориентация, однако, ее можно изменить, указав параметр android:orientation="vertical".

Рассмотрим пример использования LinearLayout:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Привет!" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Введите текст" />
```

```
</LinearLayout>
```

В этом примере параметр android:orientation не установлен, следовательно, используется горизонтальная ориентация и TextView и EditText будут расположены друг под другом. Если бы android:orientation был установлен в horizontal, они бы отображались рядом.

Рассмотрим параметры для организации отступов для элементов экрана:

- android:paddingHorizontal - отступы по горизонтали
- android:paddingVertical— вертикальные отступы
- android:paddingLeft — левый отступ
- android:paddingTop — верхний отступ
- android:paddingRight — правый отступ
- android:paddingBottom — нижний отступ
- android:paddingStart — отступ от начала экрана
- android:paddingEnd — отступ от конца экрана.

Для позиционирования элементов с помощью параметра android:layout_gravity можно использовать пары значений left/right и start/end. При этом параметры left/right определяют, к какой стороне экрана должен быть прижат элемент, а start/end указывают, к началу или к концу строки надо прижать элемент.

При определении размеров элементов в коде необходимо учитывать, что если вы используете для разработки прототипа интерфейса приложения сервис Figma, то размеры элементов в нем указываются в пикселях px, а в XML-разметке все размеры необходимо указывать в sp или dp.

Иногда необходимо сделать так, чтобы размер элемента зависел от размера экрана. В этом случае для элементов, вложенных в LinearLayout, можно воспользоваться параметром android:layout_weight, установив так называемый «вес» элемента. «Вес» всех элементов суммируется, и каждому из элементов выделяется столько места, сколько его «вес» занимает в общей пропорции. При этом для элемента, у которого указан вес, должен быть установлен размер 0dp.

## 1.2. Ресурсы приложения: строки, размеры, цвета и изображения

Android использует ресурсы для хранения данных, которые не являются кодом, таких как строки, размеры, цвета, изображения и макеты. Использование ресурсов упрощает поддержку, локализацию и изменение внешнего вида вашего приложения.

Строковые ресурсы предоставляют текстовые константы для вашего приложения. В них можно хранить:

- текст для отображения на экране (например, названия кнопок, заголовки);
- сообщения пользователю;
- подсказки;
- форматированный текст (с использованием HTML-тегов).

В корневом каталоге приложения есть папка res/values. В этой папке находится файл strings.xml, который содержит список всех строковых ресурсов приложения.

Пример файла strings.xml:

```
<resources>
    <string name="app_name">Мое приложение</string>
      <string name="button_text">Нажми меня</string>
</resources>
```

Чтобы использовать строковый ресурс в XML-разметке, необходимо воспользоваться @string/resource_name:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/app_name" />
```

Наиболее часто используемые размеры, например, отступы, ширина, высота элемента, размер текста, также рекомендуется сохранять в ресурсах. Для этого используется файл dimens.xml в папке res/values.

Пример файла dimens.xml:

```
<resources>
    <dimen name="activity_margin">15dp</dimen>
    <dimen name="text_size">12sp</dimen>
    <dimen name="button_width">100dp</dimen>
</resources>
```

Теперь мы можем использовать размеры в XML-разметке:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="@dimen/text_size"
    android:padding="@dimen/activity_margin" />
```





Цветовые ресурсы определяют цвета, которые используются в приложении. Они хранятся в файле colors.xml в папке res/values.

Пример файла colors.xml:

```
<resources>
    <color name="textColor">#000000</color>
    <color name="backgroundColor">#FFFFFF</color>
</resources>
```

Использование в XML-разметке:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="@color/textColor"
    android:background="@color/backgroundColor" />
```

Drawable ресурсы – это графические ресурсы, такие как изображения (PNG, JPG), векторная графика (XML), анимации и слои. Они хранятся в различных подкаталогах папки res/drawable/, например, res/drawable-mdpi/, res/drawable-hdpi/, res/drawable-xhdpi/ (для разных плотностей экранов). В современных проектах часто используются векторные изображения (SVG), помещенные в drawable/.

Пример использования в XML-разметке:

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/my_image" />
```

## 1.3 Типы масштабирования изображений (scaleType в ImageView)

Атрибут **android:scaleType** в ImageView определяет, как изображение будет масштабироваться, чтобы соответствовать размерам ImageView.

Основные типы:

- fitXY: Соотношение сторон не сохраняется. Изображение растягивается, чтобы заполнить ImageView, что может привести к искажениям.
- fitCenter: Соотношение сторон сохраняется. Изображение масштабируется, чтобы полностью поместиться в ImageView, при этом остается центрированным. Могут оставаться пустые поля.
- centerCrop: Соотношение сторон сохраняется. Изображение масштабируется так, чтобы полностью заполнить ImageView, обрезая

части, которые не помещаются. Часто используется для отображения аватарок.

- centerInside: Соотношение сторон сохраняется. Изображение не масштабируется, если оно меньше ImageView. Если изображение больше ImageView, оно уменьшается, чтобы поместиться, оставаясь центрированным.
- center: Изображение не масштабируется и центрируется в ImageView. Если изображение больше, чем ImageView, его части будут обрезаны.
- matrix: Масштабирование определяется матрицей преобразования (используется реже).

Рассмотрим пример использования масштабирования:

```
<ImageView
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:src="@drawable/my_image"
    android:scaleType="centerCrop" />
```

В этом случае изображение будет находится в центре ImageView и будет масштабировано таким образом, чтобы заполнить полностью данный ImageView. Для этого изображение будет соответственно увеличиваться или уменьшаться до тех пор, пока полностью не заполнит ImageView по наибольшей стороне View элемента.

В Kotlin коде можно динамически изменять свойства ImageView.

Например, установка цвета фона:

```
val image: ImageView = findViewById(R.id.my_image_view)
image.setBackgroundColor(getColor(R.color.black))
```

Или установка типа масштабирования:

```
image.scaleType = ImageView.ScaleType.CENTER_CROP
```

Например, установка изображения из ресурсов:

```
image.setImageResource(R.drawable.poster)
```

## 1.4. Взаимодействие с View: обработка нажатий (OnClickListener)

Чтобы приложение взаимодействовало с пользователем, его необходимо сделать интерактивным с помощью обработки различных действий пользователя, таких как нажатия на кнопки, клики по элементам списка и т.д. Самый распространенный способ реализации этих действий - использование OnClickListener.

Для того, чтобы добавить слушатель нажатия, сначала необходимо присвоить уникальный идентификатор (ID) элементу, к которому вы хотите добавить слушатель. Для этого используется атрибут android:id.

Пример:

```
<Button
    android:id="@+id/my_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Нажми меня" />
```

Существует несколько способов добавить OnClickListener. Рассмотрим каждый их них.

Первый способ — это использование анонимного класса.

Пример:

```
val button: Button = findViewById(R.id.my_button)
button.setOnClickListener(object : View.OnClickListener {
    override fun onClick(v: View?) {
        // Действия, которые будут выполнены при нажатии на кнопку
            }
})
```

Давайте подробно разберем, что происходит в коде примера выше. В первой строке объявляется переменная button типа Button. Ключевое слово val означает, что это неизменяемая переменная (ее нельзя переназначить после инициализации). Функция findViewById(R.id.my_button) ищет View в текущем макете по его идентификатору (id). R.id.my_button – это ссылка на id, который мы присвоили кнопке в XML-макете. findViewById возвращает найденный элемент. Найденная кнопка присваивается переменной button. Во второй строке с помощью метода button.setOnClickListener(…) устанавливается слушатель нажатия (OnClickListener) на кнопку. Этот слушатель будет уведомлен, когда пользователь нажмет на кнопку. Мы создаем объект анонимного класса, который реализует интерфейс View.OnClickListener. Интерфейс View.OnClickListener требует реализации одного метода: onClick(). В третьей строке происходит реализация метода onClick() интерфейса View.OnClickListener. Этот метод будет вызван, когда пользователь нажмет на кнопку. Параметром метода onClick() является v: View? Он представляет View, на который нажали (в данном случае, это наша кнопка). Тип View? означает, что параметр может быть null.

Второй способ — использование лямбда-выражение. Он более компактный и современный.

Пример:

```
val button: Button = findViewById(R.id.my_button)
button.setOnClickListener {
    // Действия, которые будут выполнены при нажатии на кнопку
}
```

Давайте также подробно разберем, что происходит в коде примера выше.

Первая строка идентична строке из примера с анонимным классом. Во второй мы устанавливаем слушатель нажатия на кнопку, но с помощью лямбда-выражения { ... }. Напомним, что лямбда-выражение – это анонимная функция, которую можно передать в качестве аргумента другой функции. В нашем примере, лямбда-выражение заменяет анонимный класс, реализующий интерфейс View.OnClickListener. Kotlin автоматически понимает, что лямбда-выражение должно соответствовать типу View.OnClickListener, так как setOnClickListener() ожидает аргумент этого типа.

Третий способ - реализация OnClickListener на Activity. В этом способе Activity реализует интерфейс View.OnClickListener, и необходимо устанавливаете this в качестве слушателя для кнопки. Данный метод используется редко.

## 1.5 Навигация и взаимодействие между компонентами приложения

В Android-разработке навигация и взаимодействие между различными частями приложения (экранами, сервисами и т.д.) является ключевым аспектом. Android SDK предоставляет несколько основных компонентов для этого, а также механизм Intent для обмена информацией и запуска этих компонентов.

В Android SDK есть четыре основных компонента, которые являются строительными блоками для большинства приложений:

- BroadcastReceiver (широковещательный приемник). Это компонент, который прослушивает широковещательные сообщения (или события) из разных источников: других приложений, системы (например, изменение состояния сети, низкий заряд батареи) или вашего собственного приложения. Это позволяет приложению реагировать на системные события или сообщения от других приложений, даже когда оно не активно.
- Service (служба). Это компонент, работающий в фоновом режиме без видимого пользовательского интерфейса. Службы используются для

выполнения длительных операций, таких как загрузка данных из сети, воспроизведение музыки в фоновом режиме или выполнение периодических задач.

- ContentProvider (поставщик контента). Это компонент, который предоставляет структурированный доступ к данным вашего приложения другим приложениям. Это позволяет обмениваться данными с другими приложениями (например, контактами, изображениями, музыкой) безопасным и контролируемым образом.
- Activity (активность). Это основной компонент Android SDK, представляющий один экран пользовательского интерфейса. Через Activity происходит взаимодействие между пользователем и приложением. Большинство приложений состоит из нескольких Activity, между которыми пользователь может перемещаться.

Итак, Activity – это основной компонент, отвечающий за взаимодействие с пользователем. Он представляет собой один экран в приложении.

Чтобы создать новую Activity, необходимо унаследоваться от базового класса androidx.appcompat.app.AppCompatActivity. Этот класс предоставляет базовую функциональность для работы с пользовательским интерфейсом, панелью инструментов и другими элементами.

Пример:

