|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |
| **Институт кибербезопасности и цифровых технологий** |

Кафедра КБ-14 «Цифровые технологии обработки данных»

**ОТЧЕТ**

**Практическая работа 4**

**по дисциплине «Разработка мобильных компонент анализа безопасности информационно-аналитических систем»**

Выполнил:

Студент группы БСБО-07-22

Сладкина Анастасия Андреевна

**Москва, 2025 г**

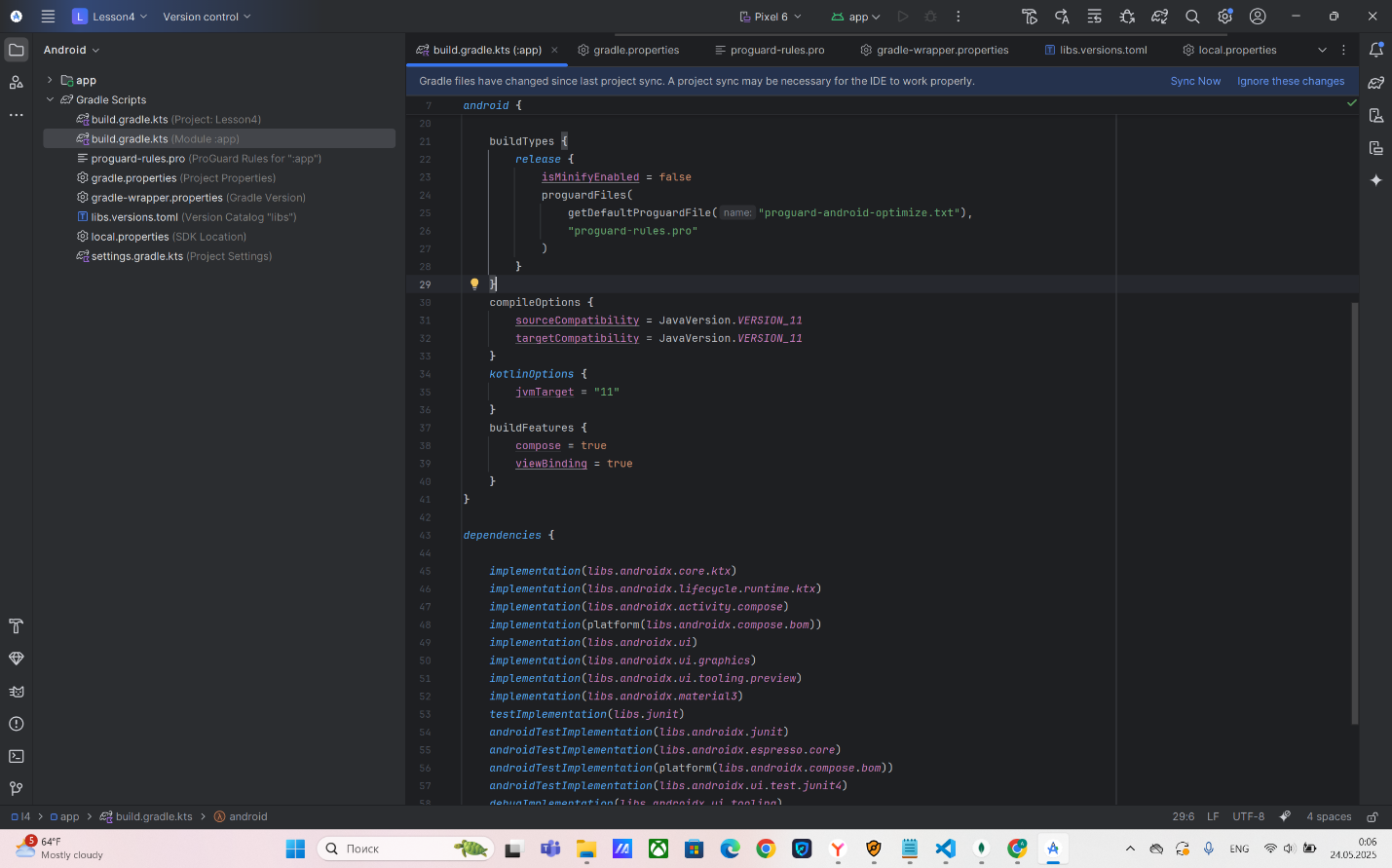
«ViewBinding» позволяет сокращать объем кода для взаимодействия с «view». При активированном «ViewBinding» генерируется «binding» классы для каждого файла разметки («layout»). Объект сгенерированного «binding» класса содержит ссылки на все графические компоненты с указанным «android:id» из файла разметки. Подклюим это свойства в наш проект (см. рис. 1). 

Рисунок 1. Подключение binding

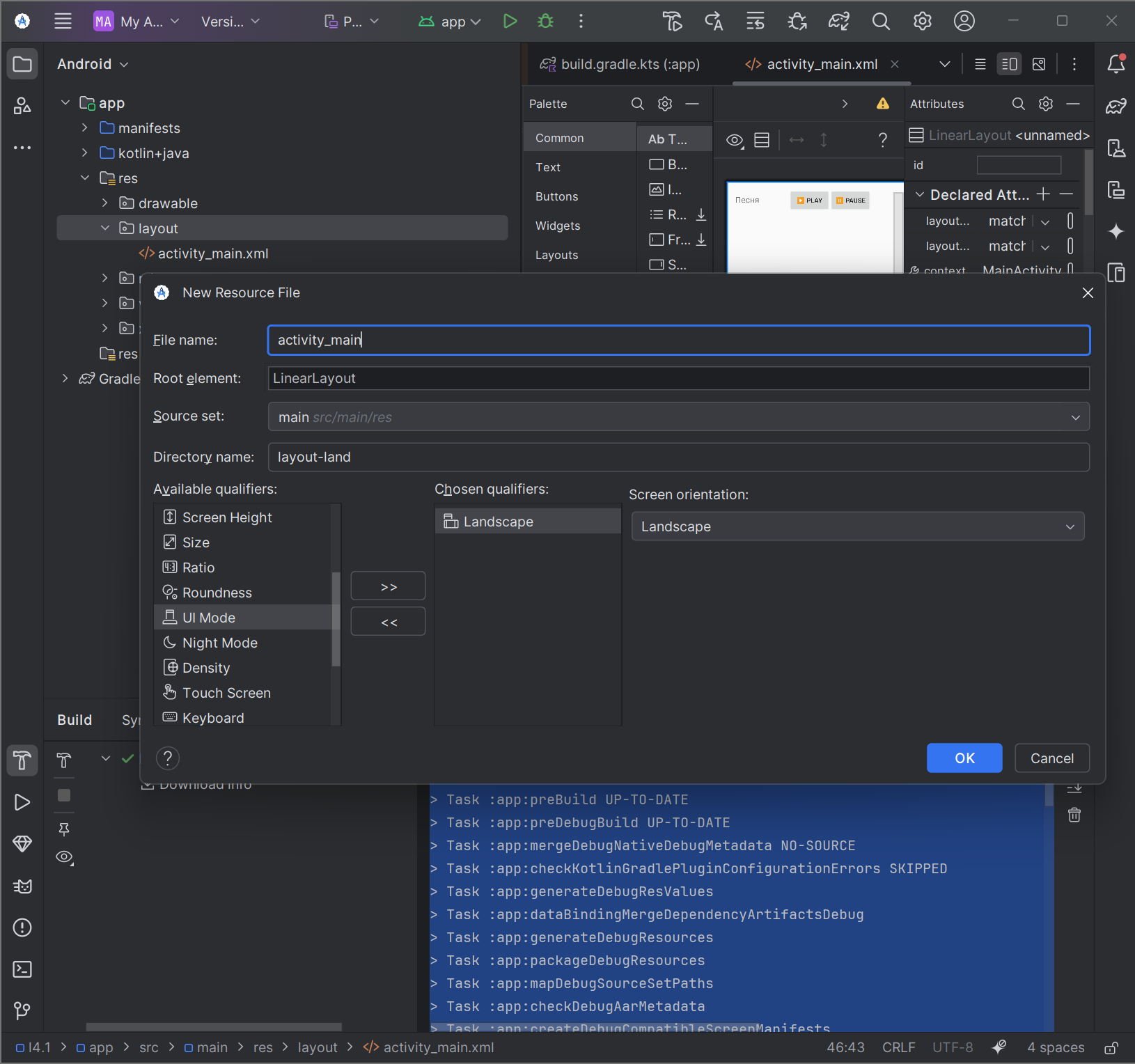


Рисунок 2. Создание экрана музыкального плеера с использованием «binding» для горизонтальной и портретной ориентации

