<TeachMeSkills/>

курс

Android разработчик

Фоновая работа

Агенда занятия:

Services

Work Manager

- **Services**

  Work Manager

Services

Сервис – это компонент приложения, выполняющий длительные операции в фоновом режиме без пользовательского интерфейса. После запуска сервис может продолжать работу некоторое время даже при переключении пользователя на другое приложение. Другие компоненты (даже из других приложений) могут связываться с сервисом для взаимодействия с ним через межпроцессное взаимодействие (IPC). Сервисы пригодны для задач, требующих фонового выполнения независимо от UI, например сетевые транзакции, загрузка/выгрузка файлов, воспроизведение музыки и другие операции ввода-вывода.

Разновидности сервисов

Started Service (Запущенный сервис) – сервис, запущенный вызовом startService() (или с Android 8+ – startForegroundService() для услуг переднего плана).
Запускается и работает самостоятельно до явной остановки методом stopService() или stopSelf().
Подходит для фоновых операций, которые должны выполняться независимо от активности.
Например, сервис для загрузки файла: однажды запущен, он продолжает скачивание в фоне даже если пользователь закрыл активность.
Важно, что запущенный сервис по умолчанию работает в основном потоке приложения, поэтому для длительных или тяжелых операций внутри него нужно запускать отдельный поток. Если этого не сделать, долгий код в сервисе заблокирует UI-поток.

Разновидности сервисов

Foreground Service (Сервис переднего плана) – особый вид запущенного сервиса, который получает повышенный приоритет и работает с уведомлением, видимым пользователю.

Foreground-сервис используют для задач, о которых пользователь должен знать. Например, музыкальный плеер запускает foreground-сервис для воспроизведения музыки, показывая уведомление с контролами проигрывателя. Пока сервис переднего плана активен, система старается его не убивать (он менее подвержен выгрузке).

Разновидности сервисов

Однако разработчик обязан показать уведомление и вызвать **startForeground()** внутри сервиса сразу после запуска, иначе система остановит сервис. Уведомление нельзя убрать, пока сервис работает как foreground.
С Android 8 Oreo вводятся ограничения: приложение не может запускать фоновые сервисы, находясь в фоне. Вместо этого необходимо использовать **startForegroundService(),** и в течение 5 секунд перевести сервис в foreground с помощью startForeground(). Начиная с Android 12, введены дополнительные ограничения – нельзя запускать foreground-сервис из фона без явного пользовательского действия, иначе будет брошено исключение

Разновидности сервисов

Bound Service (Связанный сервис) – сервис, к которому другие компоненты приложения могут привязаться вызовом bindService(). При привязке сервис предоставляет интерфейс (обычно через IBinder) для взаимодействия "клиент-сервер" – компоненты могут вызывать методы сервиса, получать результаты, выполнять RPC-вызовы.
Связанный сервис живёт пока есть хотя бы один привязанный к нему компонент. Как только клиенты отвязались, сервис автоматически останавливается и уничтожается системой. Этот вид сервисов полезен, когда нужно длительное взаимодействие с фоновым компонентом. Например, музыка может воспроизводиться в сервисе, а Activity привязывается к нему чтобы управлять воспроизведением (плей/пауза) и получать информацию о треках.

Разновидности сервисов

В Android ранее существовал также класс **IntentService** – вспомогательный подкласс сервиса, предназначенный для удобного выполнения одиночных фоновых задач (сервис сам запускал поток и прекращался по окончании задачи).
Однако начиная с API 30 IntentService помечен устаревшим (deprecated), поскольку его функциональность можно заменить сочетанием современных API (например, того же WorkManager) или вручную управлять потоками в обычном сервисе.

Жизненный цикл сервиса

У сервисов есть свой жизненный цикл, управляемый системой.

Основные callback-методы сервиса:

- **onCreate()** вызывается при первоначальном создании сервиса (до вызова **onStartCommand() или onBind()**).

Здесь выполняются разовые операции и инициализация. Если сервис уже был запущен и не уничтожен, повторно onCreate() не вызывается.