```
import androidx.appcompat.app.AppCompatActivity
  import android.os.Bundle
  class MyActivity : AppCompatActivity() {
      override fun onCreate(savedInstanceState:
Bundle?) {
          super.onCreate(savedInstanceState)
          setContentView(R.layout.activity_my)    //
Связываем Activity с XML-макетом
      }
  }
```

Давайте рассмотрим код более подробно. С помощью import androidx.appcompat.app.AppCompatActivity импортируем класс AppCompatActivity для наследования. Затем с помощью class MyActivity: AppCompatActivity() { ... } определяем новый класс MyActivity, который наследуется от AppCompatActivity. Потом переопределяем метод onCreate(), который вызывается при создании Activity. Это место, где происходит инициализация Activity, например, загрузка макета. Затем вызываем метод

onCreate() родительского класса (AppCompatActivity). Это необходимо для правильной инициализации базовой функциональности. И, наконец, с помощью setContentView(R.layout.activity_my) связываем Activity с XML-макетом, определяющим ее пользовательский интерфейс. R.layout.activity_my – это ссылка на XML-файл макета с именем activity_my.xml, расположенный в папке res/layout/.

Каждая Activity должна быть объявлена в файле AndroidManifest.xml с помощью тега <activity/>. Это необходимо для того, чтобы система Android знала о существовании Activity и могла ее запустить.

Activity проходит через различные состояния в течение своего жизненного цикла (создание, запуск, приостановка, остановка, уничтожение). Метод onCreate() вызывается системой в самом начале процесса создания Activity.

Открытие новой Activity помещает ее на вершину стека Activity. Предыдущая Activity переходит в режим stop. Task – это набор Activity, организованных в стек. В стандартной ситуации каждое приложение имеет свой Task и свой стек.

Activity проходит через различные состояния в течение своей жизни, от момента создания до уничтожения. Система Android управляет этими состояниями и вызывает соответствующие методы жизненного цикла. Понимание этих методов позволяет выполнять определенные действия в нужное время, чтобы гарантировать, что приложение работает правильно.

## 1.6. Методы жизненного цикла Activity

Рассмотрим каждый метод жизненного цикла Activity от момента создания до закрытия (рис. 1).

Метод onCreate() вызывается системой один раз при создании Activity или ее перезапуске после уничтожения системой. Именно здесь происходит основная инициализация Activity:

- создание пользовательского интерфейса (UI) с помощью setContentView(R.layout.your_layout);
- инициализация переменных и объектов, необходимых для работы Activity.
- настройка UI-элементов (например, установка слушателей нажатий на кнопки);
- подключение данных к UI-элементам (например, заполнение списка данными из базы данных).

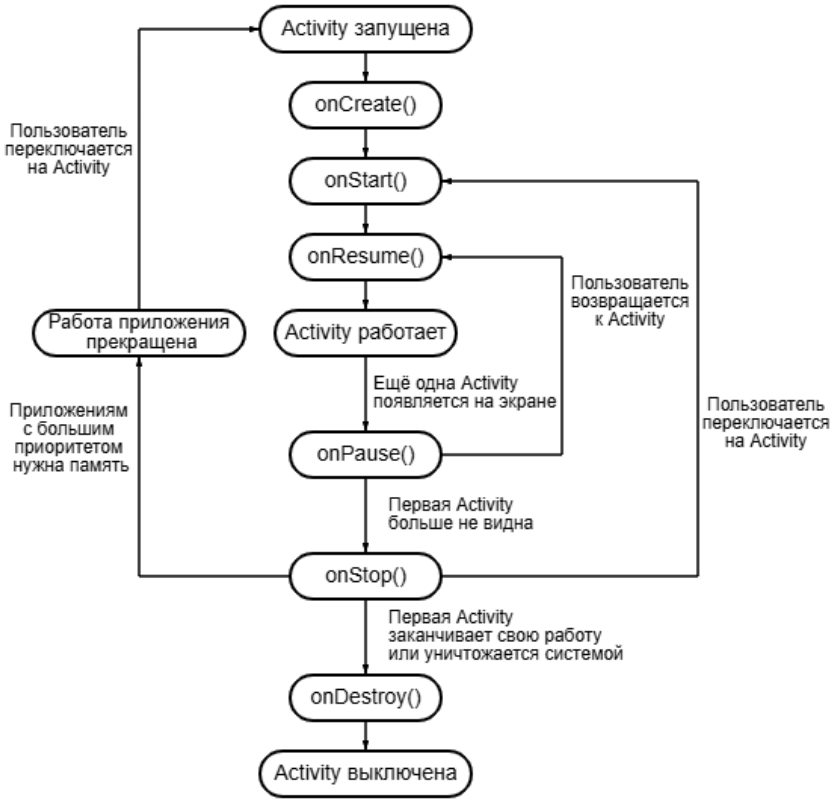


*Рисунок 1. Жизненный цикл Activity*

Однако, в данном методе пользователь еще не видит UI, так как Activity только подготавливается,

Метод onStart() вызывается, когда Activity становится видимой пользователю, но еще не находится в состоянии взаимодействия. Он вызывается после onCreate() (при первом запуске) или после onRestart() (если Activity возвращается на передний план). Здесь можно выполнять действия, связанные с отображением UI:

- запуск анимаций, переходов;
- запуск потоков или корутин для загрузки данных, необходимых для отображения;
- регистрация слушателей системных событий.

Метод onResume() вызывается, когда Activity находится на переднем плане и пользователь может с ней взаимодействовать. Это состояние активно до тех пор, пока пользователь не перейдет на другую Activity или не выключит экран. Activity находится в активном состоянии, могут выполняться следующие действия:

- возобновление приостановленных процессов (например, воспроизведение музыки);
- регистрация слушателей (например, для датчиков);

- обновление UI, если данные изменились.

Метод onPause() вызывается, когда Activity теряет фокус, но еще не полностью невидима. Это может произойти, если другая Activity отображается поверх текущей (например, диалоговое окно), или если пользователь переключается на другое приложение. Выполняются действия по освобождению ресурсов и приостановке процессов, чтобы снизить нагрузку на систему:

- приостановка анимаций и потоков;
- остановка воспроизведения звука или видео;
- отмена регистрации слушателей;
- сохранение несохраненных данных (например, черновиков).

Метод onStop() вызывается, когда Activity становится полностью невидимой для пользователя. Это может произойти, если Activity уничтожается или если другая Activity полностью перекрывает ее. Поэтому происходит освобождение ресурсов, которые не нужны, когда Activity невидима:

- освобождение больших объемов памяти;
- сохранение данных в постоянное хранилище (например, в базу данных);
- отмена сетевых запросов.

Метод onDestroy() вызывается непосредственно перед уничтожением Activity. Это может произойти, если система Android решает освободить память или если вы явно вызываете метод finish(). Выполняется окончательная очистка:

- освобождение всех ресурсов, которые еще не были освобождены;
- отмена регистрации всех слушателей;
- закрытие соединений с базой данных.

## 1.7 Стили и темы

Стили и темы в Android позволяют централизованно управлять внешним видом элементов пользовательского интерфейса приложения. Использование стилей и тем делает код более организованным, упрощает поддержку и обеспечивает согласованный пользовательский опыт.

Стиль (Style) — это набор значений атрибутов, применяемых к View-элементу. Он определяет, как элемент будет выглядеть: цвет текста, размер шрифта, фон, отступы и т.д. Стили, как и другие ресурсы (строки, размеры, цвета), описываются в XML-формате.

Преимущества использования стилей:

- переиспользование: можно определить стиль один раз и применять его к нескольким View-элементам, избегая дублирования кода;
- централизованное управление: изменение стиля автоматически отражается на всех View-элементах, которые его используют;
- читаемость кода: стили делают XML-макеты более чистыми и понятными.

Файлы стилей хранятся в папке res/values. Если папка values не существует, ее нужно создать.

Формат объекта стиля в файле styles.xml выглядит следующим образом:

```
<resources>
    <style name="ИмяСтиля">
        <item
name="имяАтрибута1">ЗНАЧЕНИЕ_АТРИБУТА</item>
        <item
name="имяАтрибута2">ЗНАЧЕНИЕ_АТРИБУТА</item>
        ...
    </style>
</resources>
```

Разберем элементы этого формата:

- <resources>: корневой элемент XML-файла ресурсов. Содержит все стили и темы.
- <style>: тег, обозначающий объект стиля.
- name="ИмяСтиля": атрибут name задает имя стиля. Имя должно быть уникальным в пределах файла styles.xml. Имя используется для ссылки на стиль в XML-макетах.
- <item name="имяАтрибута1">ЗНАЧЕНИЕ_АТРИБУТА</item>: тег item определяет значение конкретного атрибута для стиля.
- name="имяАтрибута1»: атрибут name указывает имя атрибута, который нужно установить. Например, android:textColor, android:textSize, android:background.
- ЗНАЧЕНИЕ_АТРИБУТА: значение, которое будет присвоено атрибуту. Это может быть цвет (например, #FF0000), размер (например, 14sp), ссылка на строковый ресурс (например, @string/my_string), и т.д.

Пример. Необходимо создать стиль для текстовых полей (TextView), который будет устанавливать синий цвет текста и размер шрифта 18sp:

```
<resources>
```

```
    <style name="MyTextViewStyle">
        <item name="android:textColor">#0000FF</item>
        <item name="android:textSize">18sp</item>
    </style>
</resources>
```

Рассмотрим построчно данный код.

- <style name="MyTextViewStyle">: создает стиль с именем MyTextViewStyle.
- <item name="android:textColor">#0000FF</item>: устанавливает цвет текста в синий (#0000FF).
- <item name="android:textSize">18sp</item>: устанавливает размер шрифта в 18sp.

Чтобы применить стиль к View-элементу в XML-макете, необходимо использовать атрибут style:

Пример:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Привет, РТУ МИРЭА"
    style="@style/MyTextViewStyle" />
```

Здесь атрибут style="@style/MyTextViewStyle" указывает, что к TextView нужно применить стиль MyTextViewStyle. Теперь текст TextView будет синим и иметь размер 18sp.

Стили могут наследовать атрибуты от других стилей. Это позволяет создавать иерархию стилей и избегать дублирования кода.

Для наследования стиля используется атрибут parent в теге <style>:

Пример:

```
<style name="MyTextViewStyle.Bold">
    <item name="android:textStyle">bold</item>
</style>
```

В этом примере стиль MyTextViewStyle.Bold наследует все атрибуты от стиля MyTextViewStyle и добавляет атрибут android:textStyle="bold", делая текст жирным. Обратите внимание на использование точки (.) в имени стиля для обозначения наследования.

Рассмотрим пример применение наследованного стиля:

```
<TextView
    android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
android:text="Привет, РТУ МИРЭА"
style="@style/MyTextViewStyle.Bold" />
```

Теперь текст TextView будет синим, иметь размер 18sp и быть жирным.

Если стиль содержит атрибуты, несвойственные элементу, к которому применяется данный стиль, то они будут игнорироваться.

Если указать значение для атрибута, который уже присутствует в стиле, после обозначения стиля, то для данного View-элемента указанное значение будет проигнорировано и система при отрисовке элемента воспользуется последним указанным значением этого атрибута.

Темы – это набор стилей, которые применяются ко всему приложению или к отдельным Activity. Темы определяют общий внешний вид приложения, например, цвета, шрифты и стили элементов UI.

Темы – это мощный механизм в Android, позволяющий централизованно управлять стилем приложения. В отличие от индивидуальных стилей, применяемых к отдельным View-элементам, тема влияет на внешний вид всего приложения, конкретной Activity или даже целой иерархии View. Это упрощает поддержание единообразного стиля и быстрое изменение дизайна.

При создании нового проекта Android Studio (за исключением шаблона "No Activity"), автоматически генерируется файл themes.xml в папке values. Именно в этом файле определяются основные атрибуты темы приложения.

Material Design предлагает набор стандартных цветовых атрибутов, которые помогают реализовать современный и интуитивно понятный интерфейс. Рассмотрим ключевые из них:

- colorPrimary и colorSecondary: эти атрибуты представляют основные цвета бренда. Они используются для элементов, которые наиболее часто привлекают внимание пользователя (например, панель навигации, кнопки, акцентные элементы).
- colorPrimaryVariant и colorSecondaryVariant: эти атрибуты предназначены для обозначения более светлых или тёмных оттенков colorPrimary и colorSecondary соответственно. Они используются для создания визуальной иерархии и глубины в интерфейсе.
- colorOnPrimary и colorOnSecondary: эти атрибуты определяют цвет текста и иконок, которые отображаются поверх colorPrimary и colorSecondary. Они обеспечивают достаточный контраст для читаемости и удобства использования.

В папке values также находится файл colors.xml, который содержит предопределенные цвета, предлагаемые Android Studio по умолчанию. Этот файл служит хранилищем цветовых значений, на которые ссылаются атрибуты в themes.xml:

```
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```

Чтобы применить созданную тему ко всему приложению, необходимо указать ее в файле AndroidManifest.xml. Необходимо добавить атрибут android:theme к тегу <application>:

Пример:

```
<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.YourAppName">
    </application>
```

Необходимо заменить @style/Theme.YourAppName на имя темы, определенной в themes.xml.

Android поддерживает ночной режим, который позволяет пользователям переключаться на тёмную цветовую схему для снижения нагрузки на глаза в условиях низкой освещенности. Для этого создаются отдельные темы для светлого и темного режимов.

В Android Studio сгенерируется файл themes.xml (night) в папке values. В этом файле необходимо переопределить цветовые атрибуты, чтобы они соответствовали темной теме. Например, можно сделать фон темным, а текст светлым.

Операционная система автоматически выберет соответствующую тему в зависимости от системных настроек пользователя. Также есть возможность программно управлять темой приложения, используя AppCompatDelegate.setDefaultNightMode().

# ГЛАВА 2. БАЗОВЫЕ КОМПОНЕНТЫ ANDROID

## 2.1. Context

Context – это фундаментальный абстрактный класс в Android, предоставляющий доступ к системным ресурсам и функциям. Он служит связующим звеном между приложением и операционной системой, позволяя выполнять разнообразные операции, такие как запуск Activity, отправка Intent, запуск Service и многое другое. Context является базовым классом для Application, Activity и Service, что подчеркивает его центральную роль в архитектуре Android.

Application – это базовый класс для всех приложений Android. Он предназначен для хранения глобального состояния приложения и предоставляет доступ к системным сервисам и ресурсам. Важно отметить, что класс Application (или ваш подкласс Application) создается системой до любых других компонентов вашего приложения.

Можно создать свой собственный класс Application, расширив базовый класс и переопределив необходимые методы для управления глобальным состоянием и ресурсами. Чтобы указать Android использовать ваш пользовательский класс Application, необходимо добавить атрибут android:name в тег <application> в файле AndroidManifest.xml:

```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        // Инициализация глобальных ресурсов и состояний
    }
}
```

В AndroidManifest.xml:

```
<application
    android:name=".MyApplication"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.MyApp">
</application>
```

Context предоставляет методы для взаимодействия с системными ресурсами и другими приложениями. Рассмотрим некоторые примеры:

- getResources(): возвращает объект Resources, который позволяет получить доступ ко всем ресурсам приложения, таким как строки, изображения, макеты и т.д.
- getAssets(): возвращает объект AssetManager, который используется для доступа к вспомогательным файлам, хранящимся в директории assets вашего проекта. Это могут быть видео, аудио, базы данных и другие файлы, которые не обрабатываются системой ресурсов Android.
- getPackageManager(): возвращает объект PackageManager, который позволяет получить информацию о пакете приложения, установленных приложениях и разрешениях.
- getString(resourceId: Int): возвращает строковый ресурс по его ID. Этот метод особенно важен для локализации приложения.

Существуют два основных типа Context:

Application Context: Привязан к жизненному циклу всего приложения. Он доступен на протяжении всего времени работы приложения и используется для операций, которые не зависят от конкретной Activity (например, инициализация глобальных ресурсов, запуск фоновых задач). Получить экземпляр Application Context можно вызвав метод getApplicationContext() или приведя экземпляр класса Application к Context.

Activity Context: Привязан к жизненному циклу конкретной Activity (экрана). Он должен использоваться только во время существования Activity для операций, связанных с пользовательским интерфейсом и обработкой событий. Получить экземпляр Activity Context можно, приведя объект класса Activity к Context, используя ключевое слово this (внутри Activity) или используя метод getActivity() (во фрагменте).

Выбор правильного типа Context критически важен для предотвращения утечек памяти и обеспечения стабильной работы приложения. В общем случае:

- используйте Application Context для долгоживущих задач и ресурсов, которые не зависят от Activity.
- используйте Activity Context для операций, связанных с пользовательским интерфейсом и жизненным циклом Activity.

Android предоставляет мощный механизм для отделения текстовых ресурсов от кода. Это позволяет легко локализовать приложение для разных языков и регионов. Вместо жесткого кодирования строк в коде, можно

определить их в файле strings.xml (расположенном в директории res/values/) и обратиться к ним по ID ресурса.

Пример:

```
val message = getString(R.string.roosevelt_phrase)
println(message)
```

Android автоматически подставит соответствующую локализацию строки в зависимости от настроек устройства пользователя.

## 2.2 Intent

Класс Intent (в переводе с английского — "намерение") используется для запуска другого компонента приложения (например, другой Activity, Service или BroadcastReceiver). Это основной механизм навигации и взаимодействия между компонентами в Android.

Основными элементами Intent являются:

- Действие (Action): определяет, что намерение должно сделать. Например, запустить браузер (ACTION_VIEW), сделать звонок (ACTION_DIAL), отправить данные (ACTION_SEND). Действия обычно представлены строковыми константами.
- Данные (Data): определяет, над чем намерение должно действовать. Обычно это URI (Uniform Resource Identifier), представляющий ресурс, например, веб-страница, номер телефона или файл.Класс Uri используется для представления данных в структурированном формате, обычно в виде URL-адреса.
- Дополнения (Extras): это набор пар "ключ-значение" для передачи дополнительной информации в целевой компонент. Похоже на ассоциативный массив.

Intent бывает двух типов: явный и неявный.

Явный (Explicit) Intent используется если указывается, какой компонент (например, конкретную Activity) необходимо запустить, то есть известно, какой компонент должен обработать Intent. Например:

```
val intent = Intent(this, SettingsActivity::class.java)
startActivity(intent)
```

Таким образом мы открыли новую Activity используя явный Intent.В первой строке мы создаем новый Intent, при этом this в это — Context текущей Activity. Context – это базовый абстрактный класс, который предоставляет доступ к различным ресурсам системы: файлам, базам данных, службам, и т.д. Он также используется для запуска Activity, Service и отправки Broadcast. Activity является

одним из компонентов Android, который реализует класс Context. SettingsActivity::class.java - класс Activity, которую нужно запустить (SettingsActivity). ::class.java – это способ получить объект Class для класса SettingsActivity. Вторая строка запускает Activity, указанную в Intent.

Неявный (Implicit) Intent используется, если вы не знаете, какой компонент хотите открыть, но вы знаете, какое действие хотите совершить. Другими словами, когда вы не знаете, какой конкретный компонент должен обработать Intent. Например, откроем главную страницу РТУ МИРЭА, не указывая при этом приложения для открытия страницы.

