



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных
технологий

Отчет по практической работе №7

по дисциплине «Разработка мобильных приложений»

Выполнил:

Студент группы ИКБО-20-23

Комисарик М.А.

Проверил:

Старший преподаватель кафедры
МОСИТ

Шешуков Л.С.

Москва 2025 г.

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ	3
1.1 Service	3
1.2 Диалоговые окна	18
2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ	34
2.1 Разметка.....	34
2.2 Реализация.....	36
2.3 Тестирование	40
ЗАКЛЮЧЕНИЕ	44

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

1.1 Service

Сервис в Android — это компонент приложения, предназначенный для выполнения длительных операций или работы с ресурсоёмкими задачами в фоновом режиме без предоставления пользовательского интерфейса. Сервисы продолжают работать в фоне даже когда пользователь переключается на другие приложения. Данный механизм идеально подходит для воспроизведения аудио, выполнения сетевых запросов, обработки данных и фонового мониторинга информации.

Все сервисы наследуются от класса `Service` и, по аналогии с `Activity`, сервис имеет свой жизненный цикл и методы, связанные с ним:

- `onCreate()`,
- `onStartCommand(Intent intent, int flags, int startId)`,
- `onBind(Intent intent)`,
- `onUnbind(Intent intent)`,
- `onDestroy()`.

На рисунке 1 представлена иллюстрация жизненного цикла сервиса.

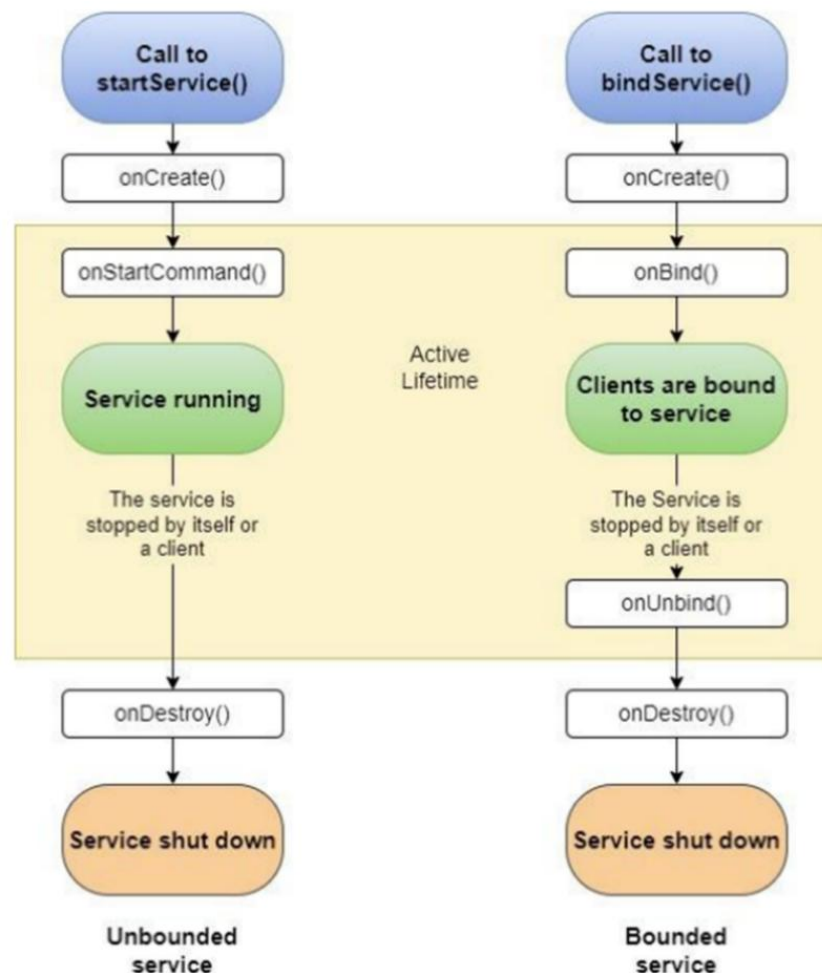


Рисунок 1 – Жизненный цикл сервиса

Метод `onCreate()` вызывается при создании сервиса. Это первый вызов, который получает сервис, и он используется для однократной инициализации, такой как создание ресурсов, которые будут использоваться в течение всего времени существования сервиса. Метод `onCreate()` вызывается только один раз перед вызовом `onStartCommand()` или `onBind()`.

Метод `onStartCommand()` вызывается каждый раз, когда компонент (например, Activity) запрашивает запуск сервиса через `startService()`. В этом методе сервис может выполнять любые операции, включая запуск потока для выполнения сложной задачи в фоне. Метод возвращает константу, указывающую, как система должна вести себя, если сервис уничтожается до того, как он завершит выполнение своей работы.

Метод *onBind()* вызывается, когда другой компонент хочет привязаться к сервису через *bindService()*. Если сервис не предоставляет интерфейс для клиентов, то он должен возвращать *null*. Метод *onBind()* вызывается только один раз для каждого клиента при первом связывании.

Метод *onUnbind()* вызывается, когда все клиенты отсоединились от определенного интерфейса сервиса. После этого вызова, если необходимо, сервис может остановить себя через *stopSelf()*.

Метод *onDestroy()* вызывается, когда сервис больше не используется и собирается быть уничтоженным. Это последний вызов, который получает сервис, и он используется для освобождения ресурсов, таких как потоки, зарегистрированные приемники, обработчики и т.д.

В качестве примера, создадим сервис, который будет воспроизводить музыку. Предварительно загрузим медиафайл формата *mp3* в наш проект.

Чтобы это сделать, необходимо создать в папке *res* папку *raw*. Именно в этой папке хранятся различные файлы, которые сохраняются в исходном виде. После этого нужно загрузить медиафайл в папку *res/raw* таким же способом, как добавляются изображения (Рисунок 2).

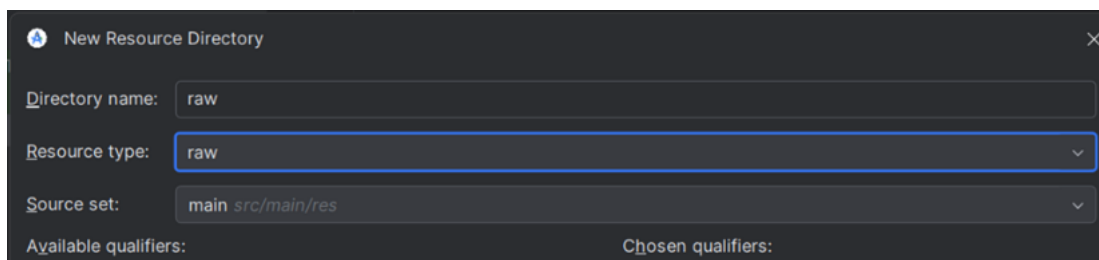


Рисунок 2 – Создание директории для хранения ресурсов в исходном виде

После того, как файл добавлен нужно создать новый класс сервиса. Это можно сделать, нажав правой кнопкой мыши на *"java"* → *"New"* → *"Service"* → *"Service"* (Рисунок 3).

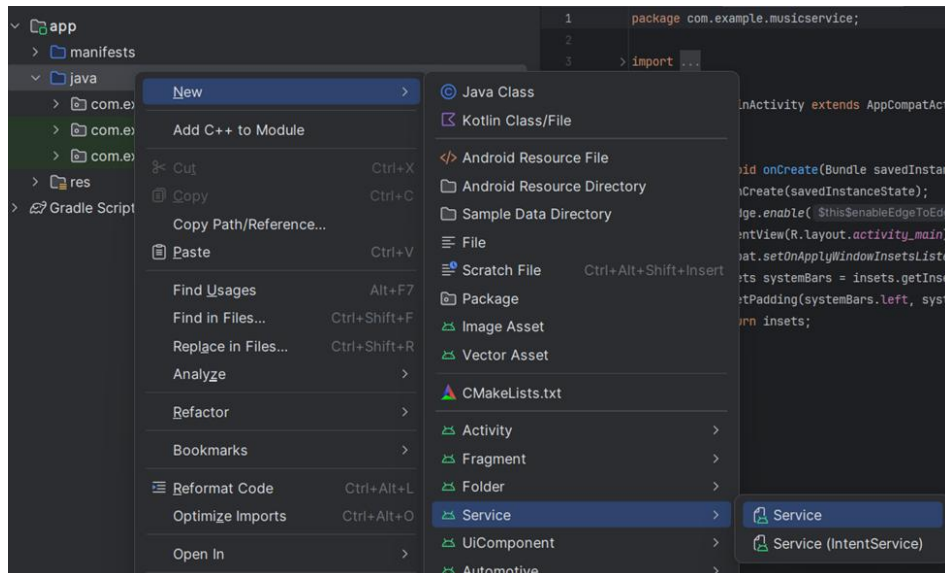


Рисунок 3 – Создание нового сервиса через контекстное меню

При создании нового сервиса через диалоговое окно "New Android Component" необходимо указать имя класса (например, MusicService), которое будет использоваться для реализации сервиса. В этом же окне доступны настройки Enabled и Exported — флаги, определяющие базовое поведение сервиса. Значение Enabled включено по умолчанию, что позволяет системе создавать экземпляр сервиса, а параметр Exported отключен, ограничивая доступ к сервису только в рамках текущего приложения (Рисунок 4).