Жизненный цикл сервиса

- **onStartCommand(Intent, int, int)** вызывается при каждом запуске сервиса через startService() (или startForegroundService()). Приходит Intent с данными. Метод должен вернуть код, определяющий поведение при неожиданной остановке сервиса системой.

Жизненный цикл сервиса

Основные варианты:

START_NOT_STICKY – не перезапускать сервис после убийства системой (если не осталось отложенных Intent’ов);

START_STICKY – перезапустить сервис после убийства, но без повторной доставки последнего Intent (подходит для сервисов, работающих неопределенно долго, например музыкальный плеер);

START_REDELIVER_INTENT – перезапустить и повторно передать последний Intent (подходит для задач, которые должны быть возобновлены, например незавершённая загрузка файла).

Жизненный цикл сервиса

Каждый вызов startService() приводит к вызову onStartCommand() в уже существующем сервисе, передавая новый Intent. Таким образом, один и тот же сервис может быть стартован много раз (с разными Intent) – все запросы будут обрабатываться последовательно.

Важно: разработчик сам управляет временем жизни запущенного сервиса. После завершения работы следует остановить сервис вызовом **stopSelf()** (внутри сервиса) или **stopService()** (извне). Иначе сервис будет оставаться запущенным неопределенно долго (пока система не решит его убить для освобождения памяти).

Жизненный цикл сервиса

- **onBind(Intent)** вызывается, когда компонент хочет привязаться к сервису через bindService(). Должен вернуть объект IBinder для взаимодействия с клиентом или null, если привязка не поддерживается.

Если сервис позволяет привязку, он должен реализовать этот метод. Связанный сервис не требует вызывать stopSelf(), он автоматически остановится, когда отвяжутся все клиенты. Можно комбинировать запуск и привязку: если сервис уже запущен через startService(), привязка не приведёт к повторному onCreate(), но onBind() будет вызван. Если сервис запущен и привязан одновременно, для его остановки нужно или явно вызвать stopSelf(), или отвязать всех клиентов и вызвать stopService().

Жизненный цикл сервиса

- **onDestroy()** вызывается перед уничтожением сервиса системой (после stopSelf/stopService или при убивании процесса).

Здесь следует освобождать ресурсы: останавливать фоновые потоки, отменять регистрацию приемников, снимать уведомления и т.п.

Жизненный цикл сервиса

Жизненный цикл зависит от типа сервиса:

Если сервис был запущен, он остаётся работать до явной остановки (независимо от компонентов, его запустивших).

Если сервис только привязанный (никогда не запускался через startService()), то его жизнь ограничена наличием привязки – когда все клиенты отключаются, сервис уничтожается автоматически.

Примеры использования сервисов

Сервисы применяются для выполнения действий в фоне вне зависимости от активности.
Воспроизведение музыки или аудио – медиаплееры используют сервис переднего плана, чтобы музыка продолжала играть при свернутом приложении, а пользователь мог видеть уведомление с управлением воспроизведением.

Загрузка или выгрузка файлов – длительные сетевые операции (скачивание больших файлов, аплоад фото/видео) удобно выполнять в сервисе. Например, можно запустить сервис загрузки, который продолжит скачивание, даже если пользователь покинул экран загрузки. Для критичных загрузок можно использовать foreground-сервис с уведомлением о прогрессе.

Примеры использования сервисов

Навигация и отслеживание геопозиции – приложения-навигаторы запускают сервис для получения GPS-координат и маршрутизации в реальном времени. Как правило, это foreground-сервис, отображающий значок навигации, поскольку пользователь ожидает постоянного сопровождения.
Фоновые вычисления и обработка данных.
Обслуживание IPC или контент-провайдеров – некоторые приложения выносят в сервис логику взаимодействия с контент-провайдерами или другими приложениями. Например, сервис может постоянно слушать входящие сообщения (как в мессенджерах для чата) или предоставлять данные другим приложениям через привязанный сервис.