```
val url = Uri.parse("https://mirea.ru")
val intent = Intent(Intent.ACTION_VIEW, url)
startActivity(intent)
```

В первой строке создает Uri объект из строки URL, затем создаем новый Intent Здесь необходимо обратить внимание на Intent.ACTION_VIEW, который указывает, что действие – просмотр данных. Последняя строка запускает Activity, которая может обработать Intent.ACTION_VIEW для данного Uri. Таким образом, указание на конкретное приложение, которое должно открыть ссылку, отсутствует. Пользователь выберет необходимое приложение из предложенных системой.

Таким образом, понимание компонентов Android (Activity, Service, BroadcastReceiver, ContentProvider) и механизма Intent является ключевым для разработки Android приложений. Intent позволяет перемещаться между экранами, запускать фоновые задачи и взаимодействовать с другими приложениями.

Intent Filter (фильтр намерений) – это механизм, который позволяет компоненту (обычно Activity, Service или BroadcastReceiver) объявить о своих возможностях и типах Intent, которые он может обрабатывать. Фильтр описывается в файле AndroidManifest.xml в теге <intent-filter> внутри объявления компонента.

Фильтр Intent определяет соответствие между Intent и компонентом на основе трех основных параметров:

Action: указывает имя действия, которое компонент может выполнить. Например, ACTION_VIEW (отображение данных), ACTION_SEND (отправка данных), ACTION_CALL (вызов).

Data: указывает тип данных, которые компонент может обрабатывать. Описывается с помощью атрибутов:

- scheme: схема URI (например, http, https, mailto, content).

- host: имя хоста (например, www.example.com).
- path: путь к ресурсу (например, /images/logo.png).
- mimeType: тип MIME данных (например, text/plain, image/jpeg).

Category: указывает дополнительную информацию о типе Intent. Наиболее распространенные категории:

- LAUNCHER: указывает, что Activity должна отображаться в списке приложений (Launcher). Это главная Activity приложения.
- BROWSABLE: указывает, что Activity может быть запущена из веб-браузера.
- DEFAULT: указывает, что Activity является обработчиком по умолчанию для Intent с определенным Action и Data.

Рассмотрим примеры Action.

- ACTION_VIEW: отображает данные, указанные в URI (например, веб-страницу, изображение).
- ACTION_SENDTO: открывает приложение для отправки сообщения пользователю, указанному в URI (например, отправка SMS).
- ACTION_CALL: осуществляет телефонный звонок на номер, указанный в URI.
- ACTION_SEND: позволяет поделиться данными с помощью разных приложений (текст, изображение и т.д.).

Рассмотрим пример Intent Filter в AndroidManifest.xml:

```
<activity
    android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action
android:name="android.intent.action.MAIN" />
        <category
android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action
android:name="android.intent.action.VIEW" />
        <category
android:name="android.intent.category.DEFAULT" />
        <category
android:name="android.intent.category.BROWSABLE" />
```





```
            <data android:scheme="http"
                  android:host="www.primer.com"
                  android:pathPrefix="/articles" />
        </intent-filter>
    </activity>
```

В данном случае MainActivity является точкой входа в приложение (MAIN и LAUNCHER). MainActivity может отображать веб-страницы с www.primer.com, начинающиеся с /articles.

Intent позволяют передавать данные между компонентами. Существуют два основных способа передачи данных:

- Extras: используются для передачи небольших объемов данных, таких как примитивные типы (Int, String, Boolean и т.д.), а также объекты, реализующие интерфейсы Serializable или Parcelable.
- URI: используются для передачи ссылок на данные, хранящиеся в файловой системе или в другом приложении.

Для добавления данных в Intent используются методы putExtra(). Первый параметр – это ключ, по которому будет извлекаться значение, а второй – само значение.

Пример:

```
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("name", "John Doe")
intent.putExtra("age", 30)
startActivity(intent)
```

Для извлечения данных из Intent используются методы getStringExtra(), getIntExtra(), getBooleanExtra() и т.д.

Пример:

```
val name = intent.getStringExtra("name")
val age = intent.getIntExtra("age", 0) // Второй параметр - значение по умолчанию
```

Для передачи объектов через Intent, они должны быть сериализованы. В Android есть два основных способа сериализации:

- Serializable: простой маркерный интерфейс. Объекты, реализующие этот интерфейс, могут быть автоматически сериализованы и десериализованы.
- Parcelable: более эффективный интерфейс для сериализации объектов. Требует ручной реализации методов для записи и чтения данных из Parcel.

Важно помнить, что размер данных, передаваемых через Intent, ограничен. Превышение лимита (обычно около 1 МБ) может привести к ошибке TransactionTooLargeException. В этом случае рекомендуется использовать другие способы передачи данных, такие как сохранение данных в файл или базу данных и передача только ссылки на данные.

## 2.3. EditText

EditText – это UI-компонент в Android SDK, предназначенный для предоставления пользователю возможности ввода и редактирования текста. Он является потомком класса TextView, что означает, что он обладает всеми возможностями отображения текста, но при этом позволяет пользователю вносить изменения.

EditText можно создать и добавить в макет двумя способами:

1. Через XML-разметку. Для этого необходимо добавить элемент <EditText> в XML-файл макета экрана и настроить его атрибуты (например, id, layout_width, layout_height, hint, inputType). Затем, в коде Activity/Fragment, связаться с этим элементом с помощью findViewById().

Рассмотрим пример:

```
<EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/enter_text" />
```

Kotlin:

```
val editText: EditText = findViewById(R.id.editText)
```

2. Программно. Для этого создаем экземпляр EditText в коде Kotlin, настраиваем его свойства и добавляем в контейнер (например, LinearLayout, ConstraintLayout) с помощью программного добавления View.

Пример:

```
val editText = EditText(this)
    editText.hint = getString(R.string.enter_text)
    val layoutParams = LinearLayout.LayoutParams(
        LinearLayout.LayoutParams.MATCH_PARENT,
        LinearLayout.LayoutParams.WRAP_CONTENT
    )
    editText.layoutParams = layoutParams
```

```
        linearLayout.addView(editText)  // Добавление в
LinearLayout
```

Атрибут inputType позволяет указать тип данных, которые пользователь может вводить в EditText. Это влияет на вид виртуальной клавиатуры и валидацию ввода. Рассмотрим некоторые распространенные значения атрибута:

- text: обычный текст.
- textEmailAddress: текст, предназначенный для ввода адреса электронной почты.
- textPassword: текст, предназначенный для ввода пароля (скрытые символы).
- number: числовой ввод.
- numberDecimal: числовой ввод с возможностью ввода десятичной точки.
- phone: ввод телефонного номера.
- date: ввод даты.
- time: ввод времени.

EditText предоставляет множество методов для управления вводом текста:

- setOnKeyListener(listener: View.OnKeyListener): устанавливает обработчик нажатий клавиш виртуальной клавиатуры. Полезен для обработки нажатия клавиши Enter или других специальных клавиш.
- addTextChangedListener(watcher: TextWatcher): устанавливает обработчик изменений текста. Реагирует на каждое изменение текста в EditText.
- setSelection(start: Int, stop: Int): устанавливает выделение текста в диапазоне от start до stop.
- setSelectAllOnFocus(selectAllOnFocus: Boolean): автоматически выделяет весь текст в EditText, когда он получает фокус.

View в Android можно разделить на статические (отображение информации, без взаимодействия) и динамические (взаимодействие с пользователем: кнопки, поля ввода, списки).

Фокус позволяет пользователю взаимодействовать с определенным элементом на экране. Рассмотрим методы работы с фокусом:

- requestFocus(): метод класса View, доступный всем его наследникам. Позволяет программно установить фокус на элемент.
- setFocusable(focusable: Boolean): позволяет разрешить или запретить элементу получать фокус.

Для управления виртуальной клавиатурой используется специальный класс InputMethodManager, который служит посредником между источником ввода (клавиатурой) и приложением.

Например, код для скрытия клавиатуры:

```
val inputMethodManager = getSystemService(Context.INPUT_METHOD_SERVICE) as? InputMethodManager
inputMethodManager?.hideSoftInputFromWindow(currentView.windowToken, 0)
```

## 2.4. TextWatcher

TextWatcher – это интерфейс, который позволяет отслеживать изменения текста в EditText в режиме реального времени. Он определяет три метода обратного вызова:

- beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int): вызывается перед изменением текста. Предоставляет доступ к старому тексту, который будет заменен. Текст внутри этого метода нельзя изменить.
- onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int): вызывается во время изменения текста. Предоставляет доступ к новому тексту, который был добавлен. Текст внутри этого метода нельзя изменить.
- afterTextChanged(s: Editable?): вызывается после изменения текста. Предоставляет доступ к окончательному тексту, который можно изменить.

Рассмотрим пример использования TextWatcher

```
editText.addTextChangedListener(object : TextWatcher {
    override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
        // Обработка перед изменением текста
        Log.d("TextWatcher", "beforeTextChanged: $s")
    }
    override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
        // Обработка во время изменения текста
        Log.d("TextWatcher", "onTextChanged: $s")
    }
```

```
        override fun afterTextChanged(s: Editable?) {
            // Обработка после изменения текста
            Log.d("TextWatcher", "afterTextChanged: $s")
            // Пример: валидация ввода
            if (s.toString().length > 10) {
                editText.error = "Превышена максимальная
длина"
            } else {
                editText.error = null
            }
        }
    })
```

В примере выше добавляется слушатель изменений текста к EditText. Слушатель реагирует на каждое изменение текста, выводя отладочные сообщения в лог и выполняя валидацию ввода после каждого изменения. Если длина текста превышает 10 символов, отображается сообщение об ошибке.





# ГЛАВА 3. РАБОТА СО СПИСКАМИ

## 3.1 Списки в Android: Spinner, GridView, ListView и RecyclerView

Список – это упорядоченный набор элементов данных, в котором один и тот же элемент может встречаться несколько раз. В Android списки играют важную роль в представлении информации пользователю. Android предоставляет несколько инструментов для отображения списков: Spinner, GridView, ListView и RecyclerView.

Spinner – это UI-компонент, который представляет собой выпадающий список, позволяющий пользователю выбрать один элемент из небольшого набора предопределенных значений.

Пример добавления Spinner в XML-разметку:

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Рассмотрим два способа задания элементов Spinner.

Первый способ использует атрибут android:entries: можно напрямую указать список элементов в XML-разметке, разделив их символом |. Этот способ подходит для небольших и статических списков.

Второй способ заключается в использовании адаптера. Это более гибкий способ, который позволяет динамически задавать элементы списка и настраивать их отображение.

Adapter – это класс, который служит посредником между источником данных (например, списком строк или объектов) и UI-компонентом, отображающим эти данные (например, Spinner, GridView, ListView). Адаптер преобразует данные из источника в формат, понятный UI-компоненту.

Пример использования ArrayAdapter для Spinner:

```
val spinner = findViewById<Spinner>(R.id.spinner)
val names = listOf("Агафья", "Екатерина", "Руслана", "Татьяна", "Ульяна")
val adapter: SpinnerAdapter = ArrayAdapter(
    this, // Context
    android.R.layout.simple_spinner_dropdown_item,  // Layout для одного элемента списка
    names // Список данных
)
```

```
spinner.adapter = adapter
```

В первой строке находим Spinner в макете по его ID, затем создаем список строк, который будет использоваться в качестве данных для Spinner. На третьей строке создаем экземпляр ArrayAdapter, который является стандартным адаптером для отображения списка строк. В данном случае this - передача контекста (обычно Activity или Fragment), android.R.layout.simple_spinner_dropdown_item - указание стандартного layout для отображения каждого элемента выпадающего списка. Names - передача списка строк, который будет отображаться в Spinner. На последней строке кода устанавливаем созданный адаптер для Spinner, чтобы он мог отображать данные.

GridView – это UI-компонент, который отображает элементы в виде таблицы (сетки). Он также использует Adapter для связывания данных и представления. Класс адаптера должен реализовать интерфейс ListAdapter.

Пример добавления GridView в XML-разметку:

```
<GridView
    android:id="@+id/gridView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:numColumns="4" >
</GridView>
```

Обратите внимание, что android:numColumns указывает количество столбцов в таблице.

Рассмотрим пример использования ArrayAdapter для GridView:

```
val gridView = findViewById<GridView>(R.id.gridView)
val names = listOf("Агафья", "Екатерина", "Руслана", "Татьяна", "Ульяна")
val girlNames = List(100) { names[Random.nextInt(0, names.size)] }
val adapter: ListAdapter = ArrayAdapter(
    this,
    android.R.layout.simple_list_item_1, // Layout для одного элемента списка
    girlNames // Список данных
)
gridView.adapter = adapter
```

Код аналогичен примеру выше.

ListView – это UI-компонент, который отображает элементы в виде вертикального прокручиваемого списка. В современных приложениях

рекомендуется использовать RecyclerView вместо ListView, так как RecyclerView более гибкий и эффективный.

Пример добавления ListView в XML-разметку:

```
<ListView
    android:id="@+id/listView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >
</ListView>
```

Пример использования ArrayAdapter для ListView:

```
val listView = findViewById<ListView>(R.id.listView)
val names = listOf("Агафья", "Екатерина", "Руслана", "Татьяна", "Ульяна")
val girlNames = List(100) { names[Random.nextInt(0, names.size)] }
val adapter: ListAdapter = ArrayAdapter(
    this,
    android.R.layout.simple_list_item_1, // Layout для одного элемента списка
    girlNames // Список данных
)
listView.adapter = adapter
```

Код аналогичен примеру для GridView, только здесь используется ListView вместо GridView.

RecyclerView – это UI-компонент, который предоставляет гибкий и эффективный способ отображения списков и таблиц данных. Он является более продвинутым и рекомендуемым вариантом по сравнению с ListView. RecyclerView позволяет переиспользовать View для отображения элементов списка, что повышает производительность при работе с большими списками.

Для работы с RecyclerView необходимо:

- Создать класс ViewHolder. ViewHolder хранит ссылки на UI-элементы одного элемента списка.
- Создать класс Adapter. Adapter отвечает за создание ViewHolder и связывание данных с UI-элементами.
- Задать LayoutManager. LayoutManager определяет, как элементы будут располагаться на экране (линейно, в виде сетки и т.д.).

RecyclerView – это мощный инструмент для отображения списков данных в Android. Для эффективной работы с RecyclerView необходимо понимать концепции ViewHolder и LayoutManager.

## 3.2. Основы работы с RecyclerView

ViewHolder – это паттерн проектирования, используемый для оптимизации производительности RecyclerView. Он позволяет избежать повторного поиска View-элементов в каждом элементе списка, что особенно важно при работе с большими списками.

Без использования ViewHolder каждый раз, когда RecyclerView должен отобразить новый элемент списка, он должен найти все View-элементы (TextView, ImageView и т.д.) внутри этого элемента с помощью findViewById(). Это ресурсоемкая операция.

С использованием ViewHolder ситуация упрощается. ViewHolder хранит ссылки на View-элементы одного элемента списка. Когда RecyclerView отображает новый элемент, он сначала пытается найти существующий ViewHolder для переиспользования. Если ViewHolder найден, он просто обновляет значения View-элементов, используя уже существующие ссылки, без повторного вызова findViewById().

Для создания своей реализации ViewHolder необходимо создать класс, который наследуется от абстрактного класса RecyclerView.ViewHolder.

Рассмотрим пример:

```
    class MyViewHolder(itemView: View) : 
RecyclerView.ViewHolder(itemView) {
        val titleTextView: TextView = 
itemView.findViewById(R.id.titleTextView)
        val descriptionTextView: TextView = 
itemView.findViewById(R.id.descriptionTextView)
        fun bind(item: MyItem) {
            titleTextView.text = item.title
            descriptionTextView.text = item.description
        }
    }
```

В примере создаем класс MyViewHolder, который наследуется от RecyclerView.ViewHolder. Затем находим TextView для отображения заголовка и TextView для отображения описания. fun bind(item: MyItem) — это метод для связывания данных (объекта MyItem) с UI-элементами.

Adapter (в контексте RecyclerView) – это класс, который наследуется от базового абстрактного класса RecyclerView.Adapter. Adapter связывает набор данных с их визуальным представлением и отслеживает изменения набора данных. Он отвечает за создание ViewHolder и связывание данных с UI-элементами.

Абстрактные методы RecyclerView.Adapter:

- onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder создает новый ViewHolder.
- onBindViewHolder(holder: ViewHolder, position: Int) устанавливает значения UI-компонентов ViewHolder из модели данных.
- getItemCount(): Int возвращает количество элементов в списке.

Рассмотрим пример реализации Adapter:

```
class MyAdapter(private val items: List<MyItem>) : RecyclerView.Adapter<MyViewHolder>() {
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
            val view = LayoutInflater.from(parent.context).inflate(R.layout.my_item_layout, parent, false)
            return MyViewHolder(view)
        }
        override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
            holder.bind(items[position])
        }
        override fun getItemCount() = items.size
    }
```

Сначала создаем класс MyAdapter, который наследуется от RecyclerView.Adapter. private val items: List<MyItem> — это список данных, который будет отображаться в RecyclerView. Затем переопределяем метод onCreateViewHolder и раздуваем (inflate) layout my_item_layout для одного элемента списка. С помощью return MyViewHolder(view) создаем и возвращаем новый MyViewHolder с раздутым представлением. override fun onBindViewHolder(holder: MyViewHolder, position: Int) - переопределяем метод onBindViewHolder. position: Int — это позиция элемента в списке. Затем вызываем метод bind ViewHolder, чтобы связать данные с UI-элементами.

override fun getItemCount() = items.size - переопределяем метод getItemCount, который будет возвращать количество элементов в списке.

LayoutManager управляет расположением элементов внутри RecyclerView: вертикально, горизонтально, в виде таблицы.

Основные реализации LayoutManager:

- LinearLayoutManager располагает элементы друг за другом в виде списка. Позволяет задать ориентацию списка: вертикальную или горизонтальную.
- GridLayoutManager предназначен для вывода элементов в виде таблицы.

Пример использования LinearLayoutManager:

```
recyclerView.layoutManager = LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false)
```

В примере устанавливается LinearLayoutManager для RecyclerView. Тип ориентации списка выбран вертикальный, параметр false указывает, нужно ли выводить список в обратном направлении.

Рассмотрим пример использования GridLayoutManager:

```
recyclerView.layoutManager = GridLayoutManager(this, 3, GridLayoutManager.VERTICAL, false)
```

В примере устанавливается GridLayoutManager для RecyclerView. Количество столбцов в таблице равно 3. Выбран вертикальный тип ориентации списка и false указывает, нужно ли выводить список в обратном направлении.

RecyclerView использует два списка ViewHolder для оптимизации:

- View Cache — это список, в котором находятся ViewHolder с уже связанными (bind) View-элементами. Чаще всего это элементы, которые находятся на экране. RecyclerView запрашивает элементы из этого списка по конкретной позиции.
- Recycled View Pool - в этом списке хранятся ViewHolder, которые RecyclerView получает по ViewType — части метода у RecyclerView.Adapter.

Алгоритм отображения элемента RecyclerView:

1. Поиск во View Cache. RecyclerView ищет ViewHolder для нужной позиции во View Cache. Если ViewHolder найден, то RecyclerView использует его для отображения элемента, минуя шаги создания и связывания.

2. Поиск в Recycled View Pool. Если ViewHolder не найден во View Cache, то RecyclerView ищет его в Recycled View Pool по ViewType. Если ViewHolder

найден, то RecyclerView использует его для отображения элемента, минуя шаг создания.

3. Создание нового ViewHolder. Если ViewHolder не найден ни во View Cache, ни в Recycled View Pool, то RecyclerView создает новый ViewHolder с помощью onCreateViewHolder() и связывает его с данными с помощью onBindViewHolder().

Преимущества использования ViewHolder:

- улучшение производительности за счет переиспользования View-элементов;
- снижение нагрузки на процессор и память;
- более плавный скроллинг списков.

Адаптер в RecyclerView выступает в роли посредника между вашим набором данных (списком объектов) и визуальным представлением этих данных на экране.

RecyclerView.Adapter – это абстрактный класс, который вы расширяете, чтобы предоставить данные для RecyclerView и определить, как эти данные должны быть отображены.

Основные обязанности адаптера:

- создание ViewHolder для каждого элемента списка;
- связывание данных с UI-элементами в ViewHolder;
- отслеживание изменений в наборе данных;
- определение количества элементов в списке.

При создании собственного адаптера необходимо переопределить три ключевых метода: onCreateViewHolder, onBindViewHolder и getItemCount. Эти методы управляют жизненным циклом ViewHolder.

Рассмотрим первый метод

```
onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder
```

Этот метод отвечает за создание новых экземпляров ViewHolder. Он вызывается RecyclerView, когда необходимо создать новый ViewHolder для отображения элемента списка. Внутри этого метода необходимо раздуть (inflate) layout для одного элемента списка и создать экземпляр ViewHolder, передав ему раздутый View. parent: ViewGroup — это ViewGroup, в который будет добавлен View-элемент, а viewType: Int — это тип View-элемента. Может использоваться для отображения разных типов элементов в одном списке.

Рассмотрим пример:

```
    override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): MyViewHolder {
        val view =
LayoutInflater.from(parent.context).inflate(R.layout.my_i
tem_layout, parent, false)
        return MyViewHolder(view)
    }
```

Сначала раздуваем layout my_item_layout для одного элемента списка, а затем создаем и возвращаем новый MyViewHolder с раздутым представлением.

Рассмотрим второй метод:

```
onBindViewHolder(holder: ViewHolder, position: Int)
```

Этот метод отвечает за связывание данных с UI-элементами внутри ViewHolder. Он вызывается RecyclerView, когда необходимо отобразить элемент списка на определенной позиции. Внутри этого метода необходимо получить данные для элемента на заданной позиции из вашего набора данных и установить их в соответствующие UI-элементы ViewHolder. holder: ViewHolder - это ViewHolder, с которым нужно связать данные, а position: Int — это позиция элемента в списке.

Рассмотрим пример:

```
    override fun onBindViewHolder(holder: MyViewHolder,
position: Int) {
        val item = items[position]
        holder.bind(item)
    }
```

В примере мы получаем элемент данных на заданной позиции из списка items. а затем вызываем метод bind ViewHolder (который мы определили ранее), чтобы связать данные с UI-элементами.

Рассмотрим пример связывания данных с UI-элементами:

```
    class MyViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    private val titleTextView: TextView =
itemView.findViewById(R.id.titleTextView)
    private val descriptionTextView: TextView =
itemView.findViewById(R.id.descriptionTextView)
        fun bind(item: MyItem) {
            titleTextView.text = item.title
            descriptionTextView.text =
item.description
```





```
                // Пример условного форматирования
                if (item.isImportant) {
                    titleTextView.setTextColor(Color.RED)
                } else {

titleTextView.setTextColor(Color.BLACK)
                }
            }
        }
```

Сначала устанавливаем заголовок из объекта MyItem в TextView. Затем с помощью descriptionTextView.text = item.description устанавливаем описание из объекта MyItem в TextView. Конструкция if (item.isImportant) { ... } else { ... } — это пример условного форматирования: если элемент важный, то устанавливаем красный цвет текста для заголовка, иначе - черный.

RecyclerView переиспользует существующие View, создавая только то количество View, которое помещается на экране. Поэтому важно правильно переиспользовать ViewHolder и не выполнять ресурсоемкие операции в onBindViewHolder (например, загрузку изображений).

Перейдем к третьему методу:

```
getItemCount(): Int:
```

Этот метод должен возвращать количество элементов в наборе данных. RecyclerView использует это значение для определения количества элементов, которые необходимо отобразить.

Класс RecyclerView.Adapter является generic-типом, что позволяет указать конкретный тип ViewHolder, который будет использоваться в адаптере. Это улучшает типобезопасность и упрощает работу с кодом.

Пример:

```
class MyAdapter(private val items: List<MyItem>) : RecyclerView.Adapter<MyViewHolder>() {
    // ...
}
```

В этом примере мы указываем, что MyAdapter будет использовать MyViewHolder. Это означает, что возвращаемый тип метода onCreateViewHolder должен быть MyViewHolder, а тип параметра holder в методе onBindViewHolder будет MyViewHolder.

Адаптер в RecyclerView не только предоставляет данные для отображения, но и отвечает за обработку изменений в этих данных. В этом разделе мы рассмотрим несколько распространенных задач, которые выполняются с

помощью адаптера: обновление списка, добавление элемента, обработка нажатий на элемент и удаление элемента.

1. Полное обновление списка (notifyDataSetChanged())

Представьте, что пользователь ввел новый поисковый запрос и получил новый список результатов. Или пользователь изменил настройки сортировки, и порядок элементов в списке изменился. В таких случаях необходимо обновить отображаемый список. Самый простой способ – вызвать метод notifyDataSetChanged() у адаптера:

```
    reverseButton.setOnClickListener {
        items.reverse() // Изменяем порядок элементов в списке
        adapter.notifyDataSetChanged() // Сообщаем адаптеру об изменении
    }
```

Сначала с помощью items.reverse() инвертируем порядок элементов в списке items. Затем используя adapter.notifyDataSetChanged() уведомляем адаптер о том, что набор данных полностью изменился. Это приведет к полной перерисовке всего списка.

notifyDataSetChanged() перерисовывает все элементы списка. Это может быть неэффективно, особенно для больших списков, так как требует больших вычислительных ресурсов. Используйте его только тогда, когда изменения затрагивают весь список (например, полная замена данных или изменение порядка).

2. Добавление элемента (notifyItemInserted() и notifyItemRangeChanged()).

Если нужно добавить новый элемент в список, например, после ввода текста в поле ввода, можно использовать методы notifyItemInserted() и notifyItemRangeChanged() для более эффективного обновления.

Рассмотрим пример:

```
    addButton.setOnClickListener {
        if (textInput.text.isNotEmpty()) {
            val newItem = Item(R.drawable.primer, textInput.text.toString())
            items.add(0, newItem) // Добавляем элемент в начало списка
            textInput.text.clear() // Очищаем поле ввода
            adapter.notifyItemInserted(0) // Сообщаем адаптеру о вставке элемента
```

```
            adapter.notifyItemRangeChanged(0,  items.size)
// Сообщаем об изменении индексов
        }
    }
```

С помощью val newItem = Item(R.drawable.primer, textInput.text.toString()) создаем новый экземпляр класса Item с картинкой и текстом из поля ввода. items.add(0, newItem) добавляет новый элемент в начало списка items. textInput.text.clear() очищает поле ввода. adapter.notifyItemInserted(0) уведомляет адаптер о том, что в список был вставлен новый элемент по индексу 0. Это приводит к анимации вставки и смещению остальных элементов вниз. adapter.notifyItemRangeChanged(0, items.size) уведомляет адаптер о том, что все элементы, начиная с индекса 0, изменили свои индексы, поскольку был вставлен новый элемент.

Преимущества использования notifyItemInserted() и notifyItemRangeChanged():

- более эффективное обновление списка, чем notifyDataSetChanged();
- плавная анимация вставки элемента;
- избежание перерисовки всех элементов списка.

3. Обработка нажатий на элемент списка.

Для обработки нажатий на элемент списка нужно использовать OnClickListener. Устанавливать его лучше всего в методе onBindViewHolder() адаптера, так как именно здесь происходит связывание данных с View-элементом:

```
    override fun onBindViewHolder(holder: MyViewHolder,
position: Int) {
        val item = items[position]
        holder.bind(item)
        holder.itemView.setOnClickListener {
            // Обработка нажатия на элемент
            onItemClick(item) // Вызываем функцию обработки
нажатия
        }
    }
```

Сначала устанавливаем OnClickListener на корневой View-элемент (itemView) ViewHolder с помощью holder.itemView.setOnClickListener { ... }. Затем используем onItemClick(item) для вызова функции onItemClick, в которой будет реализована логика обработки нажатия.





Пример функции onItemClick() (в Activity/Fragment):

```
private fun onItemClick(item: Item) {
    // Обработка нажатия на элемент item
    Toast.makeText(this, "Нажата на элемент: ${item.text}", Toast.LENGTH_SHORT).show()
```

4. Удаление элементов списка (notifyItemRemoved() и notifyItemRangeChanged()):

Для удаления элемента из списка необходимо использовать методы notifyItemRemoved() и notifyItemRangeChanged():