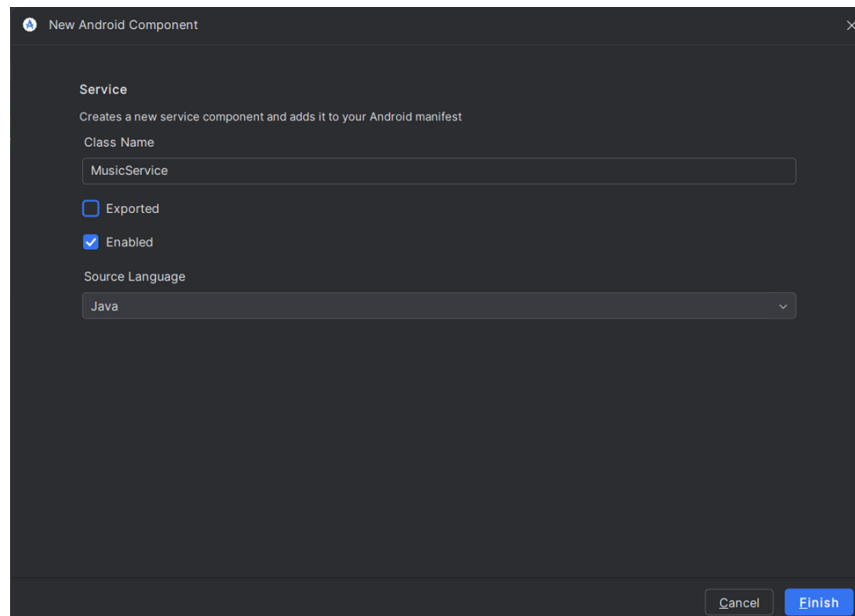


Рисунок 4 – Окно создания нового сервиса

Созданный сервис с указанными настройками автоматически добавляется в файл манифеста.

Теперь нужно прописать логику работы сервиса (Рисунок 5).

```
public class MusicService extends Service {
    private static final String TAG = "MusicService";
    private MediaPlayer mediaPlayer;

    @Override
    public void onCreate() {
        super.onCreate();
        // Инициализация медиаплеера
        mediaPlayer = MediaPlayer.create(context: this, R.raw.music);
        mediaPlayer.setLooping(true); // За цикливание воспроизведения
        mediaPlayer.setVolume(leftVolume: 100, rightVolume: 100);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int
        startId) {
        if (!mediaPlayer.isPlaying()) {
            mediaPlayer.start();
            Log.d(TAG, msg: "Музыка начала играть");
        }
        return START_STICKY;
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        if (mediaPlayer.isPlaying()) {
            mediaPlayer.stop();
            mediaPlayer.release();
            Log.d(TAG, msg: "Музыка остановлена и ресурсы освобождены");
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null; // Для сервисов без привязки возвращаем null
    }
}
```

Рисунок 5 – Описание логики работы сервиса

Для воспроизведения музыкального файла сервис будет использовать компонент MediaPlayer.

В сервисе переопределяются четыре метода жизненного цикла. Метод `onBind()` возвращает `null`, так как наш сервис не должен иметь возможность привязки.

В методе `onCreate()` инициализируется медиа-проигрыватель с помощью музыкального ресурса, который добавлен в папку `res/raw`.

В методе `onStartCommand()` начинается воспроизведение.

Метод `onStartCommand()` может возвращать одно из значений, определяющих поведение сервиса при неожиданном завершении процесса системой:

- `START_STICKY`: сервис автоматически перезапускается, но метод `onStartCommand()` вызывается с параметром `Intent`, равным `null`,
- `START_REDELIVER_INTENT`: сервис перезапускается с тем же объектом `Intent`, который был передан при последнем вызове,
- `START_NOT_STICKY`: сервис не перезапускается автоматически после завершения.

В методе `onDestory()` останавливается воспроизведение и освобождаются ресурсы.

При создании сервиса через контекстное меню Android Studio (`New → Service → ...`), объявление сервиса автоматически добавляется в файл `AndroidManifest.xml`. Однако если автоматическое добавление не сработало, требуется вручную прописать объявление сервиса в манифесте (Рисунок 6).

```
<service
    android:name=".MusicService"
    android:enabled="true"
    android:exported="false" />
```

Рисунок 6 – Объявление сервиса в манифесте

Регистрация сервиса осуществляется в узле `<application>` с помощью элемента `<service>`. Основные атрибуты:

- `android:name` — обязательный атрибут, содержащий полное имя класса сервиса (включая пакет),
- `android:enabled` — определяет, может ли система создавать сервис (по умолчанию "true"),
- `android:exported` — указывает, могут ли компоненты других приложений обращаться к сервису,
- `android:permission` — определяет разрешения, необходимые для взаимодействия с сервисом,
- `android:process` — задает имя процесса для запуска сервиса,
- `android:isolatedProcess` — если "true", сервис запускается в изолированном процессе с ограниченными правами,
- `android:icon` — значок сервиса, представляет собой ссылку на ресурс `drawable`,
- `android:label` — название сервиса, которое отображается пользователю.

В методе `onCreate()` класса `MainActivity` реализуем запуск фонового сервиса с помощью метода `startService()`. Создаем объект `Intent`, явно указывая целевой сервис `MusicService.class`, который требуется запустить. Передаем этот `Intent` в `startService()`, что инициирует создание и старт сервиса, если он не был ранее запущен.

В методе `onDestroy()` класса `MainActivity` обеспечиваем корректную остановку сервиса при завершении работы. Формируем аналогичный `Intent` с указанием того же класса сервиса `MusicService.class` и передаем его в метод `stopService()`. Это приводит к вызову метода `onDestroy()` сервиса и освобождению всех занимаемых им ресурсов (Рисунок 7).

```

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent startIntent = new Intent( packageContext: this, MusicService.class);
        startService(startIntent);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Intent stopIntent = new Intent( packageContext: this, MusicService.class);
        stopService(stopIntent);
    }
}

```

Рисунок 7 – Запуск и остановка сервиса через класс MainActivity

Мы рассмотрели создание Unbound Service, который работает в фоновом режиме независимо от активности других компонентов приложения. Такой сервис идеально подходит для выполнения длительных операций, таких как воспроизведение музыки или загрузка файлов, когда не требуется постоянное взаимодействие с другими частями приложения. Unbound Service продолжает работать даже после того, как компонент, который его запустил, был уничтожен, пока не завершит свою задачу или пока его явно не остановят.

Теперь перейдём к Bound Service, который предоставляет более тесное взаимодействие между сервисом и другими компонентами приложения, такими как Activity или Fragment. Bound Service позволяет клиентам привязываться к нему, вызывать его методы напрямую и получать данные в реальном времени. Этот тип сервиса особенно полезен, когда требуется постоянный обмен информацией, например, для управления воспроизведением музыки с возможностью паузы, перемотки или изменения громкости. В отличие от Unbound Service, Bound Service автоматически уничтожается, когда все клиенты отвязываются, если только он не был также запущен через startService().

Для реализации Bound Service необходимо добавить в класс сервиса специальный механизм привязки. Внутри класса MusicService создаётся внутренний класс MusicBinder, который наследуется от стандартного класса Binder. Этот класс содержит всего один метод getService(), предназначенный для получения ссылки на текущий экземпляр сервиса. Такой подход является стандартным для реализации привязки в Android и позволяет компонентам приложения получить доступ к публичным методам сервиса.

Объект класса MusicBinder создаётся как поле сервиса и инициализируется при создании экземпляра MusicService. Это гарантирует, что все клиенты будут получать один и тот же объект для привязки. В методе onBind() сервиса возвращается этот заранее созданный объект binder, когда компоненты запрашивают привязку к сервису (Рисунок 8).

```
public class MusicService extends Service {
    private static final String TAG = "MusicService";
    private MediaPlayer mediaPlayer;
    private final IBinder binder = new MusicBinder();

    // Класс Binder, который возвращает экземпляр этого сервиса
    public class MusicBinder extends Binder {
        MusicService getService() {
            return MusicService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }
}
```

Рисунок 8 – Реализация базовой структуры привязки

В сервис добавляются публичные методы, которые будут доступны для вызова из Activity после успешной привязки. Метод playMusic() проверяет состояние MediaPlayer и запускает воспроизведение музыки, если оно не было начато ранее. При этом в лог выводится сообщение о начале воспроизведения, что помогает отслеживать работу приложения во время отладки.

Метод pauseMusic() выполняет приостановку воспроизведения, если музыка в данный момент играет. Это позволяет временно остановить звук без

полной остановки сервиса. Как и в случае с `playMusic()`, состояние изменений фиксируется в системном логге. Для полной остановки музыки предназначен метод `stopMusic()`, который не только останавливает воспроизведение, но и возвращает `MediaPlayer` в исходное состояние с помощью метода `prepare()`, что необходимо для последующих запусков.