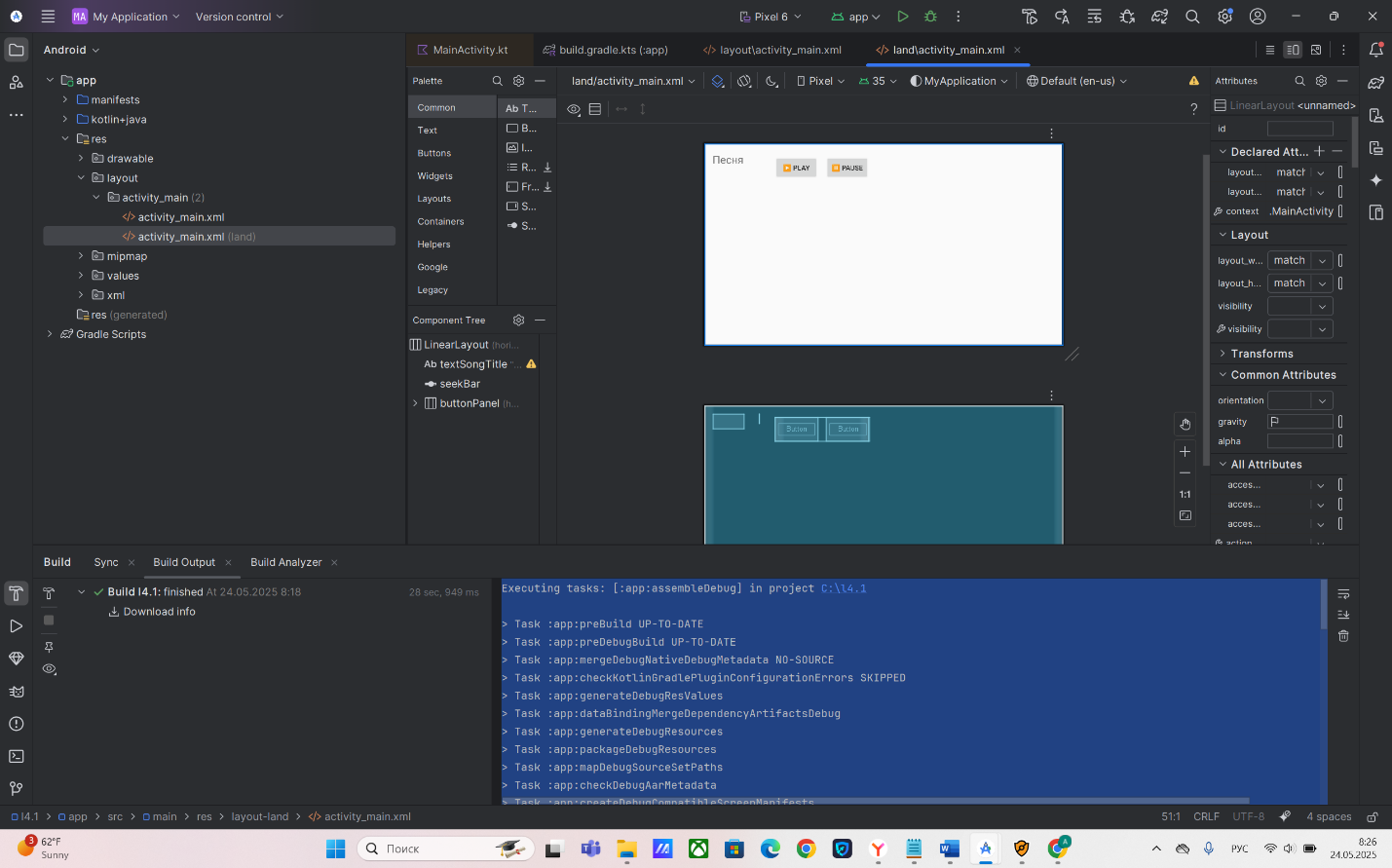


Рисунок 3. Результат горизонтальный экран

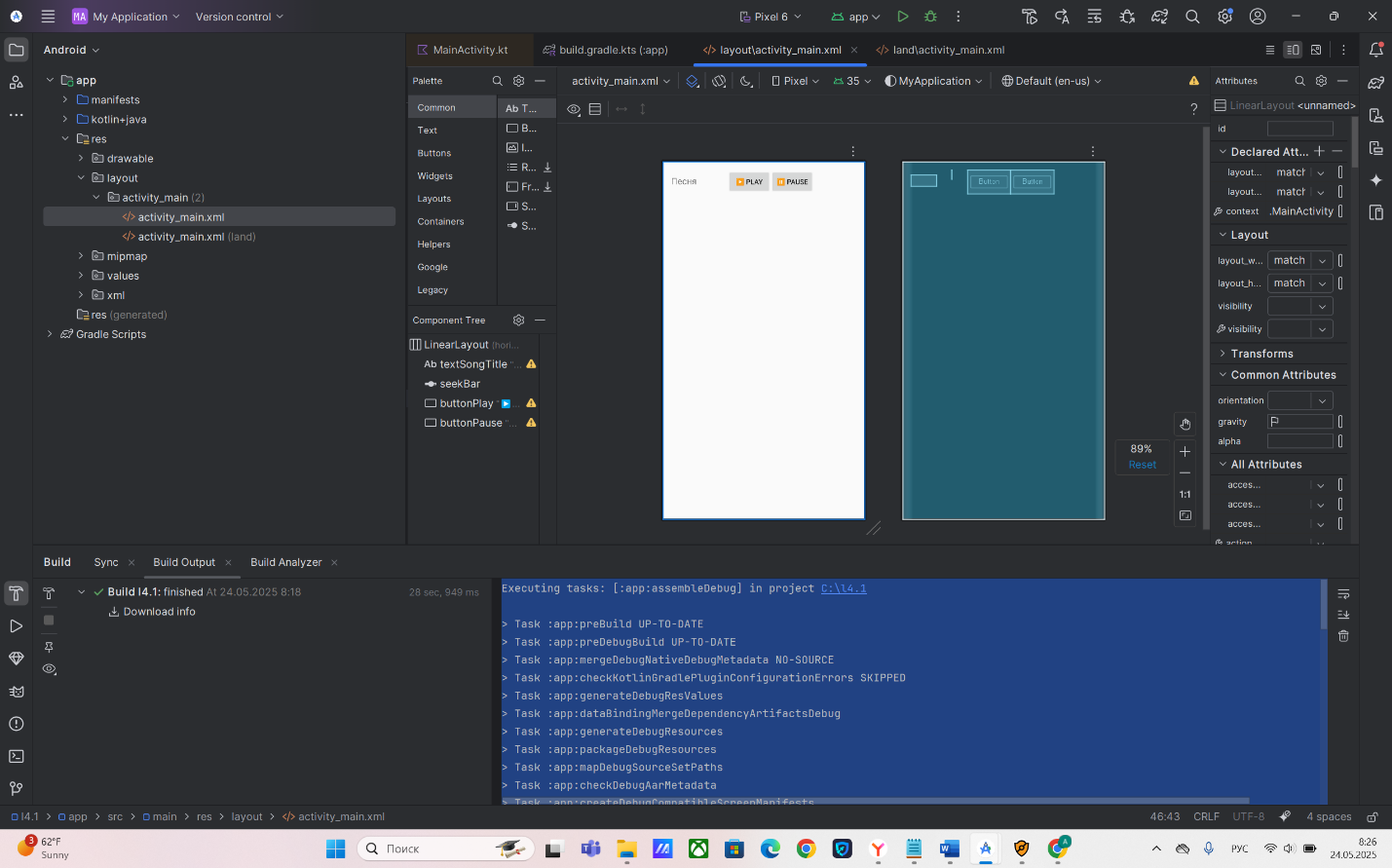


Рисунок 4. Результат вертикальный экран

Класс MainActivity наследуется от AppCompatActivity и содержит приватную переменную binding типа ActivityMainBinding, которая инициализируется в методе onCreate. В этом методе происходит привязка XML-макета через ActivityMainBinding.inflate(layoutInflater), а затем установка содержимого окна с помощью setContentView(binding!!.root). Далее, у текстового поля textSongTitle устанавливается начальное значение "My Favorite Song". Затем задаются обработчики нажатий для двух кнопок: buttonPlay и buttonPause. При нажатии на кнопку Play в лог выводится сообщение "Play clicked" и текст заголовка песни меняется на "Playing...". При нажатии на кнопку Pause аналогично выводится сообщение "Pause clicked" и текст меняется на "Paused." (см. рис. 5).

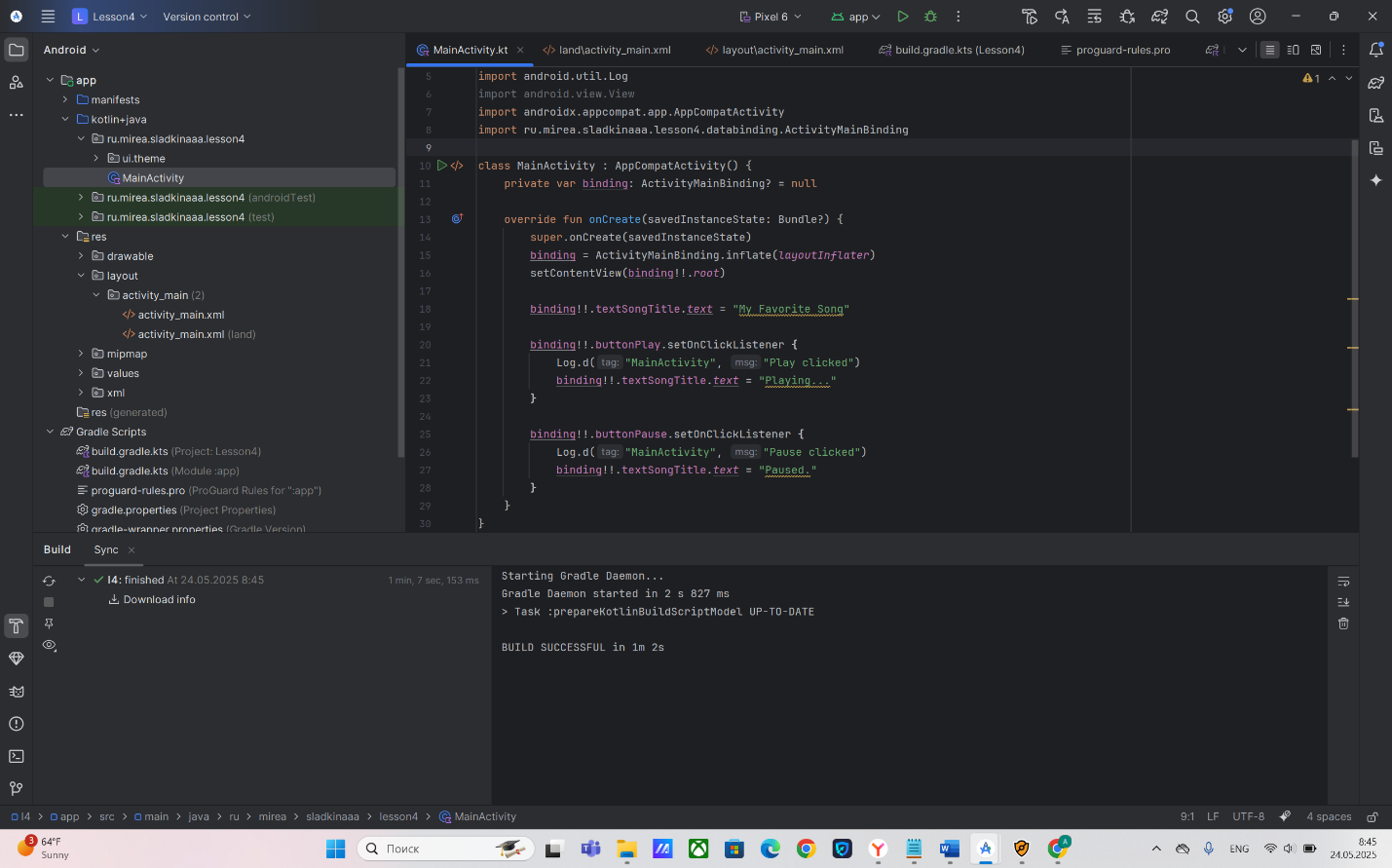


Рисунок 5. Использование binding в классе MainActivity

Далее посчитаем в фоновом потоке среднее количество пар в день за период одного месяца (см. рис. 6). Общее количество пар и учебных дней вводятся в главном экране. Отобразим результат в «TextView».

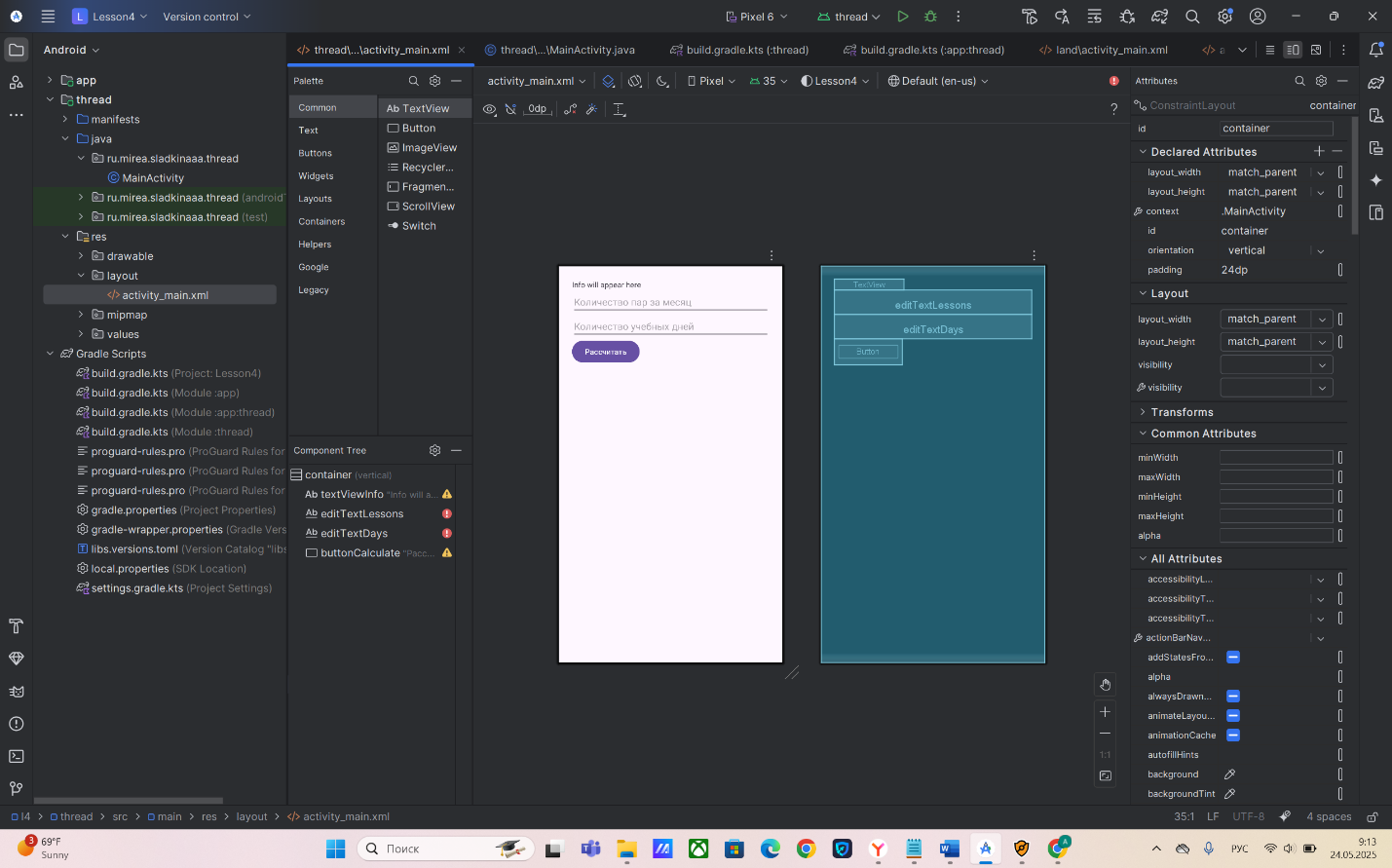


Рисунок 6. Экран подсчёта

В методе onCreate() происходит инициализация привязки: макет ActivityMainBinding раздувается, а затем устанавливается как корневой через setContentView() (см. рис. 7). Далее создаётся объект текущего потока mainThread с помощью Thread.currentThread(), и в textViewInfo выводится имя этого потока. Затем имя потока изменяется методом setName() и снова отображается в textViewInfo с добавлением информации о новом имени потока. Также в лог пишется имя текущего класса и стек вызовов. Кнопка buttonCalculate получает обработчик нажатия, в котором из текстовых полей editTextLessons и editTextDays извлекаются значения и преобразуются в целые числа (lessons и days). Затем создаётся новый поток backgroundThread, внутри которого вычисляется среднее количество пар в день как lessons / days. После этого результат отображается в textViewInfo через runOnUiThread(), чтобы обновление UI происходило в главном потоке. Поток запускается методом start().