```
fun removeItem(item: Item) {
    val position = items.indexOf(item) // Определяем позицию элемента в списке
    if (position != -1) {
        items.removeAt(position) // Удаляем элемент из списка
        notifyItemRemoved(position) // Уведомляем адаптер об удалении
        notifyItemRangeChanged(position, items.size) // Уведомляем об изменении индексов
    }
}
```

Рассмотрим код выше. val position = items.indexOf(item) - определяем индекс удаляемого элемента в списке items. if (position != -1) { ... } - проверяем, найден ли элемент в списке (indexOf вернет -1, если элемент не найден). items.removeAt(position) удаляем элемент из списка items по индексу. notifyItemRemoved(position) уведомляем адаптер о том, что элемент был удален с позиции position. Это приводит к анимации удаления. notifyItemRangeChanged(position, items.size) - ведомляем адаптер о том, что индексы всех элементов, начиная с позиции position, изменились.

Важно:

- всегда удаляйте элемент из списка перед вызовом notifyItemRemoved();
- после удаления элемента необходимо вызывать notifyItemRangeChanged(), чтобы RecyclerView корректно обновил индексы элементов.

Рассмотрим пример обработка нажатий и удаления элементов вместе:

```
override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
```

```
        val item = items[position]
        holder.bind(item)
        holder.itemView.setOnClickListener {
            // Передаем элемент для удаления во внешнюю
функцию
            removeItem(item)
        }
    }
    // В Activity/Fragment
    fun removeItem(item: Item) {
        val position = items.indexOf(item)
        if (position != -1) {
            items.removeAt(position)
            adapter.notifyItemRemoved(position)
            adapter.notifyItemRangeChanged(position,
items.size)
        }
    }
```

Обсудим более подробно код:

- override fun onBindViewHolder(holder: MyViewHolder, position: Int) { ... } - это переопределенный метод из класса RecyclerView.Adapter. Он вызывается RecyclerView, когда нужно связать данные с UI-элементами в ViewHolder для отображения элемента на определенной позиции.
- holder: MyViewHolder — это экземпляр нашего класса MyViewHolder (который мы создали ранее), содержащий ссылки на View-элементы одного элемента списка.
- position: Int — это индекс элемента в списке данных, который нужно отобразить.
- val item = items[position] - здесь мы получаем объект данных (item) из списка items на позиции position. Этот объект содержит данные, которые нужно отобразить в View-элементах.
- holder.bind(item) - здесь мы вызываем метод bind() нашего MyViewHolder. Этот метод отвечает за установку значений из объекта item в соответствующие View-элементы ViewHolder (например, установить текст в TextView, загрузить изображение в ImageView и т.д.). Мы подробно рассматривали этот метод ранее.





- holder.itemView.setOnClickListener { ... } Здесь holder.itemView — это корневой View-элемент (самый верхний элемент в layout) одного элемента списка. SetOnClickListener { ... } - здесь мы устанавливаем OnClickListener на этот корневой View-элемент. Это означает, что, когда пользователь нажмет на любой View-элемент внутри этого корневого элемента (то есть на любую часть элемента списка), будет вызван код, указанный в лямбда-выражении { ... }.
- removeItem(item) - внутри OnClickListener мы вызываем функцию removeItem(item), передавая ей объект данных item, который соответствует элементу списка, на который нажали.
- fun removeItem(item: Item) { ... } (в Activity/Fragment) - эта функция находится в вашей Activity или Fragment, где расположен RecyclerView. Она отвечает за удаление элемента из списка и уведомление адаптера об этом удалении.
- val position = items.indexOf(item) - здесь мы находим индекс объекта item в списке items. items.indexOf(item) возвращает индекс первого вхождения объекта item в список items. Если объект не найден, метод возвращает -1.
- if (position != -1) { ... } — это проверка, чтобы убедиться, что объект item действительно существует в списке items. Если position равно -1 (то есть объект не найден), код внутри if не будет выполнен.
- items.removeAt(position) - здесь мы удаляем объект из списка items по индексу position. items.removeAt(position) удаляет элемент с указанным индексом и сдвигает все последующие элементы влево.
- adapter.notifyItemRemoved(position) - это очень важный шаг! Здесь мы уведомляем адаптер (adapter) о том, что элемент был удален с позиции position. Это приводит к тому, что RecyclerView выполняет анимацию удаления элемента (если она поддерживается) и перерисовывает список.
- adapter.notifyItemRangeChanged(position, items.size) - это еще один важный шаг! Здесь мы уведомляем адаптер о том, что все элементы, начиная с позиции position, изменили свои индексы, так как элемент на позиции position был удален, и все элементы после него сдвинулись влево. Это нужно для того, чтобы RecyclerView правильно отображал оставшиеся элементы списка.

В целом, этот код делает следующее:

1. Когда пользователь нажимает на элемент списка, определяется, какой объект данных соответствует этому элементу.

2. Вызывается функция removeItem(), которая:

- находит индекс этого объекта в списке данных;
- удаляет объект из списка данных;
- уведомляет адаптер об удалении элемента, чтобы RecyclerView мог обновить отображение списка.

Почему это важно?

Согласованность данных и отображения: важно, чтобы список данных (items) и отображаемый список в RecyclerView всегда были синхронизированы.

Производительность: использование notifyItemRemoved() и notifyItemRangeChanged() более эффективно, чем полное обновление списка (notifyDataSetChanged()), так как позволяет RecyclerView обновить только те элементы, которые действительно изменились.

Анимация: notifyItemRemoved() позволяет выполнить анимацию удаления элемента, что делает пользовательский интерфейс более приятным.





# ГЛАВА 4. ОСНОВЫ СЕТЕВОГО ВЗАИМОДЕЙСТВИЯ

## 4.1 Клиент-серверная архитектура в Android-приложениях

При разработке Android-приложений можно выделить два основных типа: standalone и клиент-серверные. Каждый подход имеет свои преимущества и недостатки, и выбор зависит от конкретных задач приложения.

Standalone (англ. "стоящий в одиночестве") приложение способно функционировать независимо, без постоянного подключения к сети Интернет. Все необходимые данные и ресурсы содержатся внутри приложения с момента его установки. При этом не требуется подключения к сети для выполнения основных функций; данные хранятся локально (например, в базе данных SQLite или в файлах). Такой вариант подходит для приложений, выполняющих простые операции или работающих с относительно небольшими объемами данных. Например, калькулятор, диктофон, камера, контакты (в большинстве случаев), офлайн-карты, фоторедактор.

Преимуществами таких приложений являются: возможность работы в условиях отсутствия интернет-соединения; быстрая загрузка и отзывчивость, так как данные доступны локально, а недостатками: ограниченное количество данных, которые можно хранить локально; сложность обновления данных, требуется выпуск новой версии приложения; сложность реализации сложных вычислений, требующих больших вычислительных ресурсов; потенциальные проблемы с безопасностью при хранении конфиденциальных данных локально; увеличенный размер приложения из-за хранения всех необходимых ресурсов.

Второй тип — это клиент-серверные приложения, т. е. приложения, состоящие из двух частей: клиентской (Android-приложение) и серверной (отдельное приложение, расположенное на сервере).

Клиент отвечает за взаимодействие с пользователем, отображение данных и отправку запросов на сервер. Сервер обрабатывает запросы от клиента, выполняет сложные вычисления, хранит и управляет данными, и отправляет ответы клиенту. Взаимодействие между клиентом и сервером осуществляется по определенным правилам (протоколам), таким как HTTP/HTTPS.

Таким образом, можно выделить основные характеристики клиент-серверных приложений:

- требуют подключения к сети для работы;
- данные хранятся на сервере;

- подходит для приложений, работающих с большими объемами данных, требующих сложных вычислений и постоянного обновления данных.

Большинство современных приложений являются клиент-серверными, например, социальные сети, онлайн-игры, приложения для электронной коммерции, приложения для заказа такси, онлайн-кинотеатры.

Преимуществами клиент-серверных приложений являются:

- централизованное хранение и управление данными;
- простота обновления данных;
- возможность реализации сложных вычислений на сервере;
- улучшенная безопасность данных;
- снижение размера приложения на устройстве пользователя;
- актуальность данных.

Среди недостатков клиент-серверных приложений можно выбелить:

- требует подключения к сети;
- зависимость от доступности и производительности сервера;
- сложность разработки и поддержки (требуется разработка как клиентской, так и серверной части);
- повышенные требования к безопасности (необходимо защищать как клиент, так и сервер).

Таким образом, клиент-серверный подход является доминирующим в современной разработке мобильных приложений. Он позволяет создавать сложные, функциональные и безопасные приложения, работающие с большими объемами данных.

В клиент-серверной архитектуре сервер играет ключевую роль, обеспечивая хранение, обработку и предоставление данных клиентам. Рассмотрим подробнее компоненты и принципы работы серверной части.

Сервер — это специализированное компьютерное оборудование или программное обеспечение, предназначенное для выполнения сервисных функций и ответа на запросы от клиентов в сети.

Аппаратный сервер — это физический компьютер, обладающий высокой производительностью, надежностью и отказоустойчивостью. Часто размещается в дата-центрах.

Программный сервер (серверное приложение) — это программа, выполняющаяся на аппаратном сервере и реализующая логику обработки запросов от клиентов.

Рассмотрим подробнее процесс взаимодействия клиента и сервера (рис. 2):

1. Ожидание запросов. Серверное приложение постоянно находится в состоянии ожидания запросов от клиентов.

2. Получение запроса. Когда клиент отправляет запрос, сервер получает его.

3. Обработка запроса. Сервер анализирует запрос, определяет его тип и необходимые параметры.

4. Выполнение запроса. Сервер выполняет необходимые действия, такие как:

- Получение данных из базы данных.
- Выполнение сложных вычислений.
- Изменение данных в базе данных.

5. Формирование ответа. Сервер формирует ответ, содержащий запрошенные данные или информацию об успехе/ошибке выполнения запроса.

6. Отправка ответа. Сервер отправляет ответ клиенту.

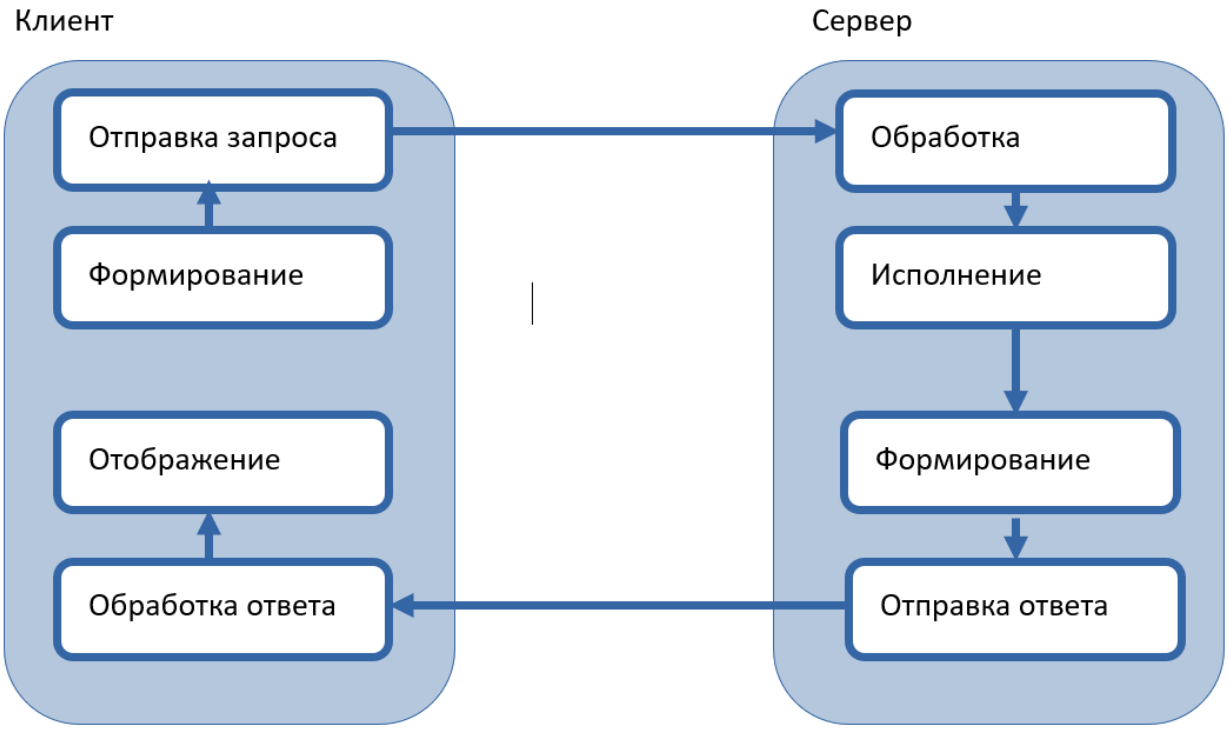


*Рисунок 2. Процесс взаимодействия клиента и сервера*

Клиент-серверные приложения взаимодействуют с базами данных, которые обеспечивают хранение, извлечение, изменение и удаление данных, необходимых для работы серверного приложения. База данных (БД) — это организованный набор структурированных данных, хранимых на компьютере. Свойство БД обеспечивать сохранение данных в течение долгого времени

называется персистентностью. Используя базу данных, мы защищаемся от потери данных в случае сбоя сервера.

Рассмотрим схему работы клиент-серверного приложения с использованием базы данных и балансировщиков (рис. 3) Сервер обрабатывает запросы от множества клиентов (Android, iOS, веб и др.). Для обеспечения отказоустойчивости и масштабируемости используются несколько копий серверов, объединенных в кластер. Балансировщик распределяет нагрузку между серверами, обеспечивая оптимальную производительность. Для распределения нагрузки на БД используются реплики БД и балансировщики.

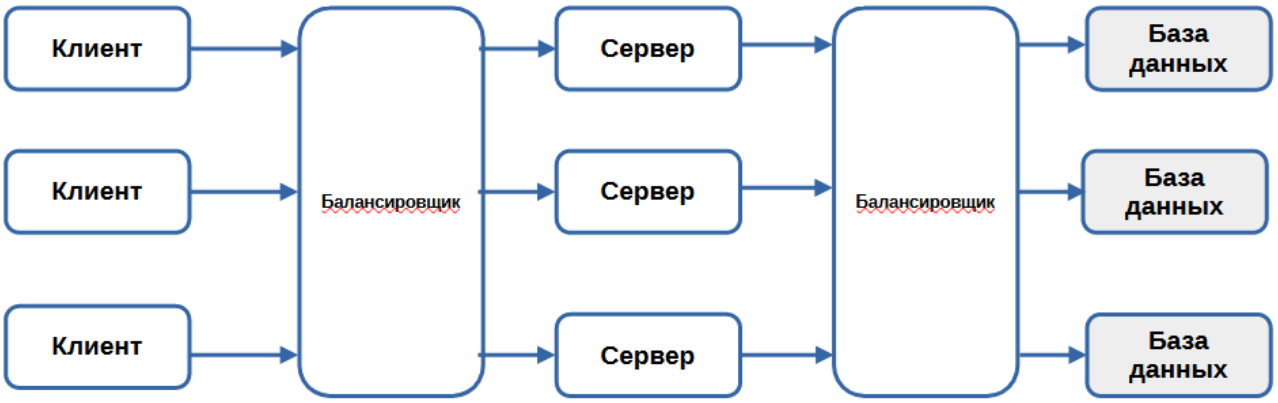


*Рисунок 3. Схема работы клиент-серверного приложения*

## 4.2. REST API: взаимодействие между клиентом и сервером

REST (Representational State Transfer) — это архитектурный стиль для создания сетевых приложений, который определяет набор принципов и ограничений для взаимодействия между клиентом и сервером.