Дополнительно реализован метод `isPlaying()`, который возвращает текущее состояние воспроизведения (Рисунок 9).

```
// Публичные методы, которые могут вызывать Activity
public void playMusic() {
    if (!mediaPlayer.isPlaying()) {
        mediaPlayer.start();
        Log.d(TAG, "Музыка начала играть");
    }
}

public void pauseMusic() {
    if (mediaPlayer.isPlaying()) {
        mediaPlayer.pause();
        Log.d(TAG, "Музыка приостановлена");
    }
}

public void stopMusic() {
    if (mediaPlayer.isPlaying()) {
        mediaPlayer.stop();
        // Подготовка к следующему запуску
        try {
            mediaPlayer.prepare();
        } catch (Exception e) {
            Log.e(TAG, "Ошибка подготовки медиаплеера", e);
        }
        Log.d(TAG, "Музыка остановлена");
    }
}

public boolean isPlaying() {
    return mediaPlayer.isPlaying();
}
```

Рисунок 9 – Методы управления сервисом

В файле разметки `activity_main.xml` создадим вертикальный `LinearLayout` как корневой элемент интерфейса. Внутри него разместим четыре кнопки управления с уникальными идентификаторами: кнопку с идентификатором `playButton` для запуска воспроизведения, кнопку с идентификатором `pauseButton`

для постановки на паузу, кнопку с идентификатором stopButton для полной остановки и кнопку с идентификатором statusButton для проверки состояния (Рисунок 10).

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <Button
        android:id="@+id/playButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Воспроизвести"
        android:layout_marginBottom="16dp"/>

    <Button
        android:id="@+id/pauseButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Пауза"
        android:layout_marginBottom="16dp"/>

    <Button
        android:id="@+id/stopButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Остановить"
        android:layout_marginBottom="16dp"/>

    <Button
        android:id="@+id/statusButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Проверить статус"/>

</LinearLayout>
```

Рисунок 10 – Разметка для MainActivity

В классе MainActivity создадим механизм привязки к сервису MusicService. Объявим поле musicService для хранения ссылки на сервис и флаг isBound, который будет отслеживать состояние подключения. Реализуем объект ServiceConnection, содержащий два основных метода для работы с привязкой.

В методе onServiceConnected получим экземпляр сервиса через переданный IBinder, преобразовав его к типу MusicService.MusicBinder. Установим флаг isBound в true и выведем уведомление о успешном подключении. Метод onServiceDisconnected сработает при неожиданном

разрыве связи с сервисом, сбросит флаг `isBound` и проинформирует пользователя об отключении (Рисунок 11).

```
public class MainActivity extends AppCompatActivity {
    private MusicService musicService;
    private boolean isBound = false;

    // Соединение с сервисом
    private final ServiceConnection serviceConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            MusicService.MusicBinder binder = (MusicService.MusicBinder) service;
            musicService = binder.getService();
            isBound = true;
            Toast.makeText(context: MainActivity.this, text: "Сервис подключен", Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {
            isBound = false;
            Toast.makeText(context: MainActivity.this, text: "Сервис отключен", Toast.LENGTH_SHORT).show();
        }
    };
};
```

Рисунок 11 – Механизм привязки к сервису в MainActivity

В методе `onCreate()` `MainActivity` реализуем привязку к сервису `MusicService` с помощью метода `bindService()`. Этот метод принимает три ключевых параметра.

Первым параметром передаем `Intent`, который явно указывает на класс сервиса `MusicService.class`, что позволяет системе точно идентифицировать, к какому сервису нужно подключиться.

Вторым параметром идет объект `ServiceConnection` (`serviceConnection`), который мы создали ранее. Этот объект содержит callback-методы `onServiceConnected()` и `onServiceDisconnected()`, которые система вызовет при успешном подключении к сервису или при неожиданном разрыве соединения соответственно.

Третий и последующие параметры — флаги, определяющие поведение привязки. Мы используем константу `BIND_AUTO_CREATE`, которая указывает системе автоматически создать сервис, если он еще не был запущен. Это стандартный вариант для большинства случаев. Другие возможные флаги включают `BIND_ABOVE_CLIENT` (приоритетное обслуживание),

BIND_IMPORTANT (повышенный приоритет) или BIND_WAIVE_PRIORITY (отказ от приоритета), однако на этом наборе не ограничиваются (Рисунок 12).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // Привязка к сервису
    Intent intent = new Intent(packageContext: this, MusicService.class);
    bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
}
```

Рисунок 12 – Привязка к сервису

В методе onCreate() MainActivity настраиваем обработку нажатий для кнопок управления музыкальным сервисом. Сначала получаем ссылки на все кнопки через findViewById(), используя их идентификаторы из разметки: playButton для воспроизведения, pauseButton для паузы, stopButton для остановки и statusButton для проверки состояния.

Устанавливаем для каждой кнопки соответствующий обработчик нажатий, который будет вызывать методы нашего сервиса. При нажатии на кнопку воспроизведения будет вызываться метод playMusic(), кнопка паузы активирует метод pauseMusic(), а кнопка остановки запустит метод stopMusic(). Отдельная кнопка статуса использует метод isPlaying() для проверки текущего состояния воспроизведения, на основе которого выводит всплывающее уведомление (Toast). Важно отметить, что все эти действия выполняются только при успешном подключении к сервису, что контролируется проверкой флага isBound перед каждым вызовом методов сервиса (Рисунок 13).


```

protected void onCreate(Bundle savedInstanceState) {
    // Настройка кнопок
    Button playButton = findViewById(R.id.playButton);
    Button pauseButton = findViewById(R.id.pauseButton);
    Button stopButton = findViewById(R.id.stopButton);
    Button statusButton = findViewById(R.id.statusButton);

    playButton.setOnClickListener(v -> {
        if (isBound) {
            musicService.playMusic();
        }
    });

    pauseButton.setOnClickListener(v -> {
        if (isBound) {
            musicService.pauseMusic();
        }
    });

    stopButton.setOnClickListener(v -> {
        if (isBound) {
            musicService.stopMusic();
        }
    });

    statusButton.setOnClickListener(v -> {
        if (isBound) {
            boolean playing = musicService.isPlaying();
            Toast.makeText(context, this, text: "Музыка " + (playing ? "играет" : "не играет"), Toast.LENGTH_SHORT).show();
        }
    });
}
}

```

Рисунок 13 – Добавление обработчиков нажатия для кнопок

В методе `onDestroy()` реализуется отвязка от сервиса при уничтожении Activity. Проверяется флаг `isBound`, и если подключение активно, вызывается метод `unbindService()`, куда передается ранее созданный объект `serviceConnection`. После успешной отвязки флаг `isBound` сбрасывается в `false`, что фиксирует разрыв соединения с сервисом.

Этот механизм обеспечивает корректное завершение работы с сервисом и освобождение ресурсов. Важно отметить, что отвязка выполняется только при активном подключении, что предотвращает попытки разрыва несуществующего соединения. Вызов `super.onDestroy()` гарантирует правильное выполнение стандартных процедур уничтожения Activity (Рисунок 14).


```

@Override
protected void onDestroy() {
    super.onDestroy();
    // Отвязка от сервиса
    if (isBound) {
        unbindService(serviceConnection);
        isBound = false;
    }
}

```

Рисунок 14 – Отвязка от сервиса

При переходе на Bound Service реализуется двустороннее взаимодействие между Activity и сервисом. Это позволяет напрямую управлять воспроизведением музыки через вызов методов сервиса. Интерфейс приложения теперь содержит четыре основные функции: запуск воспроизведения, постановку на паузу, полную остановку и проверку текущего статуса проигрывателя.

Сообщение "Сервис подключен" подтверждает успешное установление связи между Activity и MusicService. Такая архитектура обеспечивает мгновенную реакцию на пользовательские действия, так как все команды выполняются напрямую через привязанный сервис. В отличие от Unbound Service, где взаимодействие было односторонним, Bound Service позволяет не только отправлять команды, но и получать текущее состояние проигрывателя в реальном времени (Рисунок 15).