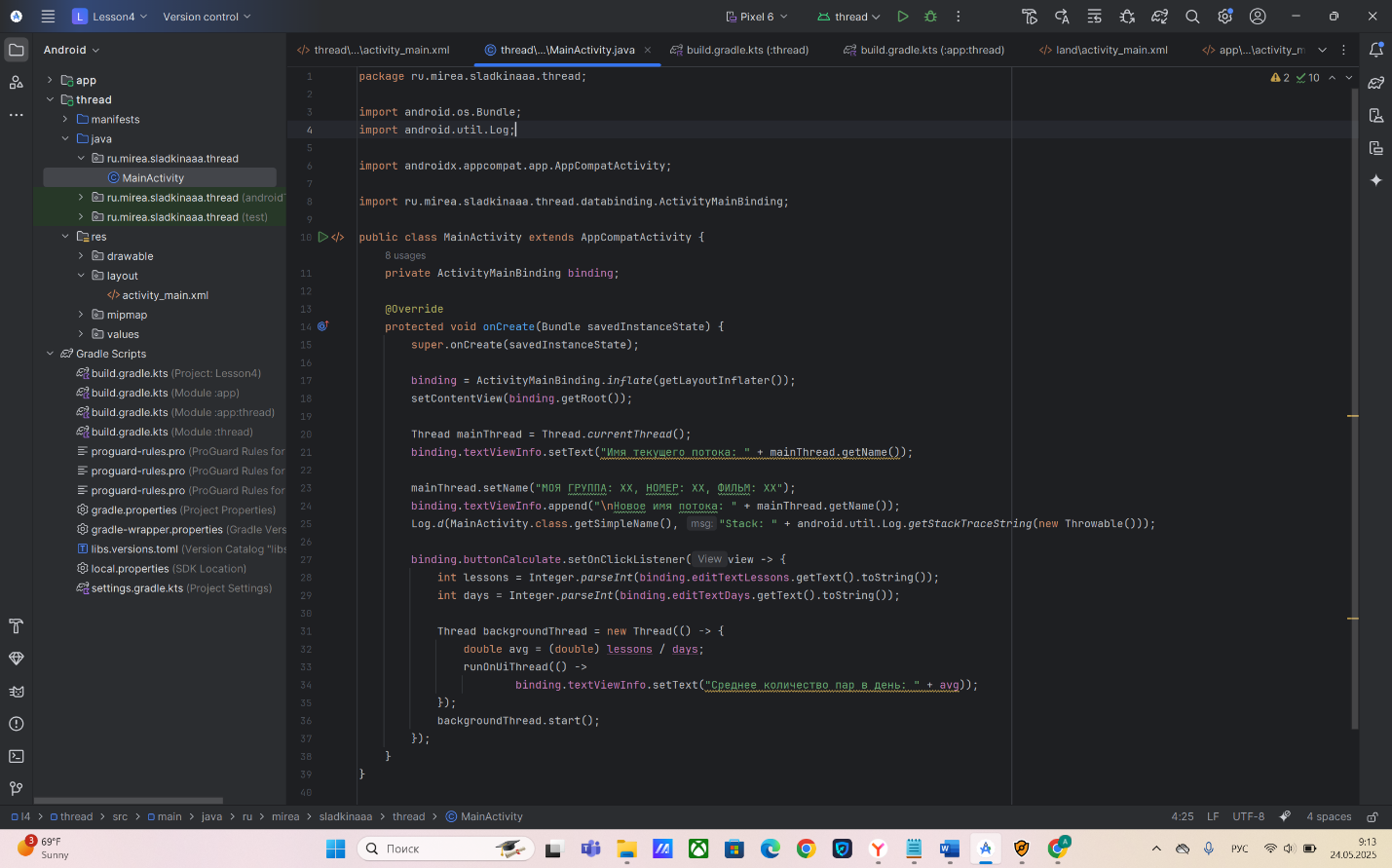


Рисунок 7. Реализация подсчёта

При нажатии кнопки сначала выполняется log("Кнопка нажата. Начало последовательности") в UI-потоке, так как обработчик клика работает в нём же, и сообщение сразу отображается в TextView и логах (см. рис. 8). Далее handler.post() асинхронно добавляет Runnable в очередь UI-потока, поэтому сообщение *"1. handler.post() → выполняется сразу в очереди UI потока"* появится чуть позже, но в том же потоке. Затем handler.postDelayed() планирует выполнение другого Runnable с задержкой в 2 секунды, и соответствующее сообщение *"2. handler.postDelayed() → выполняется с задержкой 2 сек"* отобразится через указанное время, также в UI-потоке. Метод runOnUiThread() аналогично помещает задачу в очередь UI-потока, но часто воспринимается как более явный и приоритетный способ, поэтому сообщение *"3. runOnUiThread() → помещено в UI очередь (приоритетно)"* будет выполнено после текущего кода. Наконец, log("Код вне потоков выполнен сразу") срабатывает мгновенно, так как находится непосредственно в обработчике и не зависит от асинхронных задач.

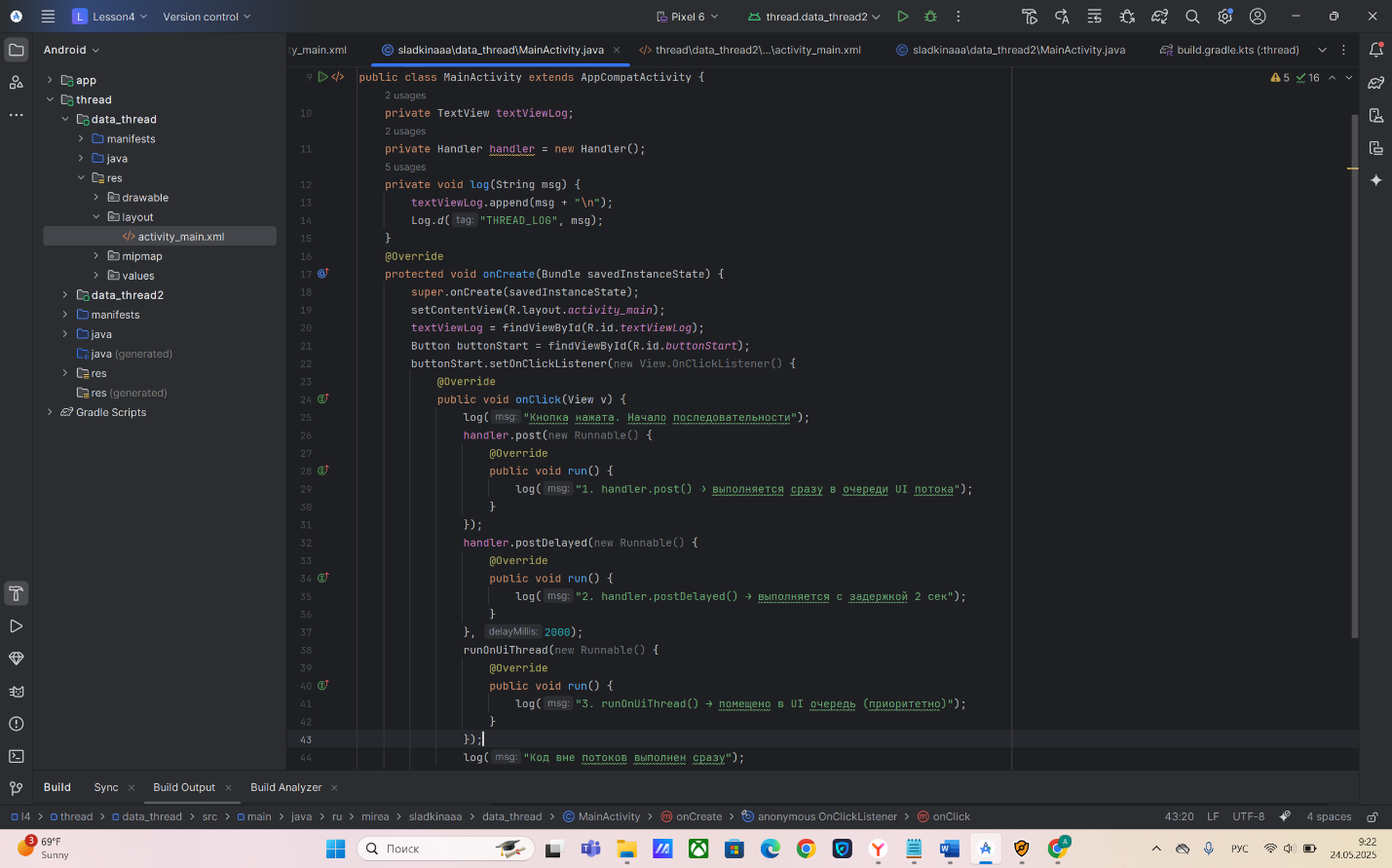


Рисунок 8. Изучение методов

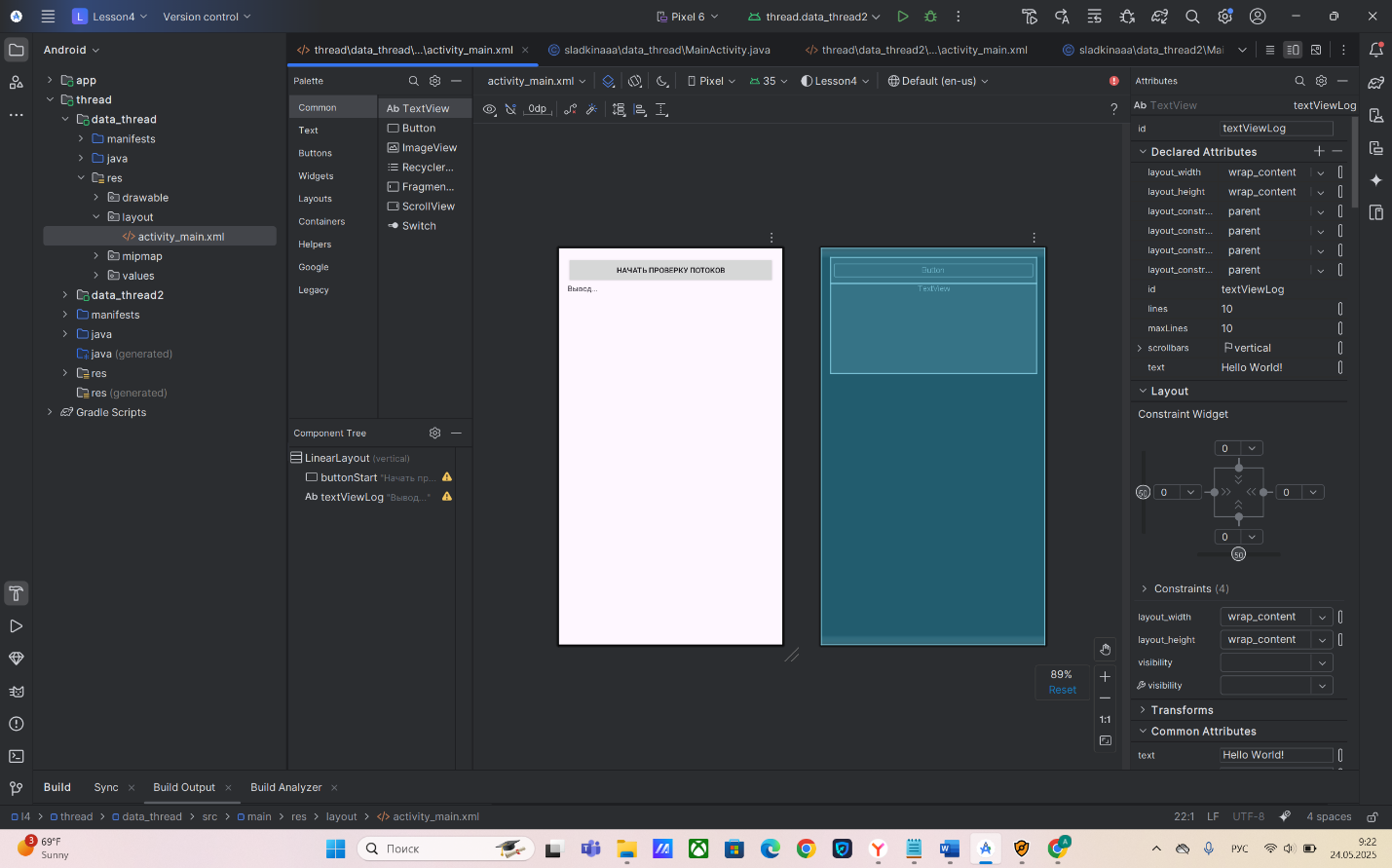


Рисунок 9. Экран проверки потоков

Для организации очереди, требуется воспользоваться классами «Handler» и «Looper»: первый отвечает за организацию очереди, а второй в бесконечном цикле проверяет, нет ли в ней новых задач для потока. Поток работает, пока связанный с ним «Looper» не будет остановлен. Следует реализовать пример, в котором вводится возраст и где работаете (см. рис. 10). Количество лет соответствует времени задержки. Результат вычисления осуществлять через Log.d.Начало формы