API — это набор правил и спецификаций, которые определяют, как различные программные компоненты могут взаимодействовать друг с другом.

REST API (Application Programming Interface) — это интерфейс, который соответствует принципам REST и используется для обмена информацией между клиентом и сервером.

REST опирается на несколько ключевых принципов, которые помогают создавать масштабируемые, гибкие и удобные в использовании API.

Основные принципы REST:

- модель "клиент-сервер". Разделение ответственности между клиентом и сервером. Клиент отвечает за пользовательский интерфейс, а сервер - за хранение и обработку данных.

- отсутствие состояния (Stateless). Сервер не хранит никакой информации о клиенте между запросами. Каждый запрос от клиента должен содержать всю необходимую информацию для выполнения.
- кэширование (Cacheable). Клиенты и серверы могут кэшировать ответы для повышения производительности.
- единообразие интерфейса (Uniform Interface). Определение стандартного набора операций для доступа к ресурсам.
- слои (Layered System). Клиент может взаимодействовать с сервером через промежуточные слои (например, прокси-серверы, балансировщики нагрузки).
- код по требованию (Code on Demand) — опционально. Сервер может предоставлять клиенту исполняемый код для расширения функциональности.

REST API определяет, как клиент может отправлять запросы на сервер и как сервер должен отвечать на эти запросы.

Понимание клиент-серверной архитектуры и принципов REST API является важным для разработки современных Android-приложений. Этот подход позволяет создавать масштабируемые, гибкие и удобные в использовании приложения, работающие с большими объемами данных и требующие постоянного обновления.

## 4.3. TCP/IP: основа для передачи данных в Android-приложениях

В современном мире Android-приложения активно взаимодействуют с сетью для получения данных, обмена информацией и предоставления разнообразных онлайн-сервисов. Однако, передача данных между устройствами через сеть – это сложный процесс, требующий согласованной работы множества компонентов. Как и в любой сложной задаче, для упрощения понимания и реализации необходимо использовать принцип декомпозиции, то есть разделения большой задачи на несколько более мелких и управляемых. Именно этот принцип был положен в основу модели TCP/IP, название которой само по себе намекает на разделение.

TCP/IP – это не просто аббревиатура, а целая сетевая модель, которая описывает стандартизированный способ передачи данных от источника информации к получателю по сети. Эта модель предполагает, что информация проходит через несколько уровней, каждый из которых имеет свои собственные правила и обязанности. Эти правила называются протоколами.

Протокол, или сетевой протокол – это формальный набор правил, который определяет, как данные передаются между участниками коммуникации в сети. Протоколы можно разделить на два основных типа:

Высокоуровневые протоколы. Ориентированы на взаимодействие между приложениями и пользователями. Они определяют формат данных, команды и правила обмена информацией между программными компонентами. Именно с высокоуровневыми протоколами чаще всего работают Android-разработчики.

Низкоуровневые протоколы. Обеспечивают взаимодействие на аппаратном уровне. Они определяют, как физически передаются сигналы по сети (например, по сетевому кабелю), как кодируются данные и как обнаруживаются ошибки.

В рамках данного курса мы будем в основном рассматривать высокоуровневые протоколы, поэтому, когда мы говорим "протокол", мы подразумеваем именно высокоуровневый протокол. Одним из наиболее важных протоколов, с которым мы познакомимся, является HTTP (Hypertext Transfer Protocol), который играет ключевую роль во взаимодействии между клиентом (Android-приложением) и сервером.

Протоколы обмениваются специальными сообщениями, которые называются пакетами. Пакет – это структурированная единица данных, которая содержит в себе:

- передаваемые данные (полезная нагрузка) — это непосредственно информация или сообщение, которое мы хотим доставить получателю. Это может быть текст, изображение, аудио, видео или любые другие данные.
- служебная информация (заголовок) — это дополнительные сведения, необходимые для работы протокола. Эта информация включает в себя адреса отправителя и получателя, информацию о типе данных, контрольные суммы для проверки целостности данных, время отправки и другие параметры, необходимые для правильной обработки пакета.

Как мы уже упоминали, TCP/IP – это не просто набор протоколов, а целая модель, состоящая из четырех уровней, каждый из которых выполняет определенные функции в процессе передачи данных (рис. 4):

- Канальный уровень (Link Layer). Отвечает за физическую передачу данных между двумя непосредственно соединенными устройствами.
- Сетевой уровень (Network Layer). Отвечает за маршрутизацию данных между различными сетями.

- Транспортный уровень (Transport Layer). Обеспечивает надежную и упорядоченную доставку данных между приложениями.
- Прикладной уровень (Application Layer). Предоставляет интерфейс для приложений, позволяющий им обмениваться данными по сети.

Важно отметить, что канальный, сетевой и транспортный уровни относятся к низкоуровневым протоколам, в то время как прикладной уровень – к высокоуровневым. Каждый уровень работает независимо от других, выполняя свои специфические задачи. Такая модульность позволяет легко заменять или обновлять протоколы на каждом уровне, не затрагивая работу других уровней.

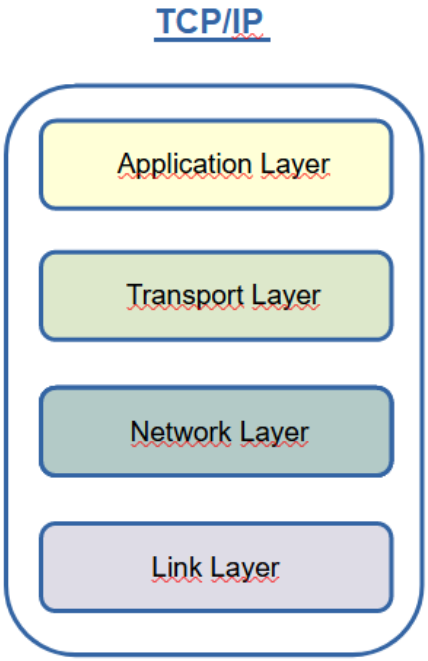


*Рисунок 4. Уровни протокола TCP/IP*

Рассмотрим каждый из этих уровней более подробно, начиная с самого нижнего уровня.

Канальный уровень – это самый нижний уровень модели TCP/IP. Он отвечает за физическую передачу данных между двумя непосредственно соединенными устройствами, например, между компьютером и маршрутизатором. Основные задачи:

- кодирование данных в биты для физической передачи;
- установление начала и конца передаваемого пакета (кадра);
- обнаружение и исправление ошибок при передаче данных;
- обеспечение помехоустойчивости.

Наиболее распространенными реализациями являются:

- Ethernet (IEEE 802.3) - стандарт для проводной коммуникации между устройствами. Ethernet определяет, как электрические сигналы передаются по кабелю и как формируются пакеты данных (кадры).
- Wi-Fi (IEEE 802.11) - стандарт для беспроводной коммуникации между устройствами. Wi-Fi использует радиоволны для передачи данных между устройствами в беспроводной сети.

Сетевой уровень отвечает за передачу данных между различными сетями. Интернет – это глобальная сеть, объединяющая миллионы локальных сетей по всему миру. Чтобы данные могли быть доставлены от одного устройства к другому, необходимо определить маршрут, по которому эти данные должны пройти.

IP отвечает за маршрутизацию данных между сетями. Он присваивает каждому устройству в сети уникальный IP-адрес и использует эти адреса для определения оптимального пути для доставки данных.

IP-адрес состоит из четырех чисел (от 0 до 255), разделенных точками (например, 192.168.1.1).

Маршрутизаторы (роутеры) — это специальные устройства, которые отвечают за пересылку данных между сетями. Маршрутизатор анализирует IP-адрес получателя и определяет, в какую сеть нужно отправить данные.

Транспортный уровень отвечает за надежную доставку данных между приложениями. Он обеспечивает, чтобы данные были доставлены в правильной последовательности, без потерь и дубликатов. Основные задачи:

- разделение данных на пакеты для передачи по сети;
- установление соединения между отправителем и получателем;
- обеспечение надежной доставки данных;
- управление потоком данных, чтобы избежать перегрузки сети;
- восстановление данных в правильной последовательности.

Ключевые протоколы:

- TCP (Transmission Control Protocol). Обеспечивает надежную доставку данных с гарантией порядка и отсутствия потерь. TCP использует механизмы подтверждения получения данных и повторной отправки потерянных пакетов.
- UDP (User Datagram Protocol). Обеспечивает быструю доставку данных без гарантии надежности и порядка. UDP не требует установления соединения и не использует механизмы подтверждения получения данных. Он подходит для приложений, где важна скорость, а потеря данных не критична (например, потоковое видео).





Прикладной уровень – это самый верхний уровень модели TCP/IP. Он предоставляет интерфейс для приложений, позволяющий им обмениваться данными по сети. Основные задачи:

- предоставление стандартных протоколов для различных типов приложений;
- преобразование данных из формата приложения в формат, понятный сети, и наоборот;
- обеспечение безопасности и аутентификации.

Ключевые протоколы:

- HTTP (Hypertext Transfer Protocol). Используется для передачи данных между веб-браузерами и веб-серверами.
- SMTP (Simple Mail Transfer Protocol). Используется для отправки электронной почты.
- FTP (File Transfer Protocol). Используется для передачи файлов между компьютерами.
- DNS (Domain Name System). Используется для преобразования доменных имен (например, google.com) в IP-адреса.

Прикладной уровень – это как интерфейс пользователя в вашем любимом приложении. Он предоставляет вам удобный способ для взаимодействия с сетью, не требуя знания технических деталей о том, как данные передаются по проводам или по воздуху.

Модель TCP/IP – это сложная, но элегантная система, которая позволяет устройствам по всему миру общаться друг с другом. Понимание основных принципов работы TCP/IP необходимо для разработки эффективных и надежных Android-приложений, работающих с сетью. В следующих разделах мы более подробно рассмотрим протокол HTTP, который является одним из ключевых компонентов прикладного уровня и играет важную роль во взаимодействии между Android-приложениями и веб-серверами.

# КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое тема в Android? Чем она отличается от стиля?
2. На что влияет тема в Android? (Отдельный View, Activity, приложение целиком)
3. Где располагается файл themes.xml?
4. Какой шаблон проекта Android Studio автоматически генерирует файл themes.xml?
5. Какие основные цветовые атрибуты Material Design вы знаете?
6. Для чего используются атрибуты colorPrimary и colorSecondary?
7. Какова роль атрибутов colorPrimaryVariant и colorSecondaryVariant?
8. Что определяют атрибуты colorOnPrimary и colorOnSecondary?
9. Каково назначение файла colors.xml?
10. Как применить тему ко всему приложению? Где это указывается?
11. Что такое ночной режим (Dark Theme) в Android?
12. Как создать тему для ночного режима? Где располагается соответствующий файл?
13. Как Android определяет, какую тему использовать (дневную или ночную)?
14. Можно ли программно изменить тему приложения? Если да, то как?
15. Почему важно тестировать приложение в светлой и темной темах?
16. Что такое Context в Android?
17. Для чего нужен Context?
18. Какие классы наследуются от Context?
19. Что такое Application Context? Какова его область видимости?
20. Что такое Activity Context? Какова его область видимости?
21. В каких случаях следует использовать Application Context?
22. В каких случаях следует использовать Activity Context?
23. Как получить экземпляр Application Context?
24. Как получить экземпляр Activity Context?
25. Что произойдет, если использовать Activity Context там, где нужен Application Context (или наоборот)?
26. Почему важно правильно выбирать Context? Какие проблемы могут возникнуть при неправильном выборе?
27. Как Android поддерживает локализацию приложения?
28. Где хранятся строковые ресурсы для разных языков?
29. Как получить строковый ресурс из кода?





30. Как использовать getString() для получения локализованной строки?

31. Что такое Intent? Для чего он нужен?