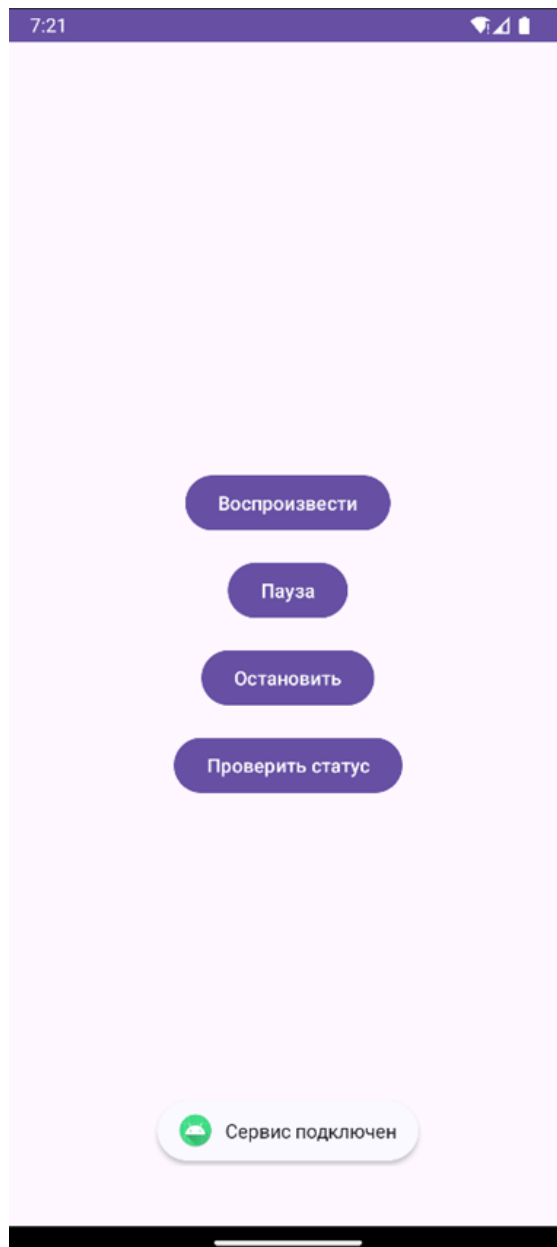


Рисунок 15 – Демонстрация приложения, подключенного к сервису

1.2 Диалоговые окна

Диалоговые окна в Android представляют собой небольшие окна, которые отображаются поверх основного содержимого приложения для передачи важного сообщения пользователю или запроса действия от пользователя. Эти окна могут использоваться для подтверждения действий, информирования о событиях, ввода данных и других задач, требующих внимания пользователя.

Среди основных типов диалоговых окон можно выделить:

- `AlertDialog`: используется для отображения предупреждений и предложения пользователю совершить выбор. Может включать кнопки для обработки действий пользователя, такие как «Да», «Нет» или «Отмена»,
- `DatePickerDialog` и `TimePickerDialog`: позволяют пользователю выбрать дату или время соответственно,
- `Custom Dialog`: позволяет создавать диалоговые окна с пользовательским макетом, что может включать любые элементы управления.

Разберем созданием каждого типа диалоговых окон.

Приведём пример создания `AlertDialog`. `AlertDialog.Builder` представляет собой мощный инструмент для создания диалоговых окон в Android-приложениях. Рассмотрим процесс построения диалога с использованием этого класса.

Инициализацию `Builder`'а выполняем, передавая контекст текущей `Activity`. Затем последовательно настраиваем основные компоненты диалога. Устанавливаем заголовок, который будет отображаться в верхней части окна, используя метод `setTitle()`. Добавляем текстовое сообщение через `setMessage()`, размещаемое под заголовком. Для привлечения внимания пользователя можно задать соответствующую иконку методом `setIcon()`, выбрав из стандартных ресурсов или собственных изображений (Рисунок 16).

```

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Создание строителя диалоговых окон
        AlertDialog.Builder builder = new
        AlertDialog.Builder(context: MainActivity.this);

        // Установка заголовка и сообщения диалогового окна
        builder.setTitle("Подтверждение");
        builder.setMessage("Вы уверены, что хотите выполнить это действие?");
        builder.setIcon(android.R.drawable.ic_dialog_alert);
    }
}

```

Рисунок 16 – Создание объекта класса AlertDialog.Builder

При настройке кнопок диалогового окна используем метод `setPositiveButton()` для создания кнопки подтверждения, куда передаём текст кнопки и объект `DialogInterface.OnClickListener` с реализацией нужной логики в методе `onClick()`. Для кнопки отмены применяем метод `setNegativeButton()` с аналогичными параметрами, где в обработчике `onClick()` обычно вызываем метод `dismiss()` у переданного объекта `DialogInterface`, чтобы закрыть диалог. Дополнительную кнопку можно добавить через метод `setNeutralButton()`, если требуется предоставить пользователю альтернативный вариант действия. Каждый обработчик получает параметры `dialog` (ссылка на само диалоговое окно) и `which` (идентификатор нажатой кнопки), что позволяет гибко управлять поведением диалога в зависимости от действий пользователя (Рисунок 17).

```

// Установка кнопки "Да" и ее обработчика
builder.setPositiveButton( text: "Да", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        // обработка подтверждения
    }
});

// Установка кнопки "Отмена" и ее обработчика
builder.setNegativeButton( text: "Отмена", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        // обработка отмены действия
        dialog.dismiss();
    }
});

```

Рисунок 17 – Добавление кнопок в диалоговое окно

При обработке нажатий в AlertDialog параметр which может содержать одно из следующих стандартных значений:

- DialogInterface.BUTTON_POSITIVE (значение -1) для положительной кнопки (подтверждение),
- DialogInterface.BUTTON_NEGATIVE (значение -2) для отрицательной кнопки (отмена),
- DialogInterface.BUTTON_NEUTRAL (значение -3) для нейтральной кнопки.

На практике проверка значения which обычно не требуется, так как каждый тип кнопки имеет свой отдельный обработчик. Однако этот параметр может быть полезен, если все кнопки обрабатываются в едином обработчике или при создании кастомных диалогов с дополнительными кнопками. В таких случаях через сравнение значения which с константами можно точно определить, какая именно кнопка была нажата.

После завершения настройки создадим экземпляр AlertDialog методом create() и отобразим его пользователю вызовом show(). Такой подход позволяет гибко конфигурировать диалоговое окно, добавляя различные элементы управления по мере необходимости (Рисунок 18).

```
// Создание и отображение AlertDialog
AlertDialog dialog = builder.create();
dialog.show();
}
```

Рисунок 18 – Создание и отображение AlertDialog

Для расширения функциональности `AlertDialog.Builder` предоставляет дополнительные методы конфигурации. Метод `setView()` позволяет добавить кастомную разметку в диалоговое окно, что дает возможность создавать сложные формы с различными элементами управления. Для работы с элементами выбора доступны методы `setSingleChoiceItems()` для радиокнопок, `setMultiChoiceItems()` для чекбоксов и `setItems()` для простых списков.

Метод `setCancelable()` определяет, можно ли закрыть диалог нажатием вне его области или кнопкой "Назад".

Посмотрим на созданное нами диалоговое окно типа `AlertDialog` (Рисунок 19).

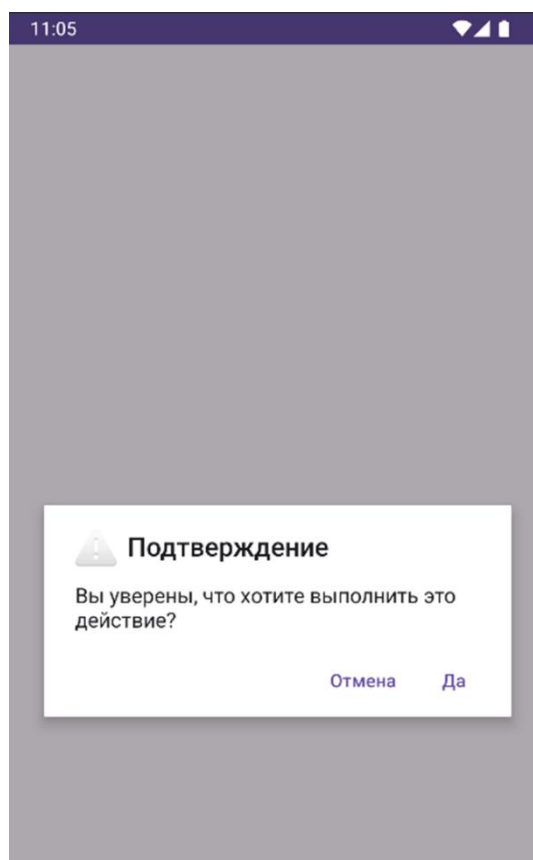


Рисунок 19 – Демонстрация AlertDialog

Класс `AlertDialog.Builder` предоставляет набор методов для настройки диалогового окна.

Основные методы конфигурации:

- `setTitle()` – задает текстовый заголовок,
- `setCustomTitle()` – устанавливает кастомное `View` для заголовка,
- `setMessage()` – добавляет текстовое сообщение,
- `setCustomMessage()` – позволяет использовать свое `View` для сообщения,
- `setIcon()` – добавляет иконку в заголовок,
- `setCancelable()` – определяет возможность отмены (по умолчанию `true`).

Методы для настройки кнопок:

- `setPositiveButton()` – добавляет кнопку подтверждения действия,
- `setNegativeButton()` – добавляет кнопку отмены действия,
- `setNeutralButton()` – добавляет дополнительную кнопку с альтернативным действием,
- `setOnCancelListener()` – устанавливает обработчик события отмены диалога,
- `setOnDismissListener()` – устанавливает обработчик события закрытия диалога.

Методы для отображения списков:

- `setItems()` – добавляет простой список элементов для выбора, где каждый элемент можно выбрать однократно. При выборе элемента диалог автоматически закрывается,
- `setSingleChoiceItems()` – реализует список с возможностью выбора только одного элемента (аналог `RadioButton`). Отображает маркер выбранного элемента и позволяет изменить выбор перед подтверждением,

- `setMultiChoiceItems()` – настраивает список с возможностью множественного выбора (аналог `CheckBox`). Позволяет выбрать несколько элементов одновременно, сохраняя состояние каждого,
- `setCursor()` – предназначен для работы с данными из базы данных через объект `Cursor`. Он позволяет отображать в диалоговом окне элементы, полученные из SQL-запроса, автоматически используя указанную текстовую колонку для вывода содержимого,
- `setAdapter()` – подключает кастомный адаптер для отображения сложных списков с индивидуальной разметкой элементов. Дает полный контроль над данными и их визуальным представлением.