Конец формы

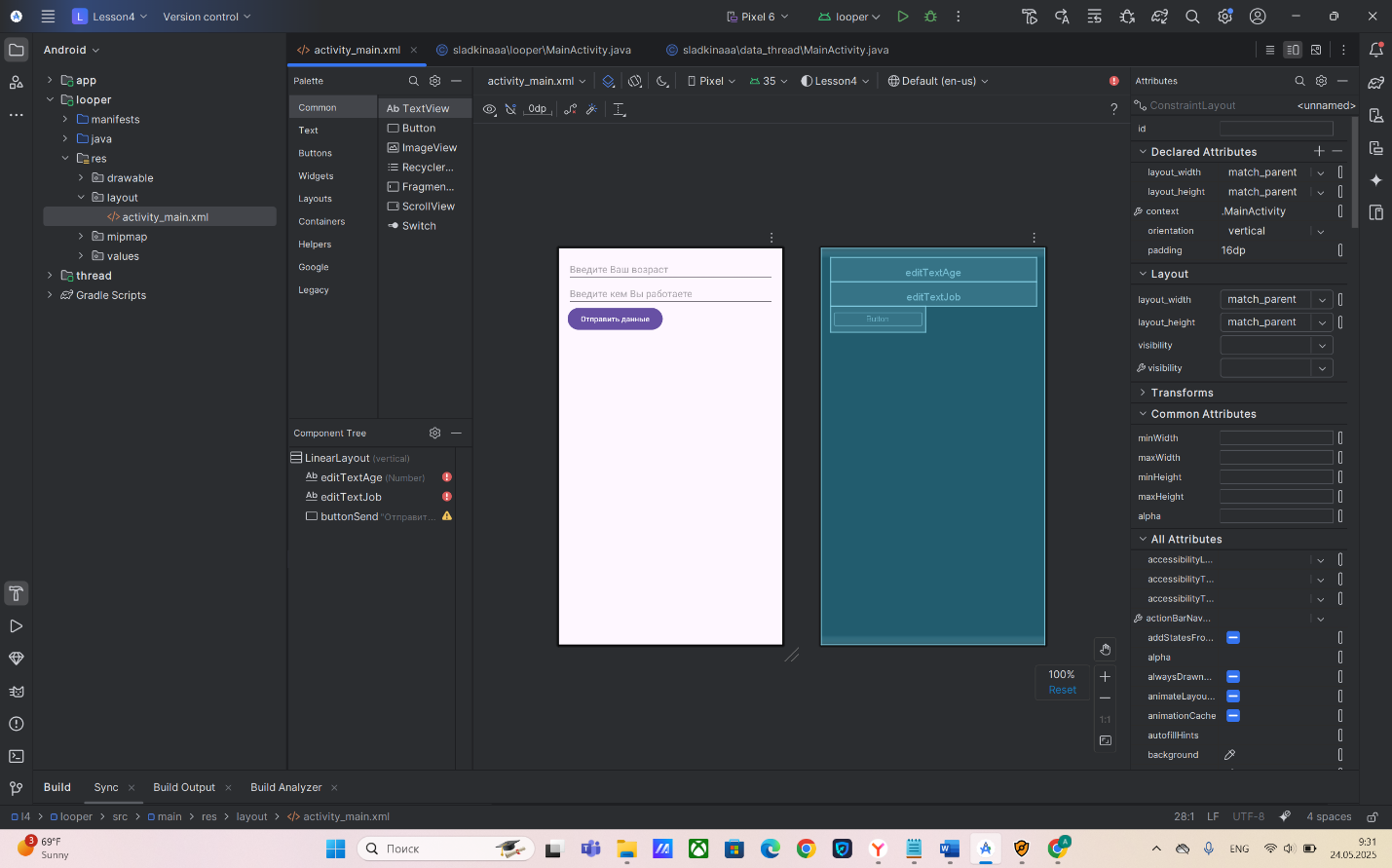


Рисунок 10. Экран перед выводом

Рассмотрим код, реализующий пользовательский класс MyLooper, который расширяет функциональность потока Thread для работы с Looper и Handler (см. рис. 11). При запуске потока run() сначала вызывается Looper.prepare(), что инициализирует Looper для текущего потока, позволяя ему обрабатывать сообщения в очереди. Затем создается Handler, ассоциированный с этим Looper, который переопределяет метод handleMessage() для обработки входящих сообщений.

Когда Handler получает сообщение Message, он извлекает из него данные Bundle, включая возраст age и профессию job. Далее поток искусственно приостанавливается на время, пропорциональное возрасту Thread.sleep(age \* 1000L). После завершения задержки в лог Log.d выводится информация о возрасте и профессии с пометкой "обработка завершена".

Цикл Looper.loop() запускает бесконечную обработку сообщений в потоке, пока Looper не будет остановлен. Поэтому, MyLooper представляет собой фоновый поток с собственной очередью сообщений, способный асинхронно выполнять задачи с задержкой, зависящей от переданных данных.

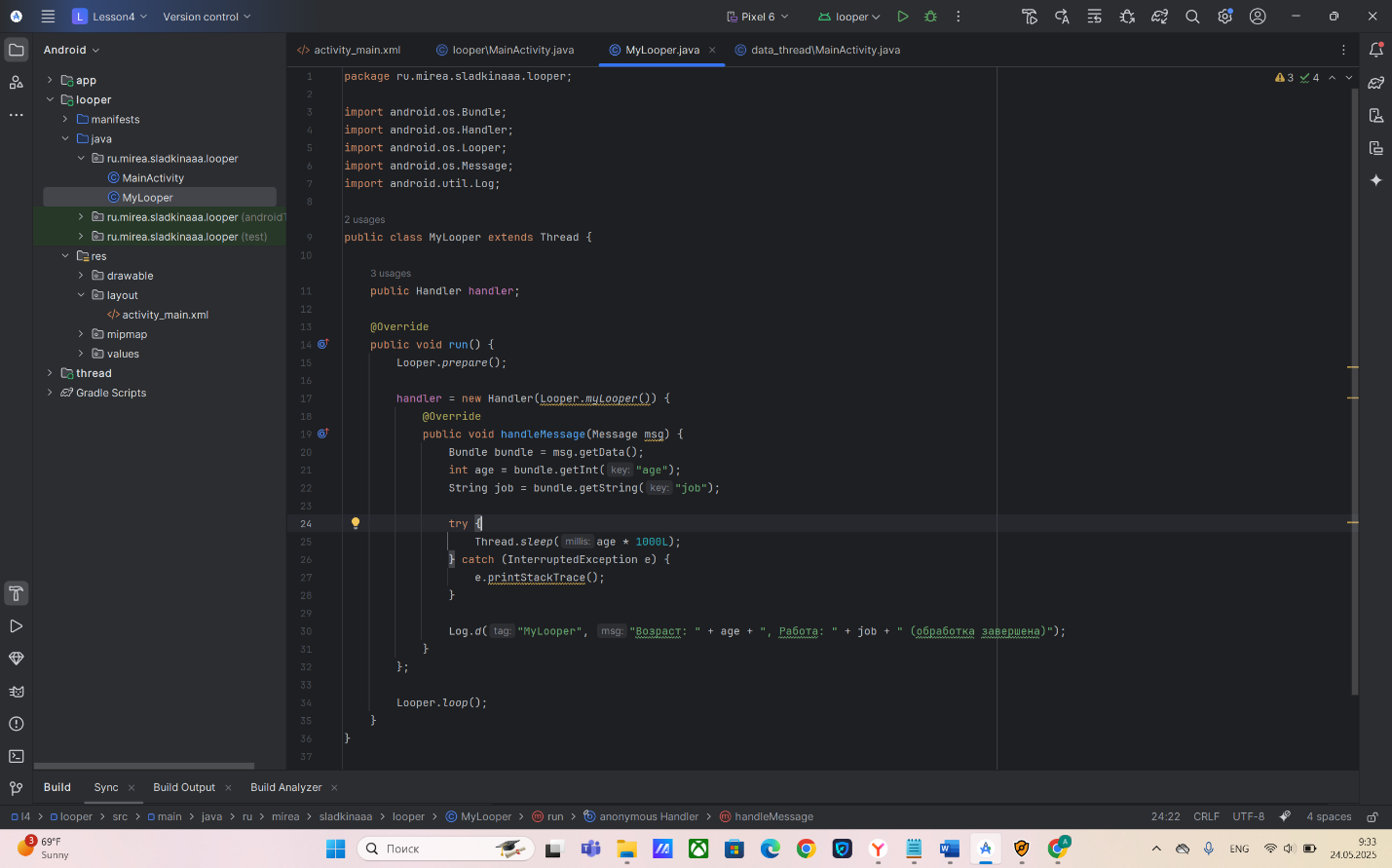


Рисунок 11. Класс MyLooper

Далее MainActivity, который взаимодействует с пользовательским MyLooper (см. рис. 12). При запуске активности onCreate инициализируются элементы интерфейса: два поля ввода (editTextAge для возраста и editTextJob для профессии) и кнопка buttonSend. Затем создается и запускается экземпляр MyLooper, который начинает работать в фоновом режиме.

При нажатии кнопки buttonSend проверяется, заполнены ли оба поля. Если какое-то из них пустое, выводится всплывающее уведомление с требованием ввести данные. Если возраст не является положительным числом или содержит недопустимые символы, также отображается соответствующее сообщение об ошибке.

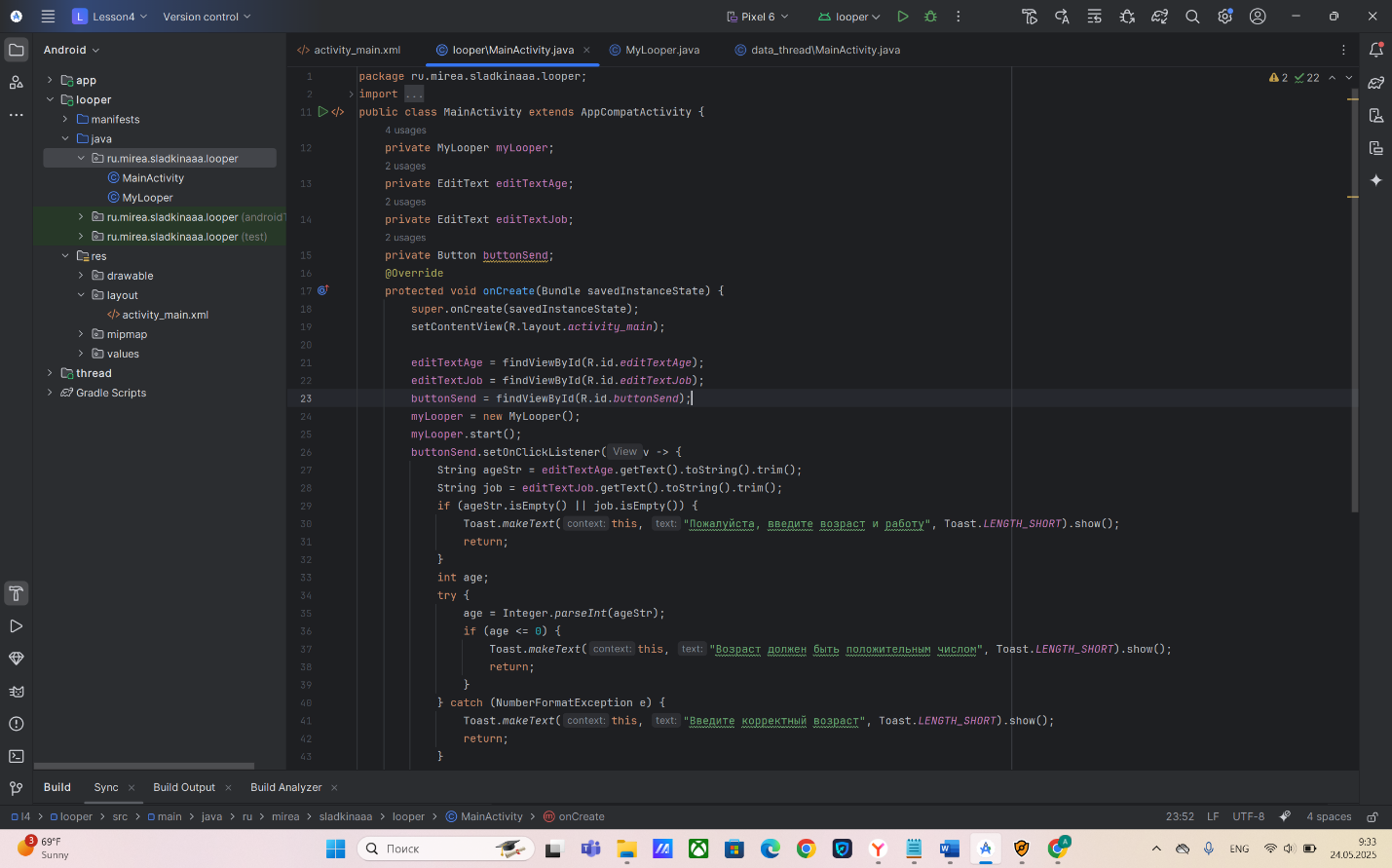


Рисунок 12. Класс MainActivity

После проверки валидности введённых данных (возраст должен быть положительным числом, а поле профессии не должно быть пустым) происходит следующее: если обработчик myLooper.handler инициализирован, создаётся сообщение с данными, упакованными в Bundle возраст age и профессия job (см. рис. 13). Это сообщение отправляется в фоновый поток через myLooper.handler.sendMessage(msg), где оно будет обработано с задержкой, пропорциональной возрасту (как было показано в предыдущих скринах).