32. Какие основные компоненты приложения могут быть запущены с помощью Intent?

33. Что такое Intent Filter? Какова его роль?

34. Где определяются Intent Filter?

35. Какие три основных параметра используются для определения соответствия между Intent и Intent Filter?

36. Что такое Action в Intent Filter? Приведите примеры Action.

37. Что такое Data в Intent Filter? Какие атрибуты используются для описания данных?

38. Что такое Category в Intent Filter? Приведите примеры Category.

39. Чем отличаются Explicit и Implicit Intent?

40. Что такое Extra в Intent? Для чего они используются?

41. Какие типы данных можно передавать через Extra?

42. Что такое сериализация? Какие интерфейсы используются для сериализации объектов в Android?

43. В чем разница между Serializable и Parcelable?

44. Что такое EditText?

45. Как создать EditText в XML и в коде?

46. Что такое атрибут inputType? Приведите примеры его значений.

47. Какие методы EditText вы знаете?

48. Для чего нужен метод setOnKeyListener()?

49. Что такое TextWatcher?

50. Какие методы определены в интерфейсе TextWatcher?

51. В какой последовательности вызываются методы TextWatcher?

52. В каком методе TextWatcher можно изменять текст EditText?

53. Для чего используется класс InputMethodManager?

54. Как скрыть виртуальную клавиатуру с помощью InputMethodManager?

55. Для чего нужны методы requestFocus() и setFocusable()?

56. Перечислите основные UI-компоненты для отображения списков в Android.

57. Что такое Spinner? Для чего он используется?

58. Как задать элементы Spinner?

59. Что такое Adapter? Какова его роль?

60. Какие интерфейсы должен реализовать класс Adapter для Spinner?





61. Что такое GridView? Чем он отличается от ListView?

62. Что такое ListView? Почему рекомендуется использовать RecyclerView вместо ListView?

63. Что такое RecyclerView? В чем его преимущества?

64. Какие три основных компонента необходимы для работы с RecyclerView?

65. Что такое ViewHolder? Какова его роль?

66. Что такое LayoutManager? Какие основные реализации LayoutManager вы знаете?

67. В чем разница между LinearLayoutManager и GridLayoutManager?

68. Какие методы RecyclerView.Adapter необходимо переопределить?

69. Что делает метод onCreateViewHolder()?

70. Что делает метод onBindViewHolder()?

71. Что делает метод getItemCount()?

72. Что такое LayoutInflater? Для чего он используется?

73. Как RecyclerView переиспользует View-элементы?

74. Что такое View Cache и Recycled View Pool?

75. Как RecyclerView определяет, нужно ли создавать новый ViewHolder или можно переиспользовать существующий?

76. Как отобразить RecyclerView в Activity/Fragment?

77. Какие основные операции можно выполнять с адаптером RecyclerView?

78. В каких случаях следует использовать notifyDataSetChanged()? Почему это не всегда эффективно?

79. Для чего нужны методы notifyItemInserted() и notifyItemRangeChanged()?

80. Как добавить новый элемент в список и уведомить адаптер об этом?

81. Где лучше всего устанавливать OnClickListener для элементов списка RecyclerView?

82. Как обработать нажатие на элемент списка и получить доступ к данным этого элемента?

83. Как удалить элемент из списка и уведомить адаптер об этом?

84. Почему важно вызывать notifyItemRangeChanged() после удаления элемента?

85. В какой последовательности нужно вызывать методы removeAt(), notifyItemRemoved() и notifyItemRangeChanged() при удалении элемента? Почему?

86. Что произойдет, если вызвать notifyDataSetChanged() вместо notifyItemInserted() и notifyItemRangeChanged() при добавлении элемента?





87. Что произойдет, если забыть вызвать notifyItemRangeChanged() после удаления элемента?

88. Какие преимущества дает переиспользование View в RecyclerView?

89. Почему важно использовать notifyItem... методы для обновления данных вместо простого обновления списка и вызова notifyDataSetChanged()?

90. Какие могут быть проблемы, если напрямую изменять данные в списке, не уведомляя об этом адаптер?

91. Опишите шаги для реализации удаления элемента из RecyclerView, начиная с момента нажатия на элемент пользователем.

92. Почему не стоит выполнять тяжелые операции (например, загрузку изображений) непосредственно в методе onBindViewHolder?

93. Что такое ViewType в RecyclerView и когда он используется?

94. Как можно реализовать возможность перемещения элементов в RecyclerView (drag and drop)?

95. Что такое standalone-приложение? Приведите примеры.

96. Какие характеристики определяют standalone-приложение?

97. Назовите преимущества standalone-приложений.

98. Какие недостатки у standalone-приложений по сравнению с клиент-серверными?

99. В чем суть клиент-серверной архитектуры? Какие компоненты она включает?

100. Какие задачи решает клиент в клиент-серверном приложении?

101. Каковы задачи сервера в клиент-серверном приложении?

102. Почему для клиент-серверных приложений необходимо подключение к сети?

103. Какие преимущества предоставляет клиент-серверный подход с точки зрения обновления данных?

104. Каким образом клиент-серверные приложения решают проблему вычислительных ресурсов?

105. Как обеспечивается безопасность данных в клиент-серверной архитектуре?

106. В чем преимущество клиент-серверных приложений в отношении размера приложения на устройстве пользователя?

107. Какие факторы следует учитывать при выборе между standalone-приложением и клиент-серверным приложением?

108. Что такое сервер в контексте клиент-серверной архитектуры?

109. Какие два типа серверов существуют? В чем их различие?





110. Кто занимается разработкой серверной части приложения?

111. Какие технологии обычно используются в backend-разработке?

112. Опишите основные шаги работы сервера при обработке запроса от клиента.

113. Что такое база данных и какую роль она играет в работе серверного приложения?

114. Почему важно отделять базу данных от серверного приложения?

115. Какие меры принимаются для обеспечения отказоустойчивости и масштабируемости серверной части?

116. Что такое балансировщик нагрузки и как он работает?

117. Зачем нужны реплики базы данных?

118. Что такое REST?

119. Что означает аббревиатура REST?

120. Какова основная цель REST API?

121. В чем заключается принцип "модель клиент-сервер" в REST?

122. Что означает "отсутствие состояния" (stateless) в контексте REST API?

123. Почему важно, чтобы REST API поддерживал кэширование?

124. Что такое "единообразие интерфейса" в REST API?

125. Что означает принцип "слои" в REST API?

126. Что такое "код по требованию" в REST (и является ли этот принцип обязательным)?

127. Как определяется "API" (Application Programming Interface)?

128. Что такое REST API и как он связан с общим понятием API?

129. Какова роль спецификации REST API?

130. Приведите примеры инструментов для создания спецификаций REST API.

131. Что такое TCP/IP и для чего он нужен?

132. Почему важна декомпозиция при разработке сложных систем, таких как сетевое взаимодействие?

133. Что такое протокол в контексте передачи данных?

134. Какие типы протоколов существуют?

135. В чем разница между высокоуровневыми и низкоуровневыми протоколами?

136. Что такое пакет данных? Какие части он включает?

137. Перечислите уровни модели TCP/IP.

138. Какую функцию выполняет канальный уровень TCP/IP?

139. Какие протоколы обычно используются на канальном уровне?

140. Что такое Ethernet и для чего он нужен?

141. Что такое Wi-Fi и как он работает?

142. Какую функцию выполняет сетевой уровень TCP/IP?

143. Что такое IP-адрес и для чего он нужен?

144. Что такое маршрутизатор и как он помогает передавать данные между сетями?

145. Какую функцию выполняет транспортный уровень TCP/IP?

146. Какие протоколы используются на транспортном уровне?

147. В чем разница между TCP и UDP? В каких случаях целесообразно использовать каждый из них?

148. Какую функцию выполняет прикладной уровень TCP/IP?

149. Приведите примеры протоколов прикладного уровня.

150. Что такое HTTP и для чего он используется?

151. В чем важность понимания модели TCP/IP для Android-разработчика?

152. Как уровни TCP/IP взаимодействуют друг с другом для обеспечения передачи данных? Приведите пример, иллюстрирующий этот процесс.

153. Представьте, что данные нужно отправить с одного приложения на другое через Интернет. Опишите путь, который эти данные пройдут, используя уровни TCP/IP.

154. Объясните на простом примере принцип маршрутизации данных в Интернете.





# ПРАКТИЧЕСКАЯ РАБОТА

## Верстка экранов

Мобильное приложение практически всегда состоит из двух частей:

- Пользовательского интерфейса. То, что видит пользователь, когда открывает приложение.
- Логической части. То, как пользовательский интерфейс будет реагировать на действия пользователя: какие запросы отправятся в сеть, какие вычисления будут выполнены на устройстве.

Пользовательский интерфейс, или UI — от английского User Interface.

Разработкой внешнего вида приложения в большинстве компаний занимаются отдельные люди — дизайнеры интерфейсов. Результат их работы — это изображения всех экранов приложения, а таких экранов у одного приложения может быть много.

Дело разработчиков — кодом отобразить на экранах этот интерфейс в том виде, как его нарисовал дизайнер. Создание интерактивного пользовательского интерфейса называется вёрсткой.

В своей работе дизайнеры интерфейсов применяют разные инструменты: Figma, Sketch, Zeplin или другие графические приложения.

У сервиса Figma есть настольная версия и веб-приложение. Эти версии практически не отличаются по интерфейсу.

На рисунке 5 представлен пример отображения макета в Figma.

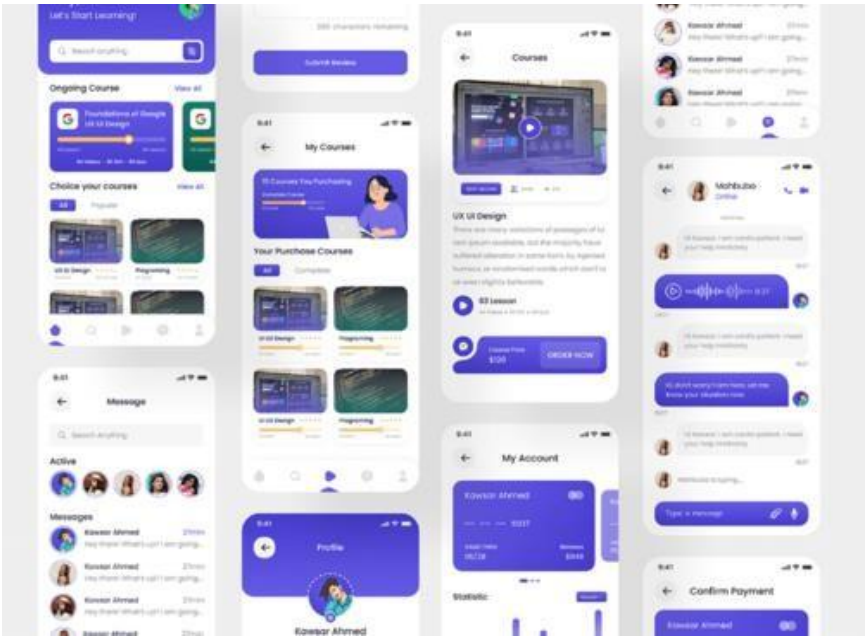


*Рисунок 5. Пример макета в Figma*

Так может выглядеть макет от дизайнера. Кажется, всё просто, но сверстать что-то подобное «на глазок» нельзя. Важно соблюсти все размеры и отступы элементов друг от друга. Нужно прочесть, понять и применить в вёрстке:

- размер, цвет и стиль шрифтов;
- радиус скругления углов;
- размер теней.

Всё, что нарисовано в макете, должно быть точно реализовано в приложении.

Чтобы увидеть параметры элементов макета необходимо на него нажать, и перейти в правой части экрана в меню Properties (рис. 6).

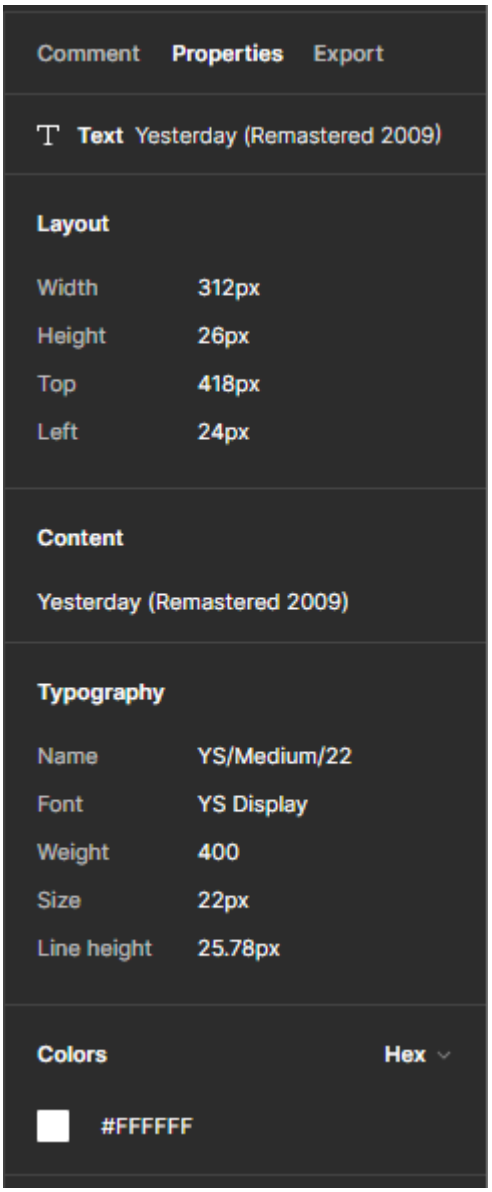


*Рисунок 6. Меню Properties*

Здесь отображены параметры, относящиеся к выделенному элементу.

Чтобы отрисовать этот элемент в приложении, нужно знать:

- Размер элемента. Раздел Layout, поля width (ширина) и height (высота), отступы.
- Стиль шрифта. Раздел Typography, поля:
- Font — шрифт, используемый в тексте. В большинстве случаев компании имеют свой набор шрифтов и предоставляют их для разработки.
- Size — размер шрифта.
- Цвет шрифта.

Чтобы посмотреть размер того или иного элемента на макете необходимо нажать на него (рис. 7).

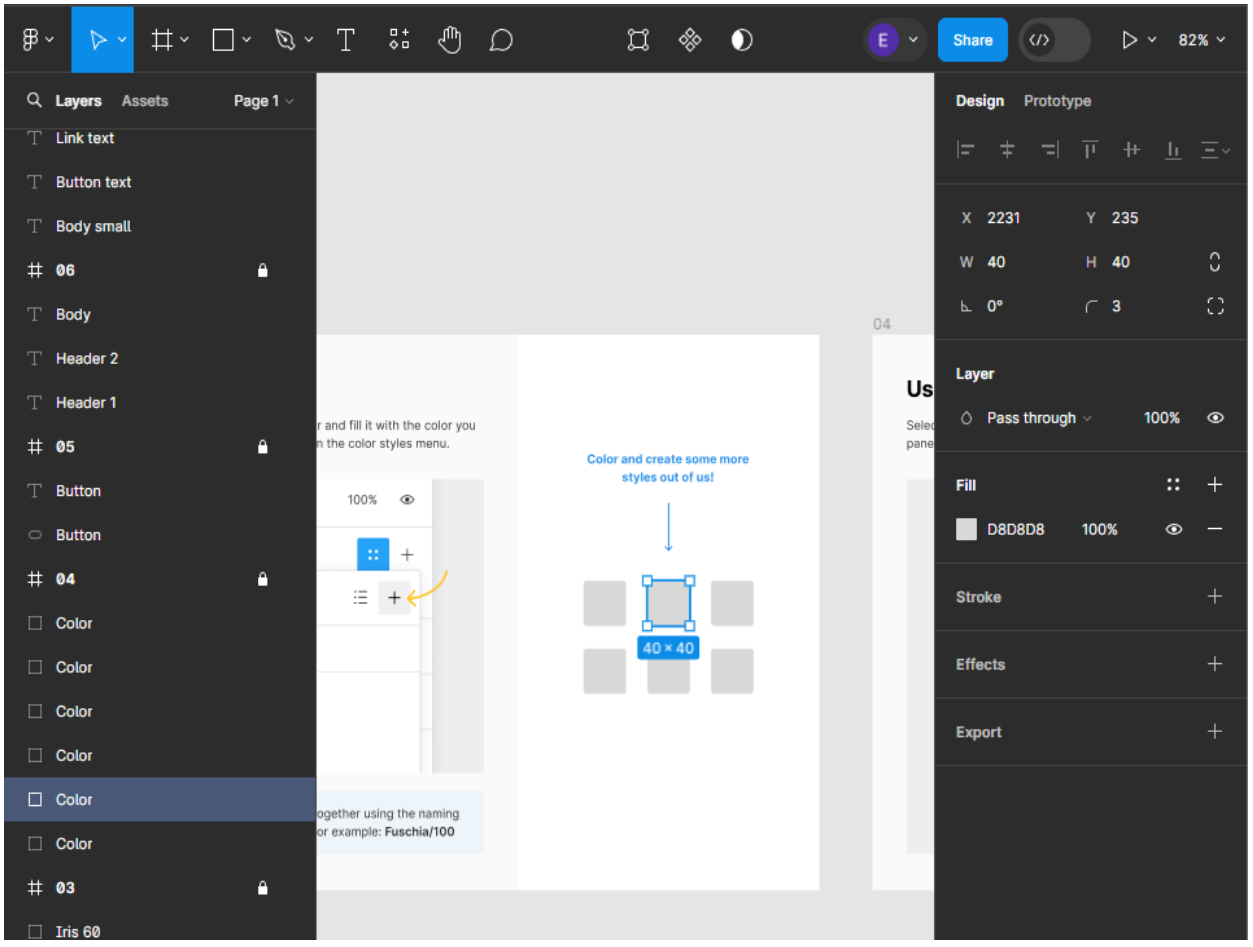


*Рисунок 7. Отображение размера элемента макета*

Для того чтобы посмотреть, на котором расположен объект от других элементов макета, необходимо зажать ALT и намести курсор мыши на элемент, отступ от которого необходимо узнать (рис. 8).

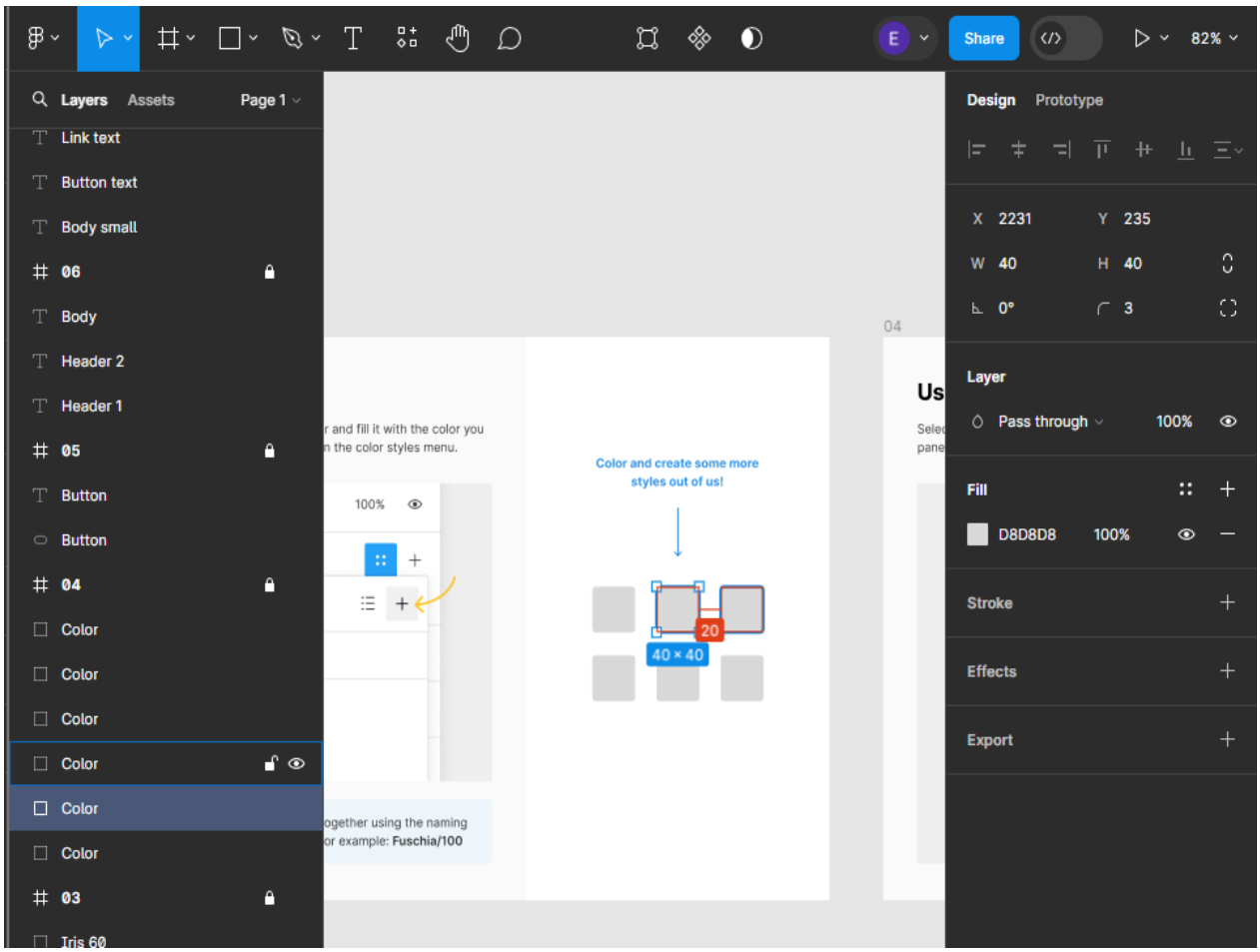


*Рисунок 8. Отображение отступов на макете*

Также в Figma есть режим разработчика или Dev Mode. Dev Mode — это режим разработки в Figma, который позволяет быстро переводить проекты в код.

В Dev Mode доступен код элементов разных платформ, который можно брать за основу для разработки. Есть возможность создавать плагины с генерацией кастомного кода под различные фреймворки.

В конце января 2024 года Figma закрыла Dev Mode для пользователей бесплатного тарифа.

Структура и синтаксис ресурсных файлов

Существует 4 типа ресурсов.

**Строковые ресурсы**

В корневом каталоге приложения есть папка res/values с файлом strings.xml. Этот файл содержит список всех строковых ресурсов приложения.

Строковые ресурсы предоставляют текстовые константы для приложения. В таких ресурсах может храниться дополнительное форматирование или стиль текста.