Завершающие методы:

- `create()` – создает объект `AlertDialog`,
- `show()` – создает и сразу показывает диалог.

Для создания `TimePickerDialog` используется конструктор класса, а не паттерн `Builder`, в отличие от `AlertDialog`. Конструктор `TimePickerDialog` принимает следующие параметры: контекст приложения, обработчик выбора времени `OnTimeSetListener`, начальные значения часов и минут, а также флаг использования 24-часового формата.

Обработчик `OnTimeSetListener` содержит метод `onTimeSet()` с тремя параметрами: `view` (ссылка на сам `TimePicker`), `selectedHour` (выбранный час) и `selectedMinute` (выбранные минуты). Этот метод вызывается при подтверждении выбора времени пользователем.

При реализации `TimePickerDialog` в классе `MainActivity` сначала объявим поля `hour` и `minute` типа `int` для хранения времени. Установим значения по умолчанию — 12 часов и 0 минут. Эти переменные будут использоваться для начальной настройки диалога и сохранения выбранных значений.

Для создания `TimePickerDialog` вызовем его конструктор, передав следующие параметры:

- контекст текущей `Activity` (`this`),

- объект `OnTimeSetListener` с переопределенным методом `onTimeSet()`,
- начальные значения часов и минут (хранящиеся в полях `hour` и `minute`),
- флаг `is24HourView` со значением `true` для 24-часового формата.

В методе `onTimeSet()` обработчика реализуем логику обновления интерфейса: сохраняем выбранные пользователем значения часов (`selectedHour`) и минут (`selectedMinute`) в соответствующие поля класса, затем форматируем их в строку вида "ЧЧ:ММ" с добавлением ведущих нулей для однозначных значений, после чего обновляем текст в `TextView`, отображая пользователю итоговый результат выбора. Это позволяет наглядно демонстрировать установленное время и сохранять последние выбранные значения для дальнейшего использования в приложении (Рисунок 20).

```
public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private int hour = 12; // Часы по умолчанию
    private int minute = 0; // Минуты по умолчанию

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);

        TimePickerDialog timePickerDialog = new TimePickerDialog(
            context: this,
            new TimePickerDialog.OnTimeSetListener() {
                @Override
                public void onTimeSet(TimePicker view, int selectedHour, int selectedMinute) {
                    // Обновляем поля времени выбранными значениями
                    hour = selectedHour;
                    minute = selectedMinute;

                    // Отображаем выбранное время в TextView
                    String time = String.format("%02d:%02d", hour, minute);
                    textView.setText("Выбранное время: " + time);
                }
            },
            hour, minute, is24HourView: true); // true для 24-часового формата

        timePickerDialog.show();
    }
}
```

Рисунок 20 – Создание экземпляра `TimePickerDialog`

Вызовем метод `show()` диалогового окна, чтобы сразу вывести его на экран (Рисунок 21).



Рисунок 21 – Демонстрация отображения TimePickerDialog

Для создания `DatePickerDialog`, аналогично `TimePickerDialog`, используем конструктор класса, а не паттерн `Builder`. В классе `MainActivity` объявим три поля для хранения даты: `year` (год), `month` (месяц, где 0 соответствует январю) и `day` (день месяца), установив им значения по умолчанию.

Вызовем конструктор класса `DatePickerDialog`, передав следующие аргументы:

- контекст текущей `Activity` (`this`),
- объект `OnDateSetListener` с переопределенным методом `onDateSet()`,
- начальные значения года, месяца и дня (хранящиеся в полях `year`, `month` и `day`).

В обработчике `onDateSet()` реализуем логику обработки выбранной пользователем даты. При получении новых значений года (`selectedYear`), месяца (`selectedMonth`) и дня (`selectedDay`) при выборе их пользователем с помощью диалогового окна обновляем соответствующие поля класса, сохраняя выбранные параметры для дальнейшего использования в приложении. Особое внимание уделяем работе с месяцами — поскольку `DatePicker` использует нумерацию месяцев от 0 (январь) до 11 (декабрь), при отображении даты пользователю

необходимо увеличивать значение месяца на 1 для приведения к привычному формату. После создания экземпляра DatePickerDialog, вызовем его метод show() для вывода диалогового окна на экран (Рисунок 22).

```
public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private int year = 2023; // Год по умолчанию
    private int month = 0; // Месяц по умолчанию (0-11, где 0 - январь)
    private int day = 1; // День по умолчанию

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);

        // Создаем DatePickerDialog
        DatePickerDialog datePickerDialog = new DatePickerDialog(
            context: this,
            new DatePickerDialog.OnDateSetListener() {
                @Override
                public void onDateSet(DatePicker view, int selectedYear,
                                     int selectedMonth, int selectedDay) {
                    // Обновляем поля даты выбранными значениями
                    year = selectedYear;
                    month = selectedMonth;
                    day = selectedDay;

                    // Отображаем выбранную дату в TextView
                    // Месяц увеличиваем на 1, так как в DatePicker месяц идет от 0 до 11
                    String date = String.format("%02d.%02d.%d", day, month + 1, year);
                    textView.setText("Выбранная дата: " + date);
                }
            },
            year, month, day);

        datePickerDialog.show();
    }
}
```

Рисунок 22 – Создание экземпляра DatePickerDialog

Отображение диалогового окна выбора времени представлено на рисунке 23.

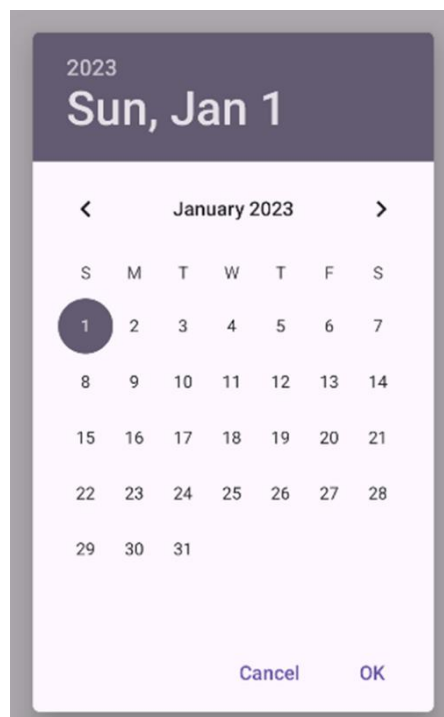


Рисунок 23 – Демонстрация отображения DatePickerDialog

Создание пользовательского диалогового окна требует особого подхода к проектированию его интерфейса. Основное отличие от стандартных диалогов заключается в необходимости самостоятельно разрабатывать макет, который будет определять внешний вид и состав элементов управления. Для реализации такого решения сначала создается отдельный XML-файл разметки, например `custom_dialog.xml`, где размещаются все необходимые компоненты пользовательского интерфейса (Рисунок 24).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="28dp"
    android:background="@android:color/white">
    <TextView
        android:id="@+id/dialog_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Это пользовательский диалог"
        android:textSize="18sp"
        android:paddingBottom="10dp" />
    <Button
        android:id="@+id/button_close"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Закрыть" />
</LinearLayout>
```

Рисунок 24 – Файл разметки пользовательского диалогового окна

В классе MainActivity реализуем отображение пользовательского диалогового окна с использованием созданного макета. Создаем экземпляр класса Dialog, передавая текущий контекст в конструктор, и устанавливаем кастомную разметку через метод setContentView(), указав идентификатор файла R.layout.custom_dialog. Для обработки нажатия кнопки закрытия находим соответствующий элемент интерфейса через findViewById() и устанавливаем обработчик события, который вызывает метод dismiss() для закрытия диалога. Завершаем настройку вызовом show(), который отображает подготовленное окно пользователю (Рисунки 25-26).

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Dialog dialog = new Dialog(context: this);
        dialog.setContentView(R.layout.custom_dialog);

        Button closeButton = dialog.findViewById(R.id.button_close);
        closeButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                dialog.dismiss();
            }
        });

        dialog.show();
    }
}
```

Рисунок 25 – Создание пользовательского диалогового окна

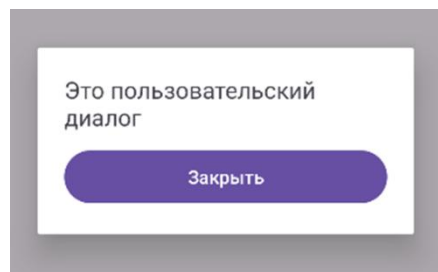


Рисунок 26 – Отображение пользовательского диалогового окна

Использование `DialogFragment` для диалоговых окон предоставляет значительные преимущества по сравнению с созданием экземпляров класса `Dialog` в `Activity`. Данный подход лучше обрабатывает поворот устройства, нажатие системной кнопки "Назад" и адаптируется под различные размеры экранов благодаря встроенной архитектуре фрагментов.