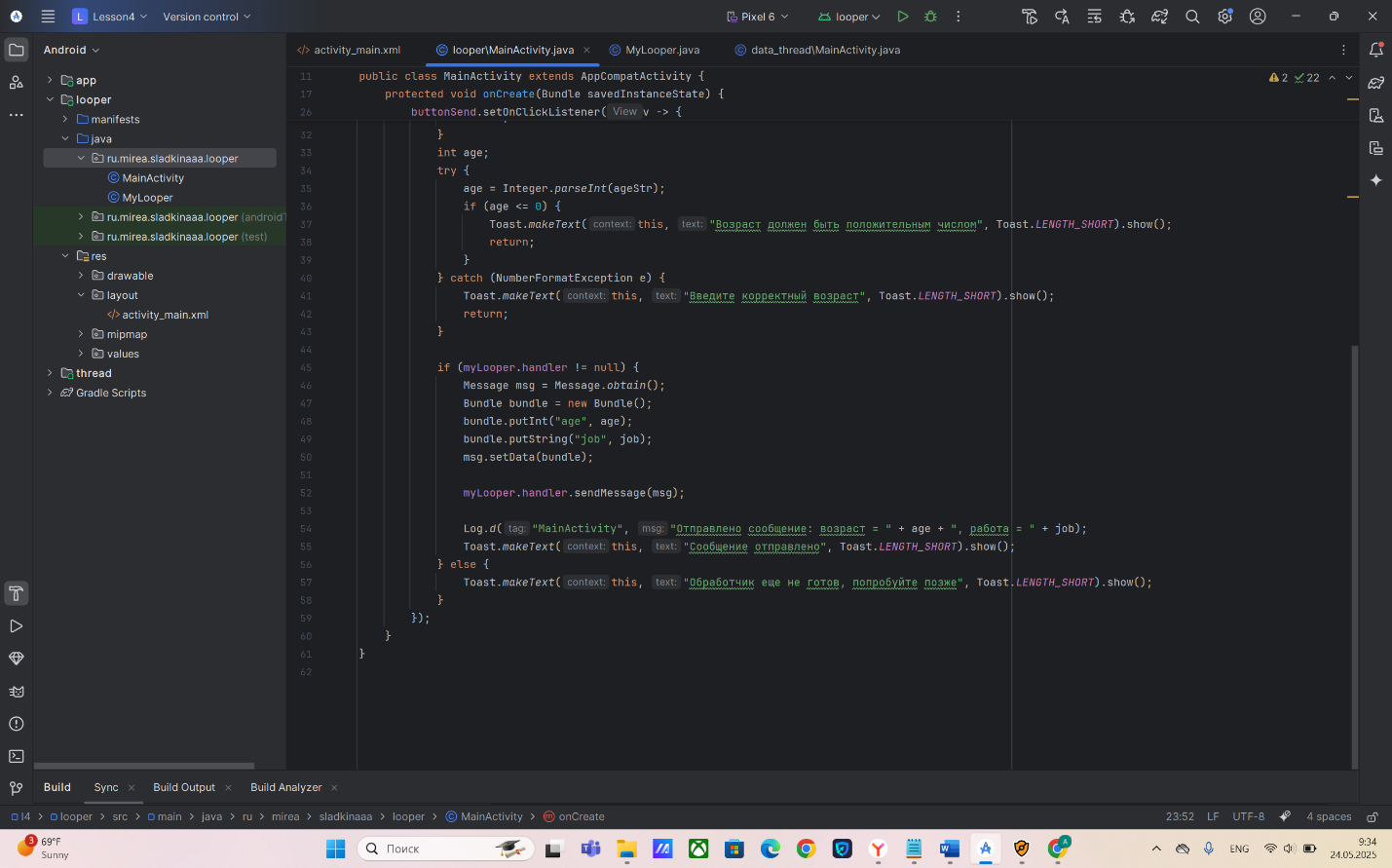


Рисунок 13. Класс MainActivity продолжение

В следующем модуле были добавлены элементы «EditText» и «Button» (см. рис. 14). Пользователь вводит фразу в «EditText», далее она шифруется с помощью алгоритма AES и передается вместе с ключом в «Loader». В «Loader» происходит дешифровка фразы и последующая передача в «MainActivity». Дешифрованная фраза отображается с помощью «toast» или «snackBar».

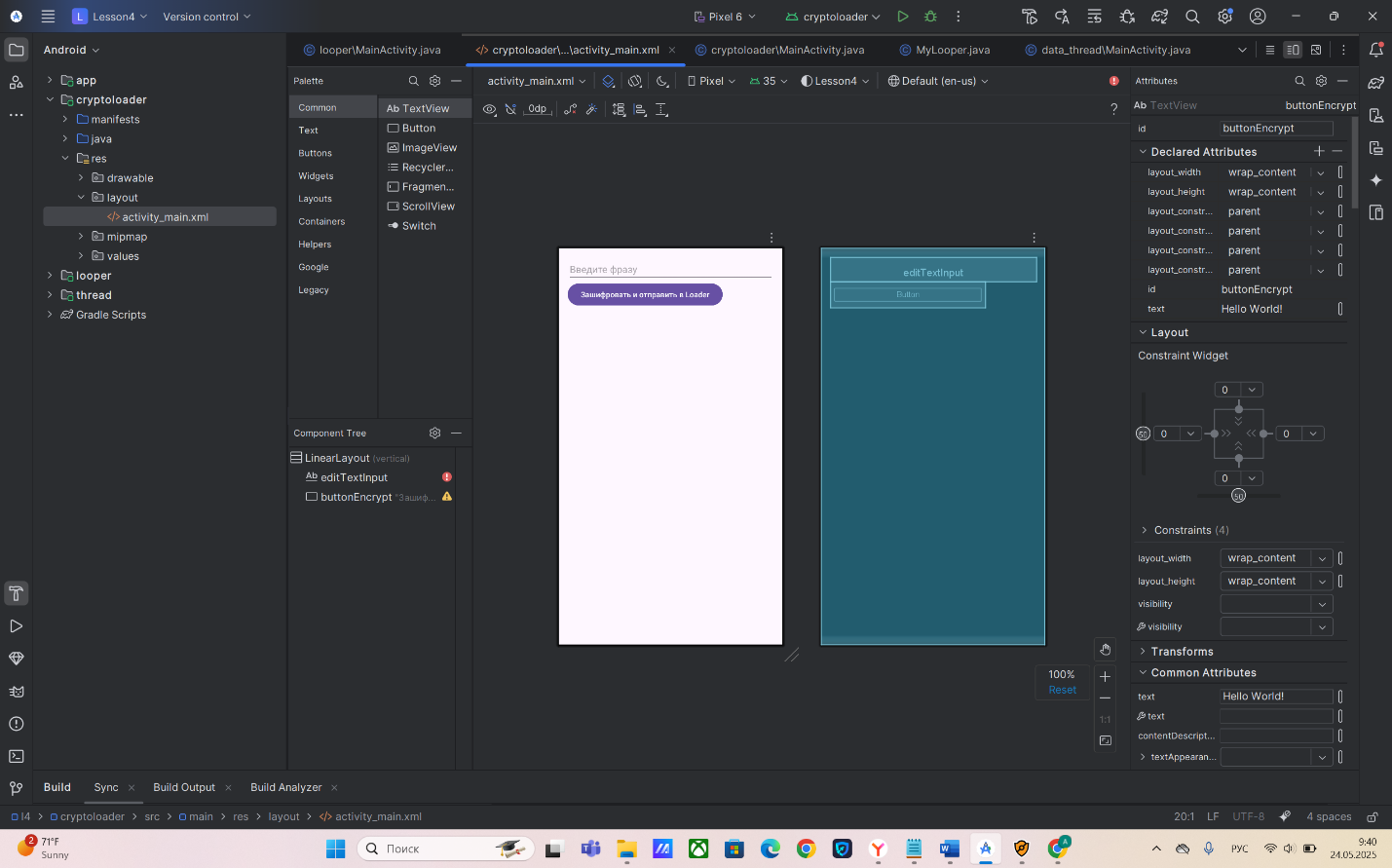


Рисунок 14. Экран ввода данных с кнопкой

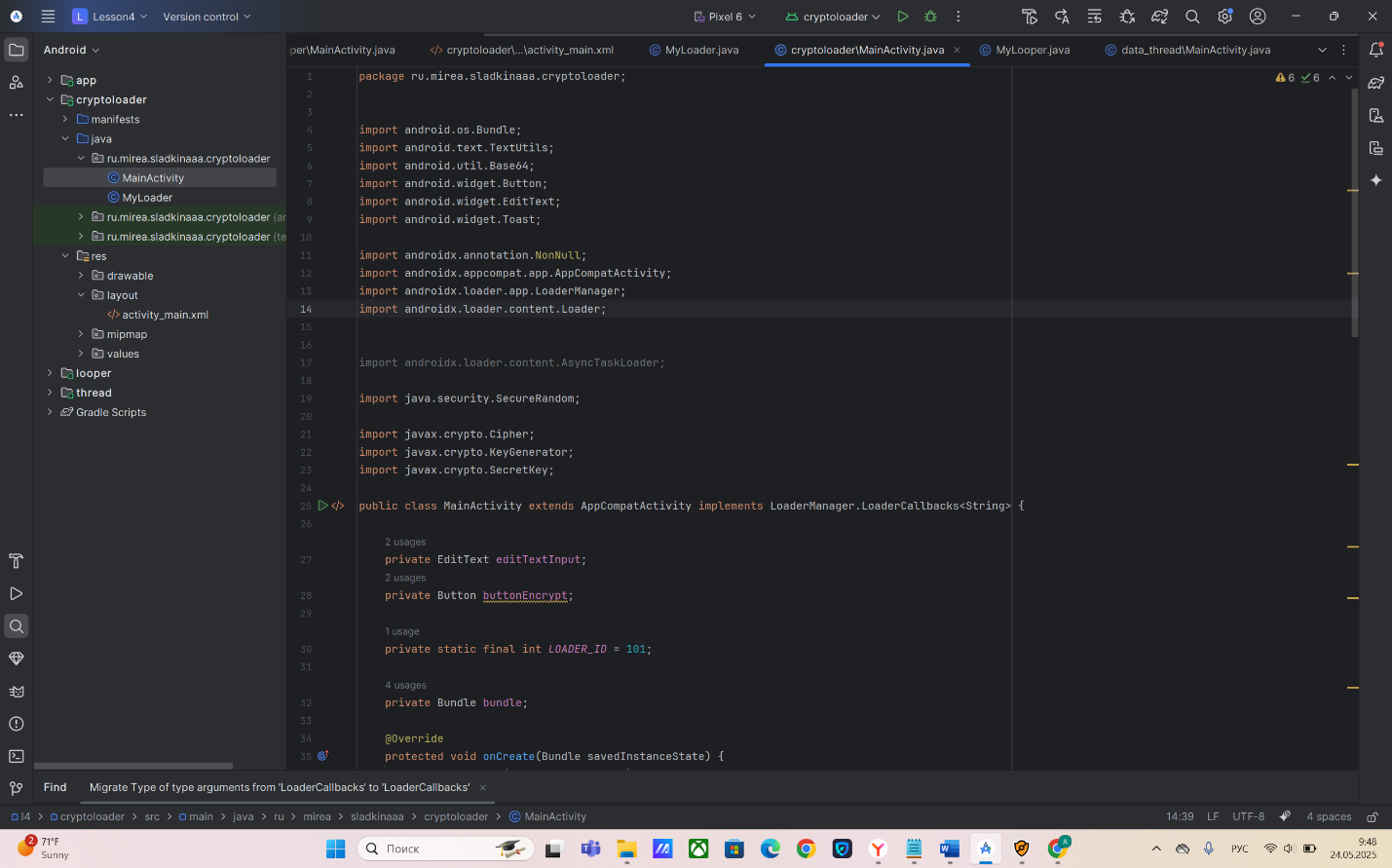


Рисунок 15. Начало работы с шифрованием, определение зависимостей

При нажатии кнопки buttonEncrypt сначала проверяется, введён ли текст в поле editTextInput. Если поле пустое, выводится уведомление с требованием ввести текст. Если текст присутствует, система генерирует 128-битный ключ AES (SecretKey) с помощью KeyGenerator и случайного числа SecureRandom. Затем создаётся экземпляр Cipher с алгоритмом AES в режиме ECB и заполнением PKCS5Padding, который инициализируется в режиме шифрования ENCRYPT\_MODE с использованием сгенерированного ключа.