```
<resources>
<string name="app_name">My flroject</string>
```

```
</resources>
```

Все строковые ресурсы должны быть перечислены между тегами <resources> ... </resources>. Для объявления нового строкового ресурса нужно открыть тег <string, указать параметр name — имя вашего строкового ресурса и после закрытой угловой скобки > указать значения ресурса — текст, который будет храниться в ресурсе. После чего нужно закрыть тег </string>.

```
<resources>
<string name="app_name"> My flroject </string>
<string name="title_hello">Привет, Мир!</string>
</resources>
```

После добавления текста в строковые ресурсы мы можем использовать его в нашей XML-разметке интерфейса:

```
<oxml version="1.0" encoding="utf-8"o>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android
"
android:layout_width="match_parent"
android:layout_height="match_parent">

<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center"
android:text="@string/title_hello" />
</FrameLayout>
```

Теперь вместо указания текста мы можем использовать строковый ресурс:

```
android:text="@string/title_hello"
```

**Dimensions (размеры)**

Ещё один вид ресурса в ОС Android — размеры, они же dimensions.

Dimensions имеют тот же принцип, что и строковые ресурсы. Вы описываете в заданном формате каждый ресурс и присваиваете ему нужный размер. Dimensions хранятся в файле dimens.xml в папке res/values.

Формат объявления dimensions аналогичен строковым ресурсам. Разница лишь в теге <dimen/>, а вместо текстового значения указывается размер в sp (для текстов) или dp (всех остальных размеров).

В разметке XML dimensions используются и как строковые ресурсы, но вместо пути @string указывается путь @dimen:

```
<oxml version="1.0" encoding="utf-8"o>
```

```
    <FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android
"
    android:layout_width="match_parent"
android:layout_height="match_parent">

    <TextView
    android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center"
android:padding="@dimen/medium_padding"
android:text="@string/title_hello"
android:textSize="@dimen/primary_text_size" />
    </FrameLayout>
```

**Цвета**

Использовать цветовые коды при разработке приложения не всегда удобно. Более того, реальные приложения зачастую имеют ограниченный набор цветов, заданный дизайн-системой, и использование сторонних цветов запрещено.

Чтобы каждый раз не искать нужный код и не бояться ошибки, основные цвета выносятся в ресурсы приложения.

Цветовые ресурсы хранятся в каталоге res/values в файле colors:

```
<oxml version="1.0" encoding="utf-8"o>
<resources>
<color  name="purple_200">#FFBB86FC</color>
<color  name="purple_500">#FF6200EE</color>
<color  name="purple_700">#FF3700B3</color>
<color name="teal_200">#FF03DAC5</color>
<color name="teal_700">#FF018786</color>
<color name="black">#FF000000</color>
<color  name="white">#FFFFFFFF</color>
</resources>
```

Выше небольшой пример файла colors. Здесь всё идентично strings и dimens. Для указания цвета используется тег <color>, а в качестве значения передаётся цветовой код. Использование цветов в XML-разметке ничем не отличается от работы со строками и размерами:

```
<oxml version="1.0" encoding="utf-8"o>
```





```
    <FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android
"
    android:layout_width="match_parent"
android:layout_height="match_parent"
android:background="@color/white">

    <TextView
    android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center"
android:padding="@dimen/medium_padding"
android:text="@string/title_hello"
android:textColor="@color/black"
android:textSize="@dimen/primary_text_size" />
    </FrameLayout>
```

В качестве пути указываем @color, после чего через / указываем название цветового ресурса: android:textColor="@color/black".

**Растровые и векторные изображения**

Drawable, они же растровые и векторные изображения. Сюда относятся все статические (известные на момент написания кода) изображения, иконки и другие векторные изображения приложения. Располагаются они в папке res/drawable.

Папка drawable хранит следующие виды изображений:

Bitmap File — изображения с расширением .png, .webp, .jpg, .gif.

Layer List — специальный вид файлов, который позволяет отрисовать массив других drawable. Например, с помощью данного вида drawable можно отрисовать тени путём наложения нескольких слоёв один на другой.

State List — XML-файл, который имеет ссылки на другие drawable, позволяющий описать разные состояния одной View. Например, мы хотим, чтобы наша кнопка была красной в активном состоянии и серой в неактивном, State List решает такую задачу.

Это часто встречающиеся виды drawable, но есть и другие. Подробнее обо всех видах в официальной документации.

Золотого правила, когда выносить в ресурсы, а когда не выносить, нет. Те размеры, которые часто переиспользуются и будут изменяться, например, общие отступы от краёв экрана или размеры иконок в приложении, стоит хранить в ресурсах. Одноразовые размеры выносить необходимости нет.

В крупных проектах часто создаётся целая дизайн-система, где описание (их названия и размеры) ресурсов описывают дизайнеры. В таких ситуациях задача разработчика — следовать этой системе и поддерживать соответствие ресурсов приложения дизайн-системе.

Можно создавать разные версии размеров для разных конфигураций. Например, создать ресурс, который будет иметь разный размер для портретной и горизонтальной ориентации телефона.

В Android-приложении ресурсы играют важную роль. Они позволяют формировать нужные параметры приложения в зависимости от разных конфигураций устройств.

**Задание 1**

Выбрать тему разрабатываемого приложения.

**Задание 2**

Создать или найти в открытых источниках макет мобильного приложения в Figma на выбранную Вами тему.

**Задание 3**

Описать межэкранное взаимодействие в рамках Вашего приложения.

**Задание 4**

Сверстать экраны, согласно макету. Реализовать какое-либо взаимодействие с пользовательским интерфейсом не нужно.

Реализовать переходы между Activity.

**Обязательные условия:**

- Приложение должно иметь экран авторизации и регистрации.
- В приложении должна быть реализована страница с поиском или поисковая строка.
- Полученные экраны идентичны макету в Figma.
- Свёрстанный экран адаптируется по размеры различных устройств.
- Все цвета вынесены в ресурсы.
- Размеры и строки, которые используются несколько раз, вынесены в ресурсы.





# СПИСОК ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Сомон., П. И. Волшебство Kotlin : руководство / П. И. Сомон. ; перевод с английского А. Н. Киселева.. — Москва : ДМК Пресс, 2020. — 536 с. — ISBN 978-5-97060-801-2. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/140599 (дата обращения: 05.05.2024). — Режим доступа: для авториз. пользователей.

2. Калгина, И. С. Разработка мобильных приложений : учебное пособие / И. С. Калгина. — Чита : ЗабГУ, 2022. — 163 с. — ISBN 978-5-9293-3137-4. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/363323 (дата обращения: 05.05.2024). — Режим доступа: для авториз. пользователей.

3. Алпатов, А. Н. Архитектура, проектирование и разработка программных средств : учебное пособие / А. Н. Алпатов, И. Е. Рогов. — Москва : РТУ МИРЭА, 2023. — 120 с. — ISBN 978-5-7339-1972-0. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/386189 (дата обращения: 05.05.2024). — Режим доступа: для авториз. пользователей.

4. Коузен, К. Kotlin. Сборник рецептов / К. Коузен ; перевод с английского А. Н. Киселева. — Москва : ДМК Пресс, 2021. — 220 с. — ISBN 978-5-97060-883-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/241007 (дата обращения: 21.05.2025). — Режим доступа: для авториз. пользователей.

5. Основы XML : учебное пособие. — 2-е изд. — Москва : ИНТУИТ, 2016. — 436 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100354 (дата обращения: 21.05.2025). — Режим доступа: для авториз. пользователей.

6. Стаяно, Ф. Figma проектирование и прототипирование интерфейсов : руководство / Ф. Стаяно ; перевод с английского В. С. Яценкова. — Москва : ДМК Пресс, 2024. — 370 с. — ISBN 978-5-93700-302-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/456791 (дата обращения: 21.05.2025). — Режим доступа: для авториз. пользователей.

## Сведения об авторах

Исабекова Ольга Александровна, к.э.н., доцент, доцент кафедры математического обеспечения и стандартизации информационных технологий РТУ МИРЭА, автор более 40 научных и учебно-методических работ.

Дворникова Екатерина Михайловна, старший преподаватель кафедры математического обеспечения и стандартизации информационных технологий РТУ МИРЭА, аспирант РТУ МИРЭА, автор более 20 научных и учебно-методических работ.