Для реализации диалогового окна необходимо создать класс, унаследованный от `DialogFragment`. В проекте следует добавить новый класс фрагмента, например `MyDialogFragment`, который будет содержать логику работы диалогового окна.

Класс фрагмента для диалогового окна сочетает стандартную функциональность фрагмента с его полным жизненным циклом и дополнительные возможности, предоставляемые базовым классом `DialogFragment`. Для создания такого диалога разработчику доступны два основных подхода:

- первый подход предполагает переопределение метода `onCreateDialog()`, который должен возвращать готовый объект `Dialog`. Этот вариант позволяет использовать стандартные диалоги `Android`, такие как `AlertDialog` или `DatePickerDialog`, с сохранением всех преимуществ `DialogFragment`,
- второй подход заключается в использовании традиционного для фрагментов метода `onCreateView()`, где определяется кастомная разметка диалогового окна. Такой способ дает полный контроль над интерфейсом и поведением диалога, позволяя реализовать любые дизайнерские решения.

Приведём пример создания диалогового окна согласно первому подходу.

В методе `onCreateDialog()` класса `MyDialogFragment` создается и настраивается стандартное диалоговое окно типа `AlertDialog`. С помощью объекта `AlertDialog.Builder` устанавливаются основные параметры: заголовок "Подтверждение" через метод `setTitle()`, текст сообщения "Вы уверены..." с

помощью `setMessage()`, иконка предупреждения из системных ресурсов через `setIcon()`.

Добавляются две кнопки взаимодействия: "Отмена" как отрицательная кнопка (`setNegativeButton()`) и "Ок" как положительная (`setPositiveButton()`). Обе кнопки пока не содержат обработчиков событий (параметр `listener` установлен в `null`). Завершающим этапом метод `create()` преобразует построитель в готовый объект `Dialog`, который возвращается как результат работы метода (Рисунок 27).

```
public class MyDialogFragment extends DialogFragment {
    @NonNull
    @Override
    public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

        builder.setTitle("Подтверждение");
        builder.setMessage("Вы уверены, что хотите выполнить это действие?");
        builder.setIcon(android.R.drawable.ic_dialog_alert);
        builder.setNegativeButton(text: "Отмена", listener: null);
        builder.setPositiveButton(text: "Ок", listener: null);

        return builder.create();
    }
}
```

Рисунок 27 – Расширение класса DialogFragment

В методе `onCreate()` класса `MainActivity` создаётся объект класса `DialogFragment`, а затем для отображения диалога используется вызов метода `show()`, в который передаются следующие аргументы:

- результат вызова `getSupportFragmentManager()`, который возвращает системный менеджер фрагментов, ответственный за их жизненный цикл внутри `Activity`,
- строковый тег "CustomDialog", выполняющий роль уникального идентификатора для последующего поиска диалогового окна и управления им.

Создание и отображение диалогового окна в MainActivity представлено на рисунке 28.

```
public class MainActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        MyDialogFragment dialogFragment = new MyDialogFragment();  
        dialogFragment.show(getSupportFragmentManager(), tag: "CustomDialog");  
    }  
}
```

Рисунок 28 – Создание и отображение экземпляра MyDialogFragment

Отображение диалогового окна, созданного с помощью расширения класса DialogFragment, представлено на рисунке .

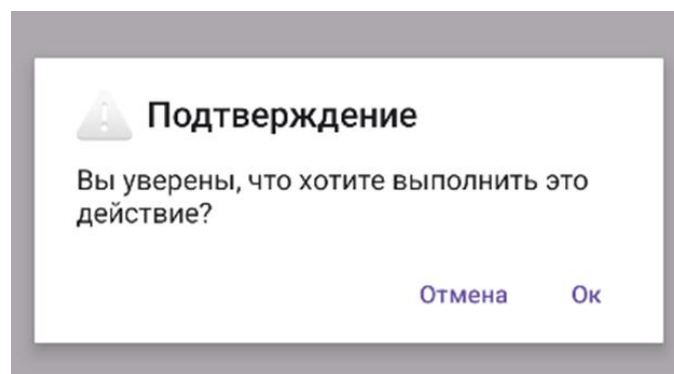


Рисунок 29 – Отображение диалогового окна, созданного с помощью расширения класса DialogFragment

Для передачи данных в диалоговое окно с целью уточнения информации, можно использовать значения, введенные пользователем в элементы интерфейса Activity. Например, при нажатии на кнопку с идентификатором buttonAlert происходит получение текста из поля ввода textName, который затем подставляется в сообщение диалогового окна, созданного с помощью AlertDialog.Builder (Рисунки 30-31).


```

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button alertButton = findViewById(R.id.buttonAlert);
        alertButton.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View view) {
                EditText nameText = findViewById(R.id.textName);
                String name = nameText.getText().toString();

                AlertDialog.Builder builder = new AlertDialog.Builder(context: MainActivity.this);
                builder.setTitle("Подтверждение");
                builder.setMessage("Вы уверены, что хотите удалить " + name + "?");
                builder.setPositiveButton(text: "Да", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        // Действие при нажатии "Да"
                    }
                });
                builder.setNegativeButton(text: "Нет", listener: null);
                builder.create().show();
            }
        });
    }
}

```

Рисунок 30 – Пример передачи данных из поля ввода в диалоговое окно

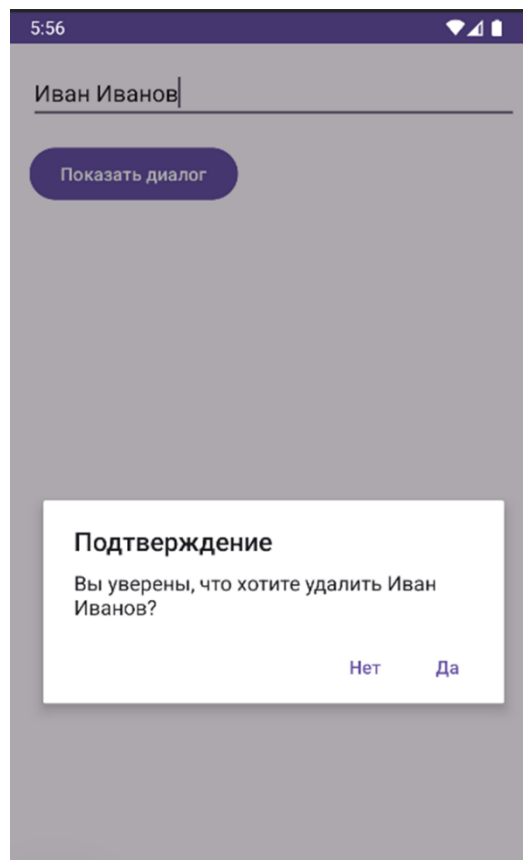


Рисунок 31 – Интерфейс подтверждения в диалоговом окне с подстановкой введенных данных

2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ

2.1 Разметка

Создадим проект Pract7 (Рисунок 32).