Текст шифруется методом doFinal(), а результат в виде байтового массива кодируется в строку Base64 для удобства хранения и передачи. Ключ также кодируется в Base64. Эти данные упаковываются в Bundle и передаются в LoaderManager для дальнейшей обработки (см. рис.16).

Если в процессе шифрования возникает ошибка (например, неподдерживаемый алгоритм или некорректные данные), она перехватывается блоком catch, а пользователь получает уведомление об ошибке.



Рисунок 16. Реализация шифрования

Далее был реализован пользовательский класс MyLoader, который наследуется от AsyncTaskLoader<String> и предназначен для асинхронной дешифровки данных в фоновом потоке (см. рис. 17). При создании экземпляра MyLoader передаются контекст и Bundle с зашифрованным текстом encryptedText и ключом key, закодированными в Base64.

Метод loadInBackground() выполняет основную работу:

1. Декодирует зашифрованный текст и ключ из Base64 в байтовые массивы.
2. Создаёт объект SecretKeySpec на основе декодированного ключа, используя алгоритм AES.
3. Инициализирует Cipher в режиме дешифровки (DECRYPT\_MODE) с указанным ключом и преобразованием "AES/ECB/PKCS5Padding".
4. Дешифрует данные методом doFinal() и преобразует результат в строку.

Если входные данные (args) отсутствуют или в процессе дешифровки возникает ошибка (например, некорректный ключ или алгоритм), метод возвращает строку *"0шибка дешифровки"* . Метод onStartLoading() принудительно запускает загрузку данных вызовом forceLoad().

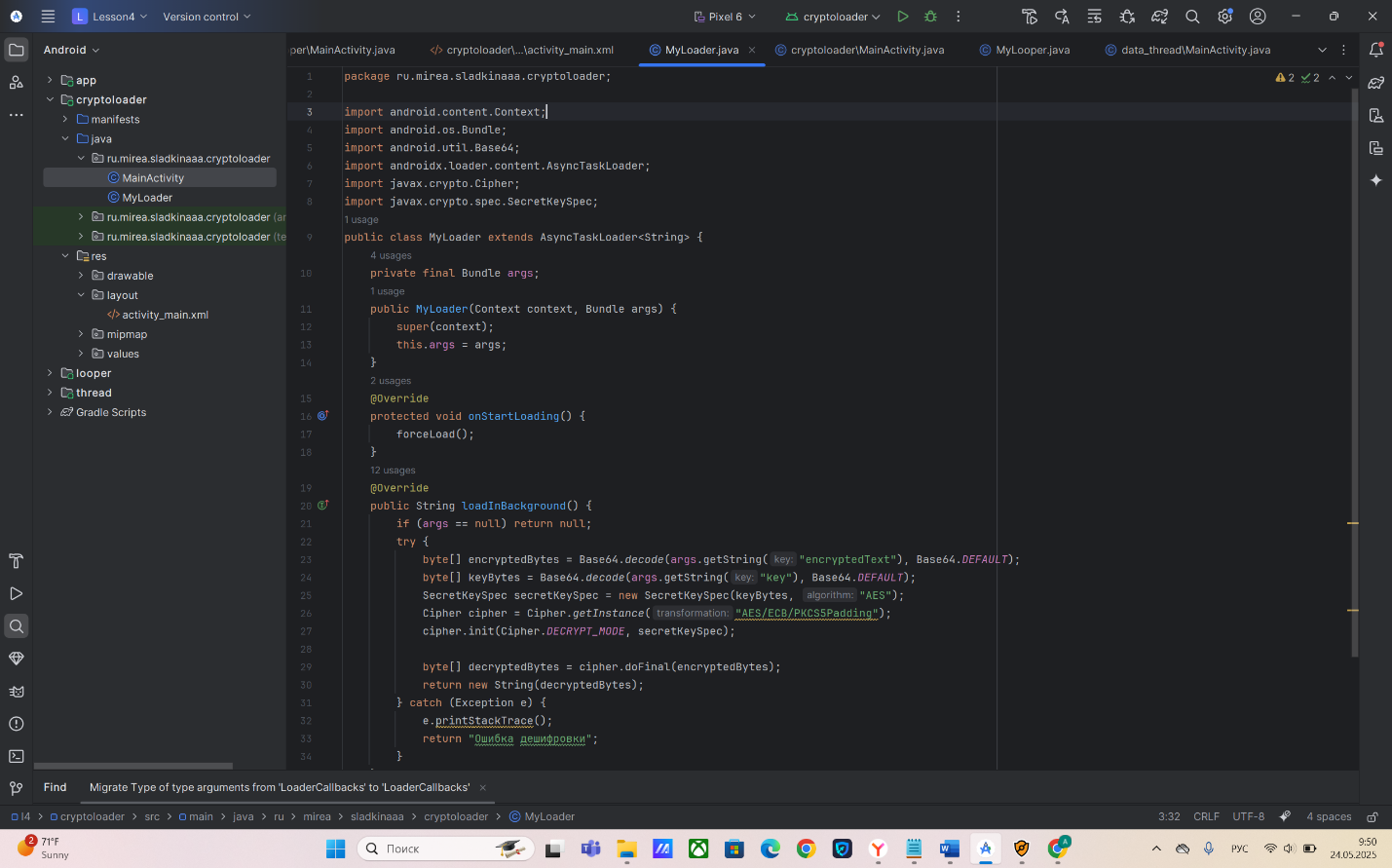


Рисунок 17. Реализация класса MyLoader

Чтобы определить службу, необходимо создать новый класс, расширяющий базовый класс Service возможно воспользоваться готовым мастером создания класса для сервиса в «Android Studio». Т.к. сервис является компонентом приложения, требуется убедиться в существовании записи о сервисе в секции «application» манифест-файла (см. рис. 18).

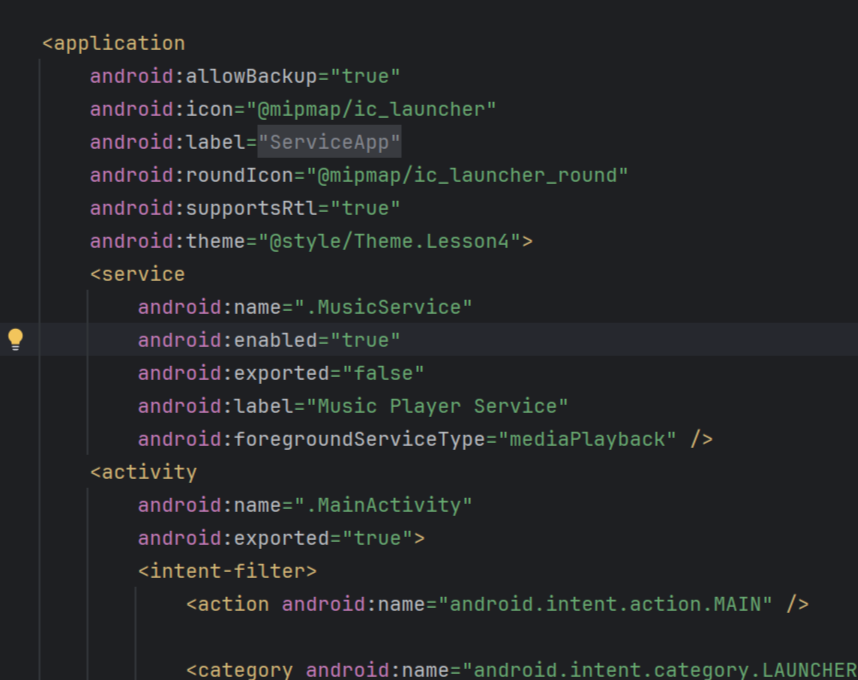


Рисунок 18. Добавление в манифест-файл

Далее был добавлен функционал воспроизведения аудиофайла. В первую очередь необходимо добавить медиа файл (типа «.mp3» и т.д.) в ресурсы. В активности добавить две кнопки «button» для воспроизведения и остановки музыкальных композиций/композиции (см. рис. 19).

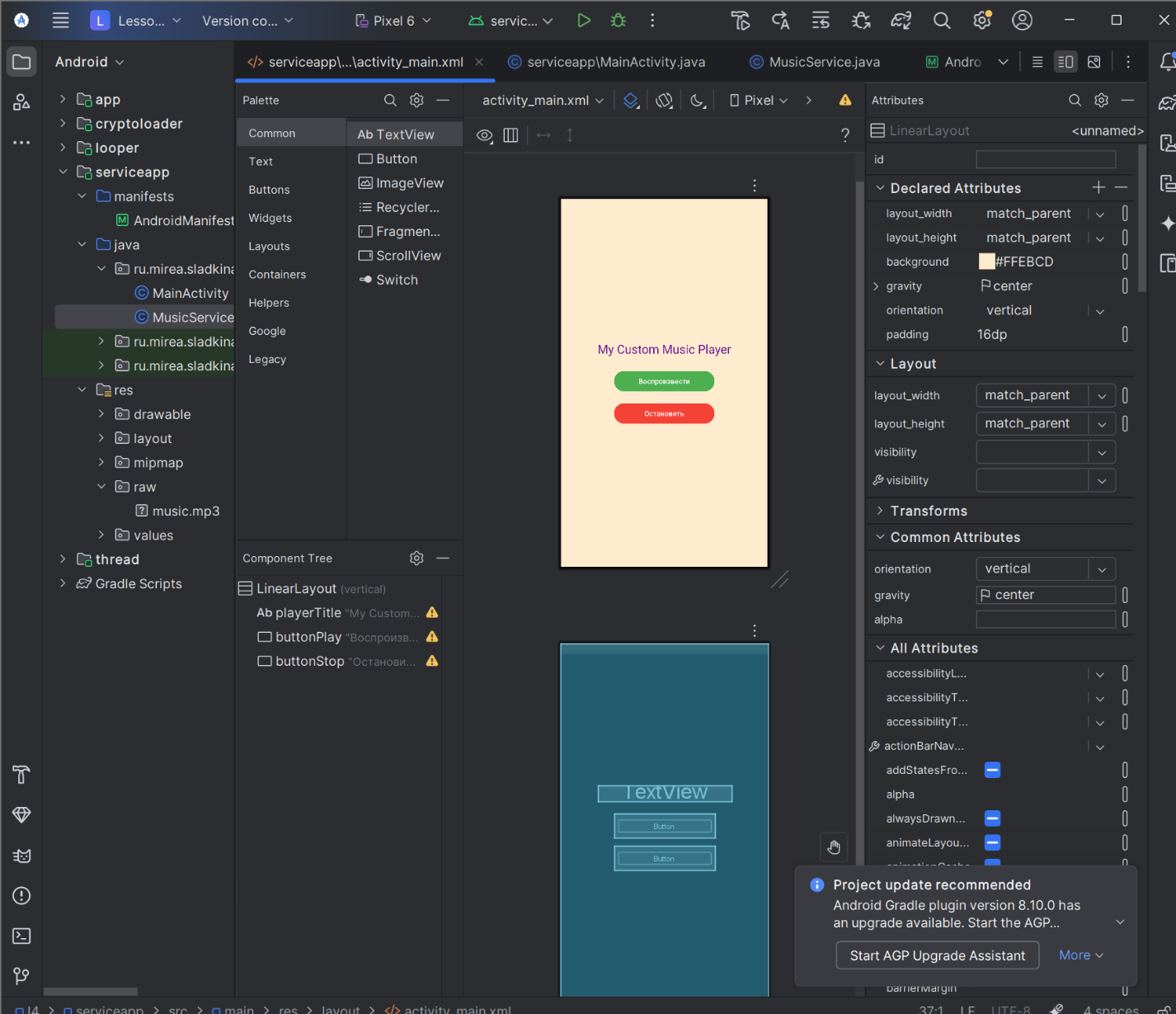


Рисунок 19. Дизайн данного проигрывателя

Далее рассмотрим реализацию сервиса MusicService, предназначенный для воспроизведения музыки в фоновом режиме (см. рис. 20). Сервис наследуется от Service и реализует интерфейс MediaPlayer.OnCompletionListener для отслеживания завершения воспроизведения трека.