Примеры использования сервисов

Важно помнить, что запуск сервиса оправдан, только если задача должна выполняться, когда пользователь не взаимодействует с приложением.
Если же фоновые действия нужны лишь во время работы активности (например, воспроизведение музыки только при открытой Activity проигрывателя), то достаточно использовать поток (Thread) или корутину без создания сервиса. Сервис стоит применять для действительно автономных фоновых задач.

Как это всё работает

Смотрим проект

Services

- **Work Manager**

WorkManager

WorkManager – это библиотека Android Jetpack, предназначенная для управления фоновой работой, которая должна выполняться гарантированно, но не обязательно немедленно.
WorkManager решает многие проблемы прямого использования сервисов и предыдущих API, обеспечивая удобный и надежный способ планировать задачи, которые должны выполняться позже, периодически или при соблюдении определённых условий.

WorkManager

WorkManager был представлен Google в 2018 году как единое решение для отложенных задач в Android (замена устаревших AlarmManager, JobScheduler, Firebase JobDispatcher и др.). Он входит в набор Jetpack и доступен через поддержку библиотек начиная с API 14. Под капотом WorkManager использует различные механизмы (AlarmManager, JobScheduler и т.д.) в зависимости от версии ОС, но предоставляет единый высокоуровневый API.

Концепции и возможности WorkManager

1. Единый API для фоновых задач.

WorkManager упрощает запуск фоновых работ, инкапсулируя работу с потоками, очередями, ограничениями. Разработчик описывает задачу в виде Worker, а WorkManager сам решает, каким образом и когда её выполнить.

Концепции и возможности WorkManager

2. Гарантированное выполнение.

Главное преимущество WorkManager – он стремится гарантированно выполнить задачу, даже если приложение было закрыто или устройство перезагружено. Для этого WorkManager хранит информацию о заданиях во внутренней базе данных. Если приложение или процесс упали, при следующем запуске WorkManager восстановит статус задач и продолжит с того места, где остановился. На современных версиях Android WorkManager задачи переживают перезагрузку устройства.

Концепции и возможности WorkManager

3. Отложенные и повторяющиеся задачи.

WorkManager сразу поддерживает как одноразовые, так и периодические задачи. То есть можно запланировать работу один раз на определённое время или с определённой задержкой, а можно запускать регулярно через заданный интервал.

При этом библиотека учитывает системные оптимизации (минимальный интервал для повторяющихся задач – 15 минут, даже если указать меньше, WorkManager округлит до 15 мин).

Концепции и возможности WorkManager

4. Условия выполнения (Constraints).

Важнейшая особенность – возможность задать условия, при которых задача должна стартовать. WorkManager поддерживает критерии: устройство на зарядке, достаточный уровень батареи, требуемый тип сети и т.д. Задача будет запущена только когда выполняются все заданные условия, что предотвращает нежелательную активность.

Концепции и возможности WorkManager

5. Цепочки задач и параллелизм.

WorkManager позволяет комбинировать работы в последовательные цепочки или запускать их параллельно, определяя зависимости между задачами. Можно настроить, чтобы после успешного выполнения одной задачи автоматически запускалась следующая. Также поддерживается уникальная идентификация задач и политика при коллизиях.

Концепции и возможности WorkManager

6. Отслеживание состояния и результаты.

WorkManager предоставляет API для мониторинга выполнения. Можно получить LiveData<WorkInfo> или колбэки о статусе задачи (запланирована, выполняется, успешно завершена, провалена, отменена и т.п.).

Worker возвращает результат (Result.success(), Result.failure(), Result.retry()), который может включать выходные данные.

Концепции и возможности WorkManager

7. Совместимость с современными технологиями.

WorkManager интегрируется с Kotlin Coroutines (класс CoroutineWorker), поддерживает RxJava, и хорошо сочетается с архитектурными компонентами (LiveData, ViewModel). Он также может работать в отдельном процессе, что повышает устойчивость тяжелых фоновых задач, изолируя их от основного приложения.