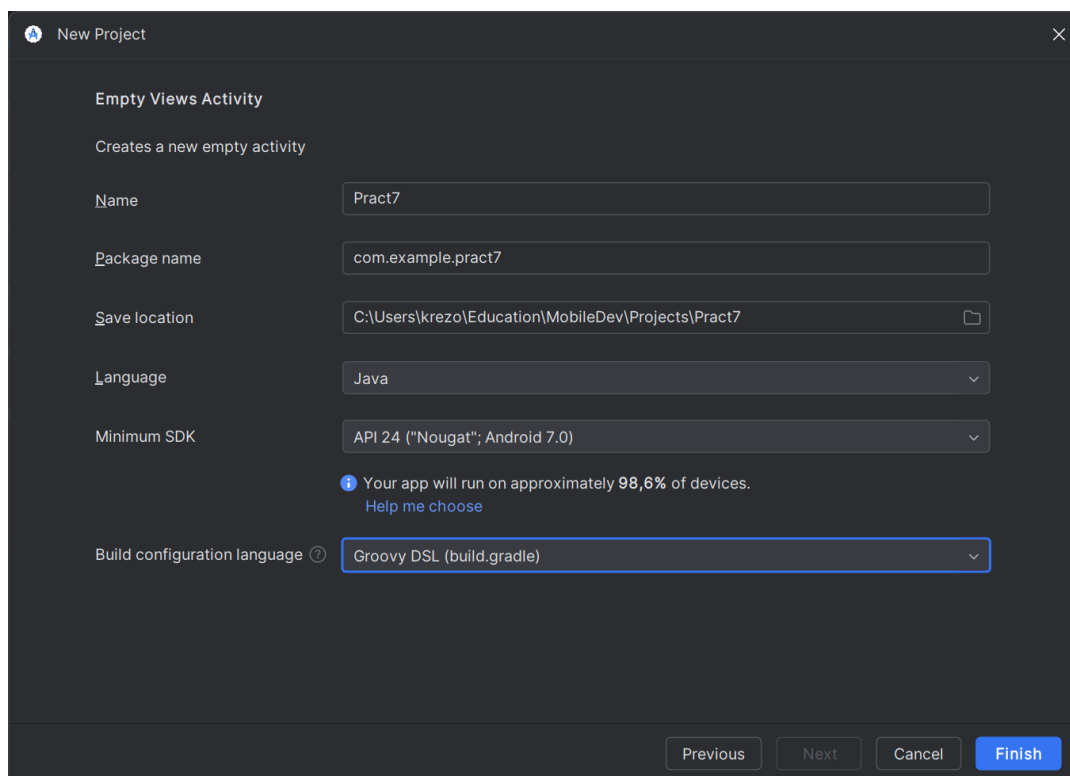


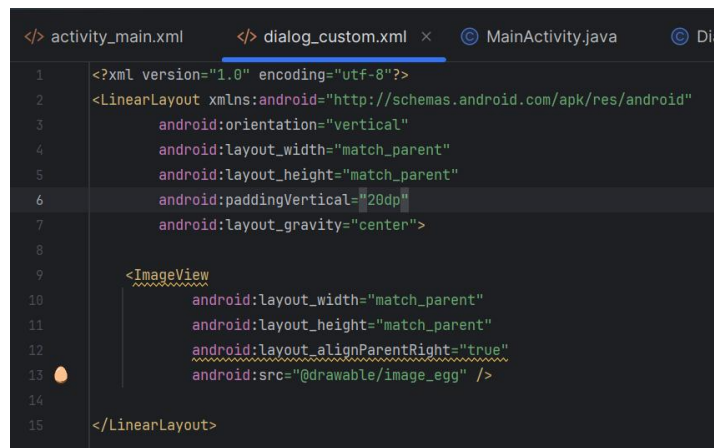
Рисунок 32 – Создание проекта

Заполним созданный файл разметки `activity_main.xml` кнопками и текстовыми полями для тестирования сервиса и диалоговых окон (Рисунки 33-34).

Рисунок 33 – Файл разметки activity_main.xml, часть 1

Рисунок 34 – Файл разметки activity_main.xml, часть 2

Также создадим файл разметки пользовательского диалога dialog_custom.xml (Рисунок 35).



```

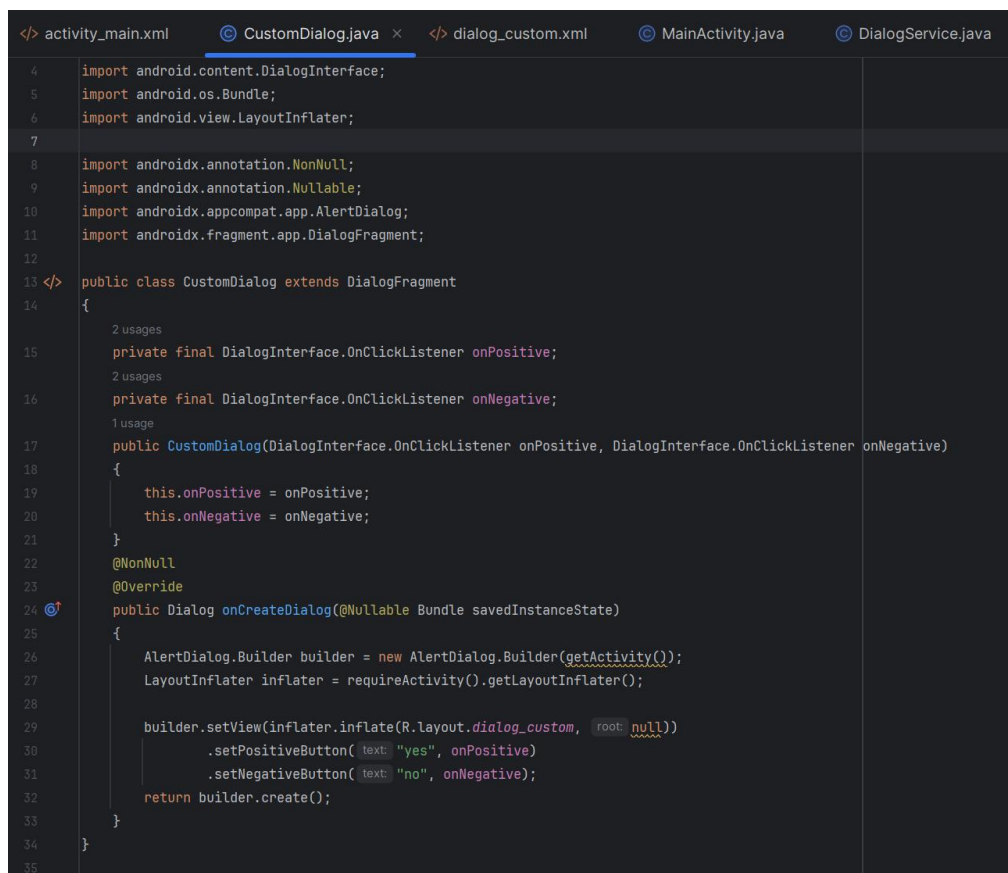
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="match_parent"
5      android:layout_height="match_parent"
6      android:paddingVertical="20dp"
7      android:layout_gravity="center">
8
9      <ImageView
10         android:layout_width="match_parent"
11         android:layout_height="match_parent"
12         android:layout_alignParentRight="true"
13         android:src="@drawable/image_egg" />
14
15  </LinearLayout>

```

Рисунок 35 – Файл разметки dialog_custom.xml

2.2 Реализация

Для начала создадим класс пользовательского диалога CustomDialog, наследуя его от DialogFragment. Реализуем в нем конструктор, принимающий слушатели нажатий на варианты ответа диалога, а также переопределим метод onCreateDialog() родительского класса (Рисунок 36).



```

4  import android.content.DialogInterface;
5  import android.os.Bundle;
6  import android.view.LayoutInflater;
7
8  import androidx.annotation.NonNull;
9  import androidx.annotation.Nullable;
10 import androidx.appcompat.app.AlertDialog;
11 import androidx.fragment.app.DialogFragment;
12
13 </> public class CustomDialog extends DialogFragment
14 {
15     2 usages
16     private final DialogInterface.OnClickListener onPositive;
17     2 usages
18     private final DialogInterface.OnClickListener onNegative;
19     1 usage
20     public CustomDialog(DialogInterface.OnClickListener onPositive, DialogInterface.OnClickListener onNegative)
21     {
22         this.onPositive = onPositive;
23         this.onNegative = onNegative;
24     }
25     @NonNull
26     @Override
27     public Dialog onCreateDialog(@Nullable Bundle savedInstanceState)
28     {
29         AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
30         LayoutInflater inflater = requireActivity().getLayoutInflater();
31
32         builder.setView(inflater.inflate(R.layout.dialog_custom, (root: null)))
33             .setPositiveButton(text: "yes", onPositive)
34             .setNegativeButton(text: "no", onNegative);
35         return builder.create();
36     }
37 }

```

Рисунок 36 – Описание класса CustomDialog

В методе onCreateDialog() создается класс AlertDialog.Builder, с помощью которого наполняется и возвращается класс диалога на основе класса AlertDialog.

Далее создадим сервис DialogService, с помощью которого и будем отображать диалоговые окна.

Сервис необходимо будет активно использовать, поэтому реализуем в нем метод onBind(), а также класс DialogBinder (Рисунок 37).

```
public class DialogService extends Service
{
    1 usage
    private final IBinder binder = new DialogBinder();

    3 usages
    public class DialogBinder extends Binder
    {
        1 usage
        DialogService getService()
        {
            return DialogService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent)
    {
        return binder;
    }
}
```

Рисунок 37 – Реализация метода onBind() класса DialogService

Затем опишем методы для создания каждого из видов диалоговых окон:

- showAlert() для создания AlertDialog;
- showTimePicker() для создания TimePickerDialog;
- showDatePicker() для создания DatePickerDialog;
- showCustom() для создания CustomDialog.

Каждый из этих методов принимает соответствующие методы-обработчики нажатия на варианты ответа в диалоговом окне (Рисунок 38).

```

1 usage
public void showAlert(Context parentContext,
                      DialogInterface.OnClickListener onPositive,
                      DialogInterface.OnClickListener onNegative)
{
    AlertDialog.Builder alertBuilder = new AlertDialog.Builder(parentContext);
    alertBuilder.setTitle("yes?")
        .setPositiveButton(text: "ok", onPositive)
        .setNegativeButton(text: "no", onNegative)
        .show();
}

1 usage
public void showTimePicker(Context parentContext, TimePickerDialog.OnTimeSetListener onPick)
{
    new TimePickerDialog(parentContext, onPick, hourOfDay: 12, minute: 0, is24HourView: true).show();
}

1 usage
public void showDatePicker(Context parentContext, DatePickerDialog.OnDateSetListener onPick)
{
    new DatePickerDialog(parentContext, onPick, year: 2025, month: 3, dayOfMonth: 25).show();
}

1 usage
public void showCustom(FragmentManager fragmentManager,
                      DialogInterface.OnClickListener onPositive,
                      DialogInterface.OnClickListener onNegative)
{
    new CustomDialog(onPositive, onNegative).show(fragmentManager, tag: "custom");
}

```

Рисунок 38 – Методы класса DialogService

Чтобы использовать данный сервис в классе MainActivity необходимо воспользоваться Binding механизмом, для чего необходимо реализовать класс ServiceConnection (Рисунок 39).

```

public class MainActivity extends AppCompatActivity
{
    5 usages
    private DialogService dialogService;
    6 usages
    private boolean isBound;
    1 usage
    private final ServiceConnection serviceConnection = new ServiceConnection()
    {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service)
        {
            DialogService.DialogBinder binder = (DialogService.DialogBinder) service;
            dialogService = binder.getService();
            isBound = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName name)
        {
            isBound = false;
        }
    };
}

```

Рисунок 39 – Описание анонимного класса ServiceConnection

На рисунке 40 показано получение сервиса DialogService с помощью метода bindService().