При создании сервиса onCreate() инициализируется MediaPlayer с аудиофайлом, указанным в ресурсах R.raw.music, и устанавливается слушатель завершения воспроизведения. Также вызывается метод createNotificationChannel(), который создаёт канал уведомлений.

При запуске сервиса onStartCommand() начинается воспроизведение музыки, если MediaPlayer не воспроизводит трек в данный момент. Одновременно создаётся уведомление с использованием NotificationCompat.Builder, которое отображается в статус-баре.

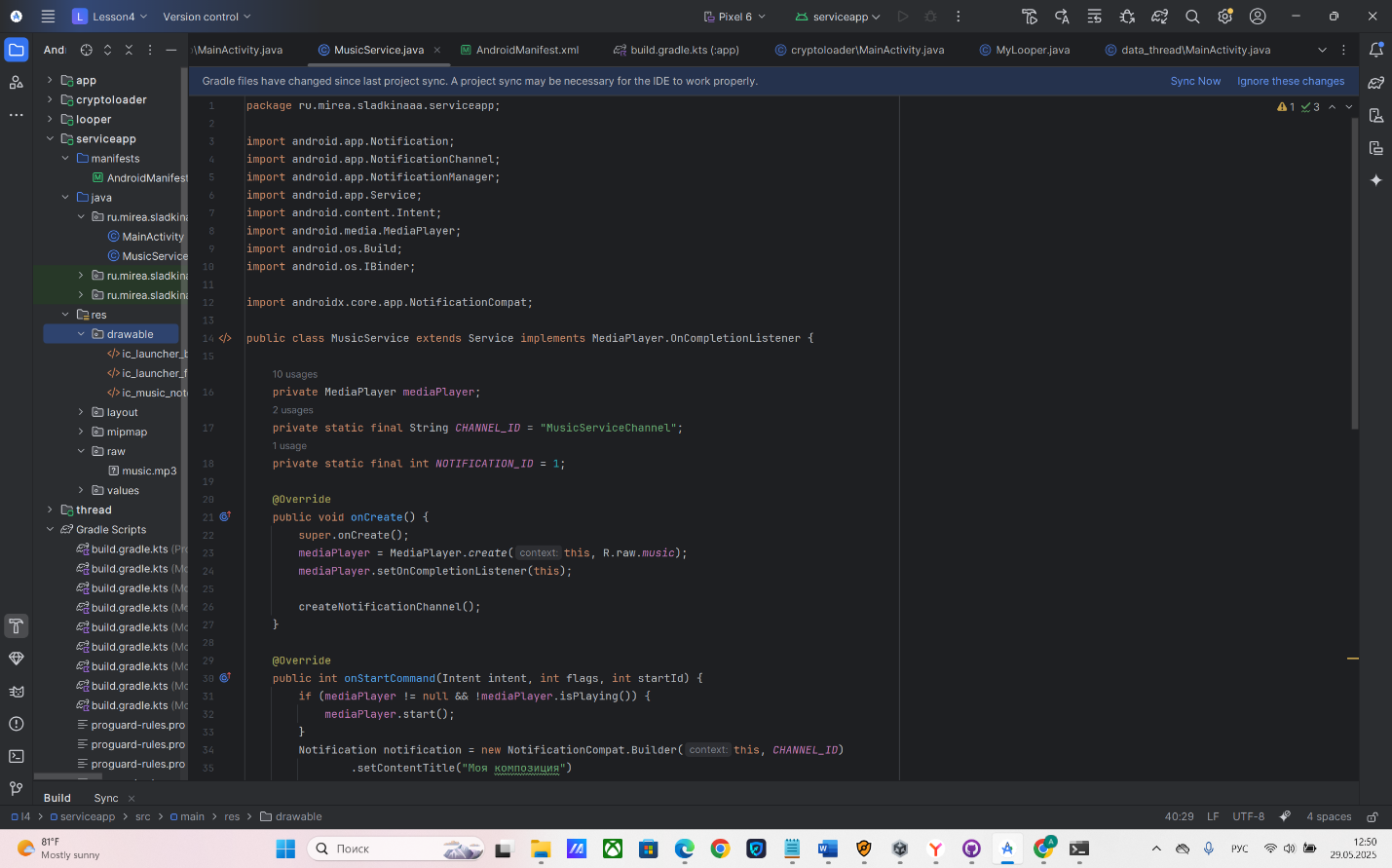


Рисунок 20. Реализация сервиса MusicService

Уведомление переводит сервис в режим startForeground, что позволяет ему работать даже при закрытом приложении (см. рис. 21).

При уничтожении сервиса onDestroy останавливается воспроизведение, освобождаются ресурсы MediaPlayer, а уведомление удаляется. Метод onCompletion автоматически вызывается при завершении трека: сервис останавливает себя stopSelf и снимает уведомление.

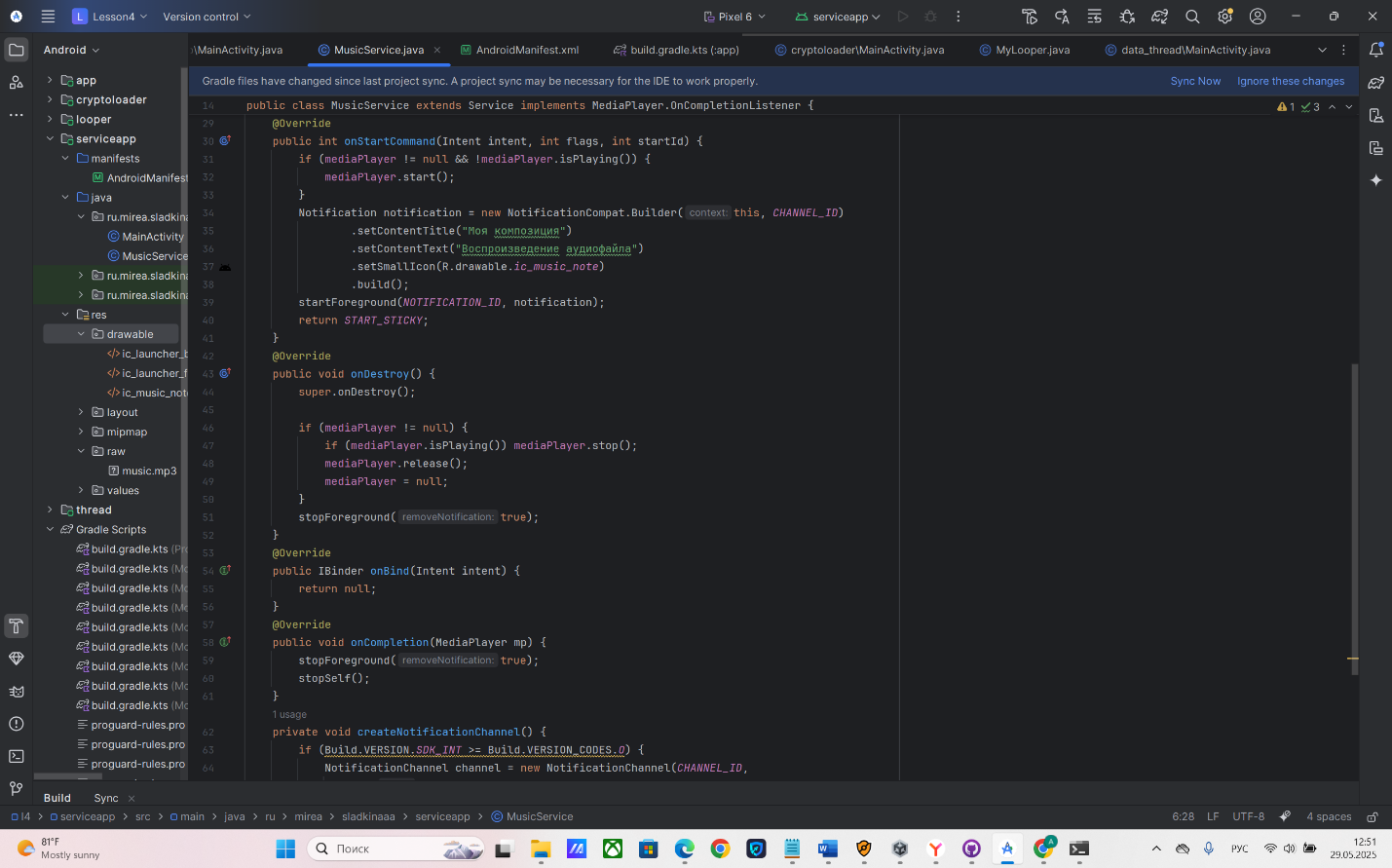


Рисунок 21. Завершение сервиса MusicService

Рассмотрим обработку сервиса через класс MainActivity (см. рис. 22). Метод checkAndRequestPermissions() проверяет наличие разрешения Manifest.permission.FOREGROUND\_SERVICE (или других, например, на чтение хранилища для аудиофайлов). Если разрешение не предоставлено, вызывается requestPermissionLauncher.launch(), который отображает стандартный диалог запроса. Результат обрабатывается в лямбда-выражении с выводом соответствующего уведомления. При нажатии buttonPlay создаётся Intent для запуска MusicService. Для Android 8.0 (API 26) и выше используется startForegroundService(), чтобы сервис мог работать в фоне с обязательным уведомлением.

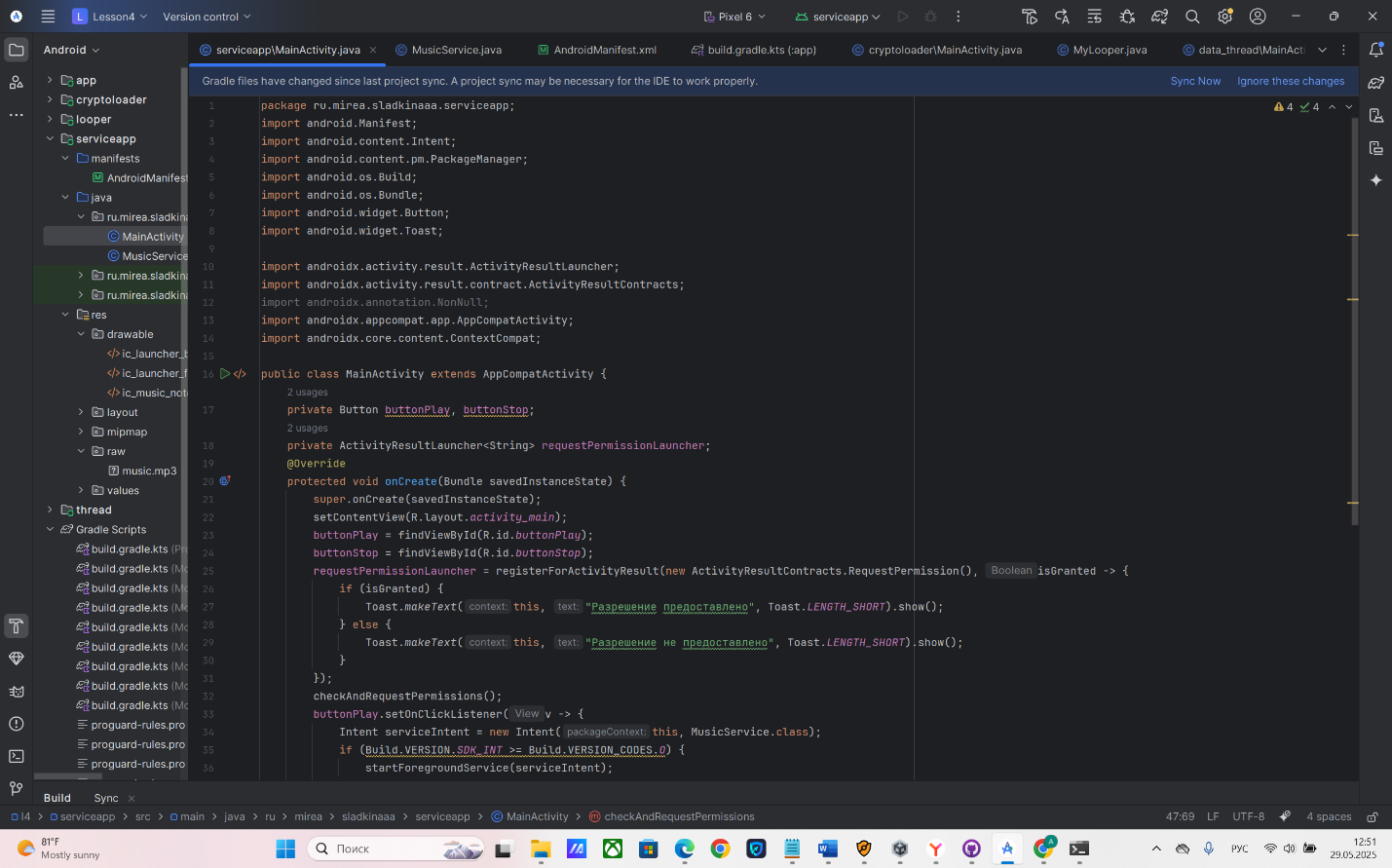


Рисунок 22. Класс MainActivity

Для работы с WorkManager в файл build.gradle(Module …) в раздел dependencies требуется добавить библиотеку для работы с worker (см. рис. 23).