Концепции и возможности WorkManager

8. Expedited Work (Срочные задачи).

В последних версиях WorkManager появилась возможность отмечать задачу как срочную (Expedited), требующую немедленного запуска. Такие задачи будут выполнены, минуя обычные ограничения оптимизации батареи. Реализовано это через использование Foreground Service под капотом – WorkManager сам запустит foreground-сервис для вашей задачи. Разработчику нужно предоставить уведомление **(ForegroundInfo)** для этой задачи, чтобы соответствовать требованиям системы. Expedited Work позволяет, например, сразу начать загрузку важного файла даже в режиме энергосбережения. Однако у срочных задач есть квота (лимит частоты использования), поэтому злоупотреблять ими не стоит.

Концепции и возможности WorkManager

9. Backward compatibility.

В отличие от чисто системного API JobScheduler (требует API 21+) или AlarmManager (ограничен на новых ОС), WorkManager доступен на устройствах с API 14 и выше, поскольку использует под капотом подходящие механизмы в зависимости от версии ОС. Это делает WorkManager универсальным решением для подавляющего большинства устройств – разработчик может писать единый код фоновых задач, и он будет работать на старых и новых Android.

Концепции и возможности WorkManager

10. В совокупности, WorkManager позиционируется как “лучшее решение для надежного выполнения фоновых задач”, рекомендуемое Google. Он способен заменить устаревшие подходы планирования (AlarmManager, GcmNetworkManager, Firebase JobDispatcher, прямое использование Service для отложенных задач) более удобным и оптимизированным способом.

Как это всё работает

Смотрим проект

Типы задач

При работе с WorkManager каждая задача ставится в очередь через **WorkRequest** – объект запроса на выполнение.

Существуют два основных типа WorkRequest: **OneTimeWorkRequest** и **PeriodicWorkRequest.**

Типы задач

OneTimeWorkRequest – запрос на одноразовое выполнение задачи. Используется, когда нужно выполнить работу один раз, возможно, с некоторой задержкой или по какому-то событию. Например, отправить логи на сервер, как только устройство подключится к Wi-Fi; или выполнить разовую синхронизацию по запросу пользователя.
Создаётся с помощью **OneTimeWorkRequestBuilder<YourWorker>().** Можно настроить начальную задержку .setInitialDelay(), условия .setConstraints() и т.д., после чего вызвать enqueue().

Типы задач

PeriodicWorkRequest – запрос на периодическое выполнение. Планирует бесконечную серию запусков задачи через заданные интервалы.
Создается через **PeriodicWorkRequestBuilder<YourWorker>(repeatInterval, timeUnit).**
Периодические задачи автоматически повторяются системой, минимальный интервал – 15 минут (это ограничение системы для экономии батареи). Можно также задать гибкое окно запуска (flex interval) – например, повторять каждые 24 часа с допуском +/- 2 часа, позволяя системе сдвигать запуск для оптимизации.
Примеры использования: ежедневная синхронизация данных, регулярное резервное копирование на сервер, периодическая очистка кеша и т.п.

Типы задач

Оба типа WorkRequest добавляются в WorkManager одинаково:
WorkManager.getInstance(context).enqueue(request).

WorkManager сам будет отслеживать, когда и как выполнить задачу (сразу или по расписанию). В случае периодических задач WorkManager гарантирует, что при пропуске одного запуска (например, устройство было выключено), задача всё равно будет поставлена при следующей возможности, но не будет выполняться параллельно два экземпляра одной периодической задачи.

Ограничения (Constraints) выполнения задач

WorkManager позволяет задать **Constraints** – условия, при которых фоновая задача должна стартовать. Эти ограничения помогают выполнять задачи наиболее подходящим образом, не мешая пользователю и не разряжая лишний раз батарею.

Основные возможности **Constraints:**

- **Наличие зарядки:** setRequiresCharging(true) – задача начнет выполняться только когда устройство подключено к зарядному устройству. Полезно для энергоемких операций откладывая их до момента, когда телефон на зарядке.