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    EdgeToEdge.enable( $this$enableEdgeToEdge: this);
    setContentView(R.layout.activity_main);
    ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) ->
    {
        Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars());
        v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);
        return insets;
    });

    Intent serviceIntent = new Intent( packageContext: MainActivity.this, DialogService.class);
    bindService(serviceIntent, serviceConnection, BIND_AUTO_CREATE);
}
```

Рисунок 40 – Получение сервиса DialogService

На рисунках 41-42 показан запуск диалоговых окон с помощью сервиса с получением информации из окон и применении ее к текстовым полям.

```
TextView textAlert = findViewById(R.id.text_main_alert);
Button buttonAlert = findViewById(R.id.button_main_alert);
buttonAlert.setOnClickListener((v ->
{
    if (!isBound) return;
    dialogService.showAlert(
        parentContext: MainActivity.this,
        (dialog, which) -> textAlert.setText("ok"),
        (dialog, which) -> textAlert.setText("no"));
}));

TextView textTime = findViewById(R.id.text_main_time);
Button buttonTime = findViewById(R.id.button_main_time);
buttonTime.setOnClickListener((v ->
{
    if (!isBound) return;
    dialogService.showTimePicker( parentContext: MainActivity.this,
        (view, hourOfDay, minute) ->
        {
            textTime.setText(String.format("%02d:%02d", hourOfDay, minute));
        });
}));
});
```

Рисунок 41 – Описание метода onCreate() класса MainActivity, часть 1

```

TextView textDate = findViewById(R.id.text_main_date);
Button buttonDate = findViewById(R.id.button_main_date);
buttonDate.setOnClickListener((v ->
{
    if (!isBound) return;
    dialogService.showDatePicker( parentContext: MainActivity.this,
        (view, year, month, dayOfMonth) ->
        {
            textDate.setText(String.format("%d.%02d.%02d", year, month + 1, dayOfMonth));
        });
});
TextView textCustom = findViewById(R.id.text_main_custom);
Button buttonCustom = findViewById(R.id.button_main_custom);
buttonCustom.setOnClickListener((v ->
{
    if (!isBound) return;
    dialogService.showCustom(
        getSupportFragmentManager(),
        (dialog, which) -> textCustom.setText("yes"),
        (dialog, which) -> textCustom.setText("no"));
});
});

```

Рисунок 42 – Описание метода onCreate() класса MainActivity, часть 2

2.3 Тестирование

Откроем приложение (Рисунок 43).

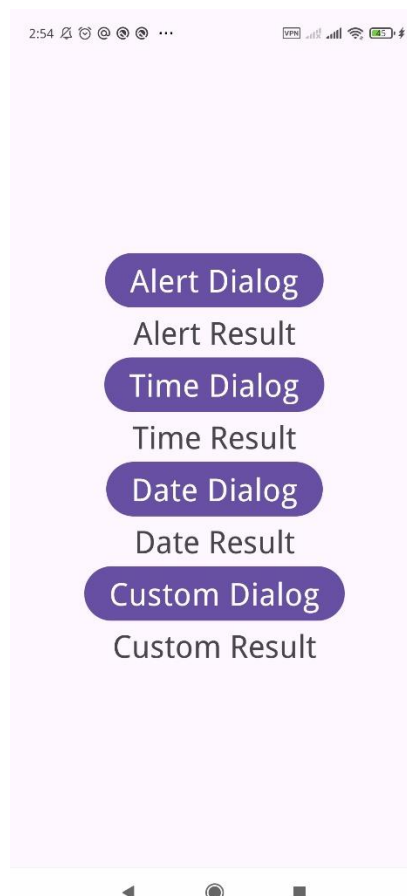


Рисунок 43 – Начальный экран приложения

После нажатия на кнопку “Alert Dialog” открылось диалоговое окно с двумя вариантами ответа (Рисунок 44).

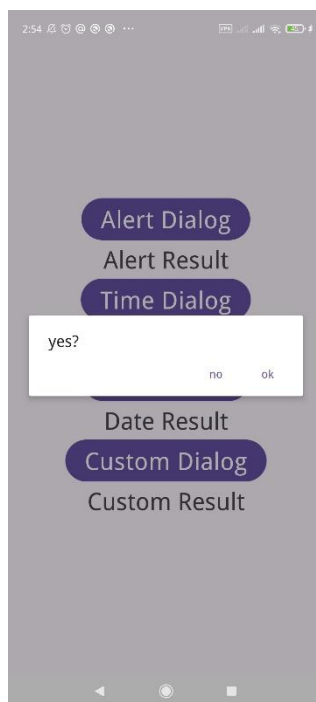


Рисунок 44 – Диалоговое окно AlertDialog

После нажатия на кнопку “Time Dialog” открылось диалоговое окно выбора времени (Рисунок 45).

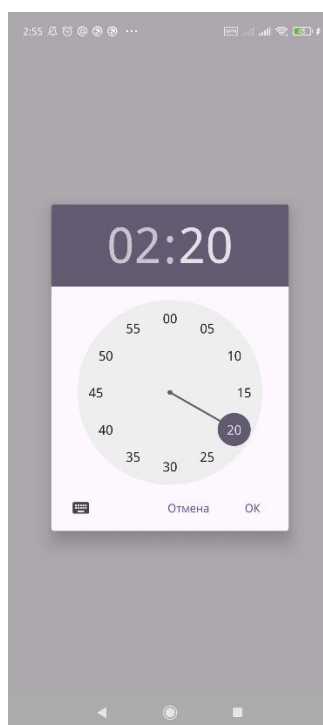


Рисунок 45 – Диалоговое окно TimePickerDialog

После нажатия на кнопку “Date Dialog” открылось диалоговое окно выбора даты (Рисунок 46).

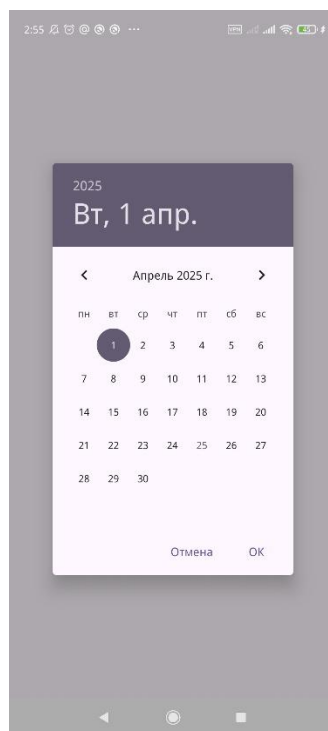


Рисунок 46 – Диалоговое окно DatePickerDialog

После нажатия на кнопку “Custom Dialog” открылось пользовательское диалоговое окно (Рисунок 47).



Рисунок 47 – Диалоговое окно CustomDialog

После взаимодействия с каждым диалоговым окном соответствующее текстовое поле изменилось в соответствии с выбранным вариантом в каждом из диалогов (Рисунок 48).

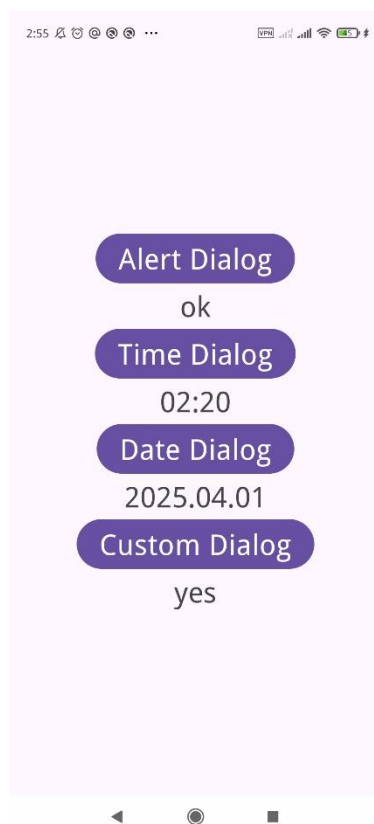


Рисунок 48 – Экран приложения после изменений
Тестирование прошло успешно.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы было изучено использование различных типов диалоговых окон в Android-приложениях. Было проведено знакомство с базовыми диалогами (AlertDialog), специализированными элементами выбора даты и времени (DatePickerDialog и TimePickerDialog), а также реализовали собственный тип диалогового окна с пользовательской разметкой. Особое внимание было уделено механизмам передачи данных между диалоговыми окнами и основной активностью. Также были изучены принципы работы Service — важного компонента Android для выполнения фоновых операций. Был создан и запущен собственный сервис.