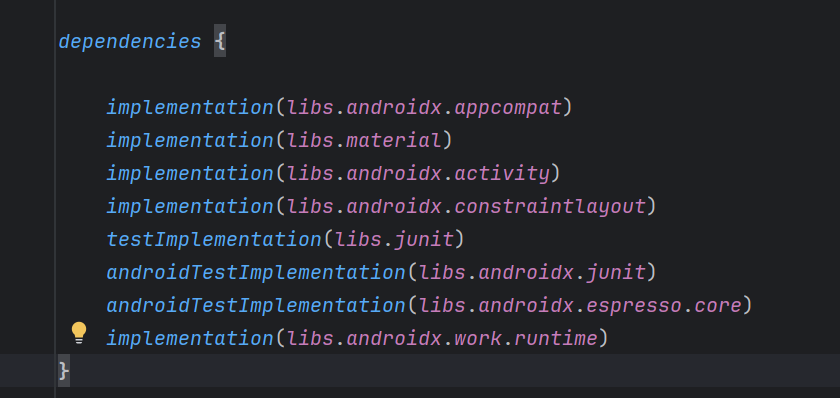


Рисунок 23. Добавление библиотеки

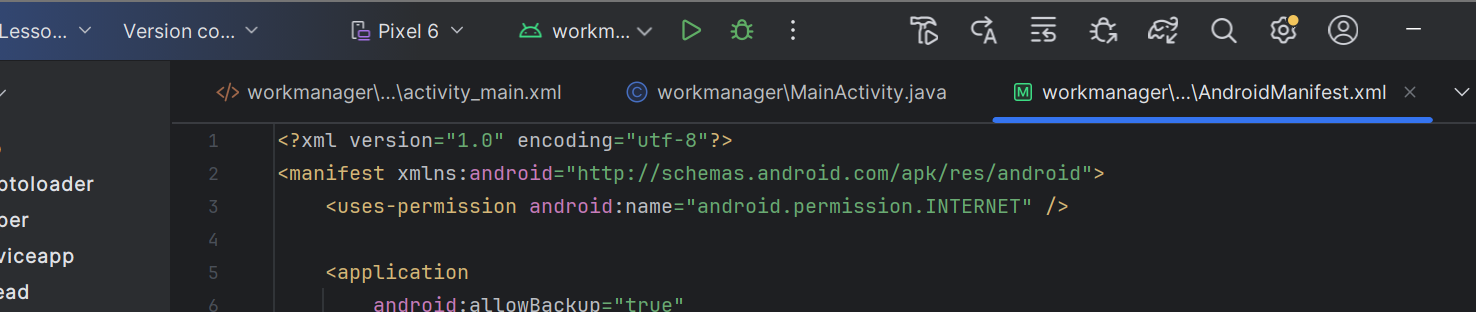


Рисунок 24. Добавление разрешения в манифесте

  MainActivity настраивает и запускает фоновую задачу с помощью WorkManager (см. рис. 25).

Создается одноразовая задача OneTimeWorkRequest для класса UploadWorker . Затем задача ставится в очередь через WorkManager.getInstance(this).enqueue().

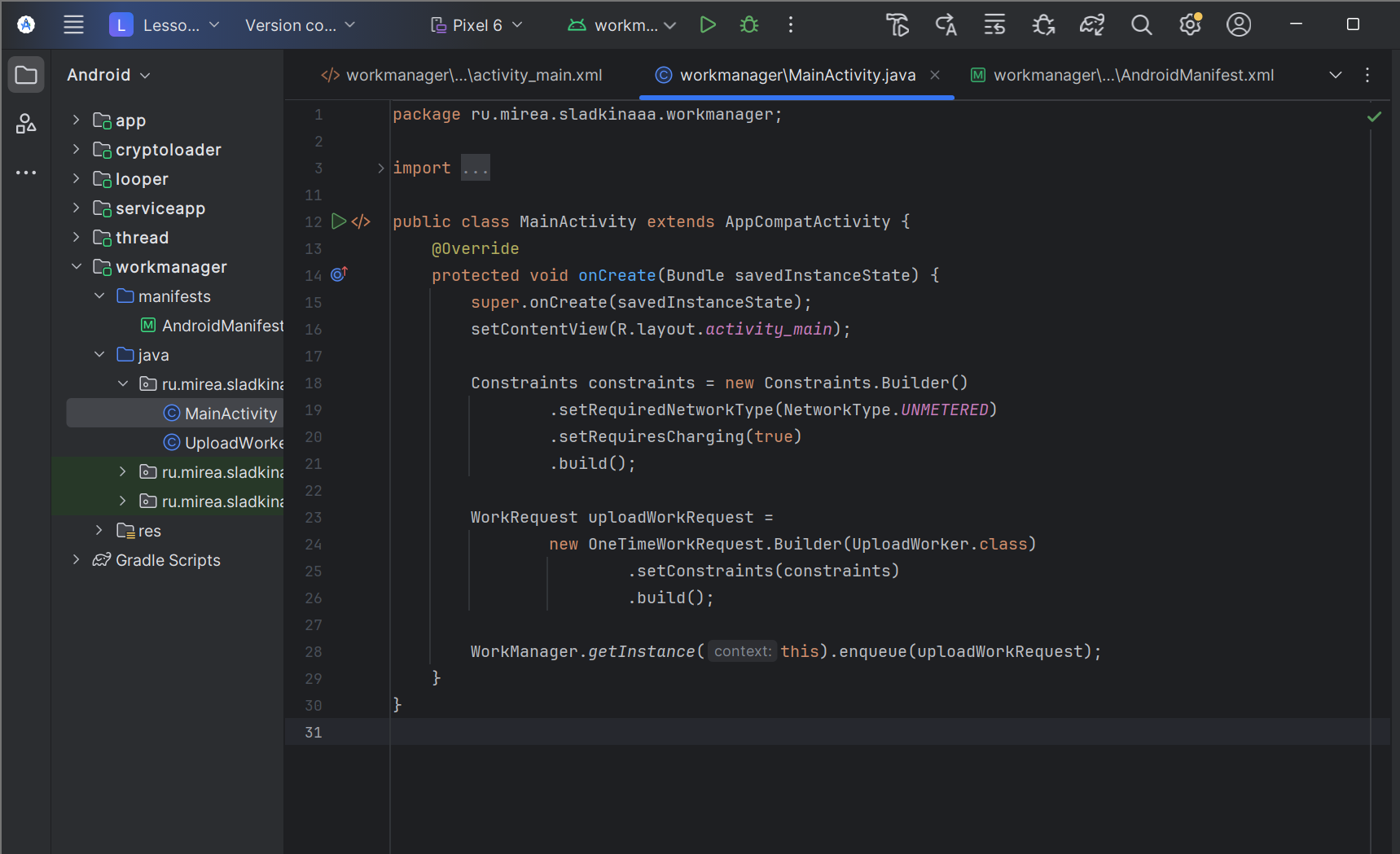


Рисунок 25. Настройка фоновой задачи в MainActivity

Класс UpLoadWorker наследуется от Worker и предназначен для выполнения фоновой задачи в рамках библиотеки WorkManager (см. рис. 26). При создании экземпляра класса передаются контекст и параметры, которые инициализируются через конструктор родительского класса.

Основная логика задачи реализована в методе doWork(), который выполняется в фоновом потоке. В начале метода выводится лог с сообщением *"doWork: start"*, после чего поток приостанавливается на 10 секунд с помощью TimeUnit.SECONDS.sleep(). Если во время ожидания возникает прерывание, метод возвращает Result.failure(), указывая на неудачное завершение задачи. В случае успешного выполнения после паузы выводится лог *"doWork: end"*, и метод возвращает Result.success().



Рисунок 26. Класс UpLoadWorker

Для проекта MireaProject можно реализовать выполнение фоновой задачи через Worker с использованием WorkManager. Фрагмент MyWorkerFragment управляет фоновой задачей с помощью WorkManager (см. рис. 27). При создании представления фрагмента onCreateView загружается макет fragment\_my\_worker, содержащий кнопку startWorkBtn и текстовое поле statusTextView. В методе onViewCreated кнопка связывается с методом startBackgroundWork(), который запускает фоновую задачу при нажатии.

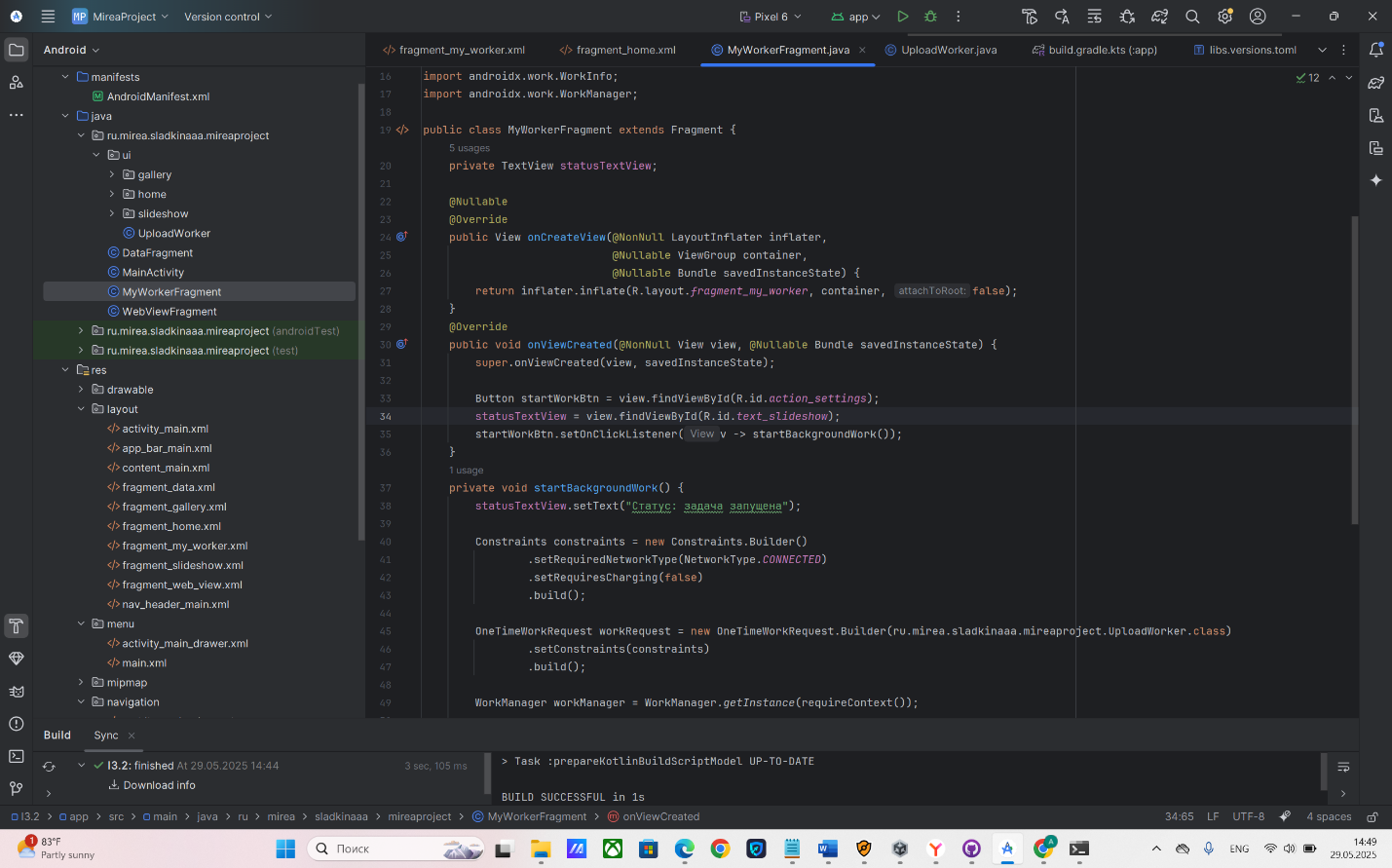


Рисунок 27. Фрагмент MyWorkerFragment