Ограничения (Constraints) выполнения задач

- **Не низкий заряд батареи:** setRequiresBatteryNotLow(true) – задача не стартует, если заряд батареи низкий (обычно ниже ~15-20%). Это защищает устройство от разряда; задача автоматически запустится, когда пользователь зарядит телефон.

- **Тип сети:** setRequiredNetworkType(NetworkType) – требуемое сетевое подключение для выполнения. Например, для синхронизации больших данных имеет смысл требовать Wi-Fi, чтобы не тратить мобильный трафик.

Ограничения (Constraints) выполнения задач

- **Простой устройства:** setRequiresDeviceIdle(true) требует, чтобы устройство было неактивно (idle). Этот флаг учитывается на API 23+ и означает, что задача подождёт, пока пользователь не перестанет активно использовать устройство. Полезно для задач, которые можно выполнить в спящий период.

Ограничения (Constraints) выполнения задач

- **Достаточно свободной памяти:** setRequiresStorageNotLow(true) – задача не запустится, если свободной памяти на устройстве мало. Например, это убережет от попытки сделать резервную копию, когда на телефоне почти не осталось места

Указанные ограничения задаются через **Constraints.Builder** и привязываются к WorkRequest (.setConstraints(constraints)).
Все условия рассматриваются совокупно: WorkManager поставит задачу в состояние "ENQUEUED" и будет ждать, пока все условия выполнены.

Ограничения (Constraints) выполнения задач

Если в процессе выполнения условия перестают выполняться (например, сеть пропала или батарея разрядилась ниже порога), WorkManager может приостановить задачу. На практике, при пропаже условий WorkManager может прервать работу и вернуть ее в состояние ENQUEUED до повторного соблюдения условий (для надёжности лучше разрабатывать таски, способные продолжить/повторить выполнение).

Примеры использования WorkManager

WorkManager удобен для ситуаций, когда задача может быть выполнена с отложением или требуется надежный гарантированный запуск. Рассмотрим несколько примеров и сценариев:

1. Синхронизация данных с сервером.
Допустим, приложение должно периодически отправлять или запрашивать обновления. С помощью WorkManager можно поставить периодическое задание, которое будет выполняться раз в N часов только при наличии интернета и только на Wi-Fi и зарядке (чтобы не разрядить телефон). WorkManager проследит, чтобы задача запустилась в соответствии с этими условиями и повторялась регулярно.

Примеры использования WorkManager

2. Резервное копирование и долгие выгрузки.

Если нужно сделать бэкап пользовательских данных на облако, не надо держать приложение открытым. WorkManager позволяет запланировать одноразовую задачу с условием "девайс на зарядке, Wi-Fi". Задача начнет выполняться, когда условия выполнены, и даже если пользователь перезагрузит телефон в процессе, WorkManager продолжит с того же места после перезагрузки.

Пример: фотогалерея, которая резервирует фото в облаке. Можно запланировать OneTimeWorkRequest на резервное копирование 1000 фотографий. WorkManager будет выполнять его в фоновом режиме и гарантирует завершение – он сохраняет прогресс в БД и может возобновить работу после рестарта приложения или телефона

Примеры использования WorkManager

3. Работа по расписанию.

WorkManager удобен для периодических работ: обновление виджета каждый час, опрос сервера раз в день, отправка статистики раз в 24 часа и т.д.

Например, приложение шагомера может с помощью PeriodicWorkRequest раз в день сохранять результат на сервер, только когда телефон на зарядке и в Wi-Fi, чтобы не тратить лишнюю батарею.

Примеры использования WorkManager

4. Комплексные сценарии с зависимостями.

За счёт цепочек WorkManager позволяет реализовать целые пайплайны фоновых операций.

Например, нужно сначала скачать список обновлений, затем для каждого обновления скачать файл, затем объединить результаты и уведомить пользователя – всё это можно оформить как последовательность WorkRequest’ов с передачей данных между ними. Если какая-то из задач в цепочке провалится, последующие автоматически отменятся, а можно настроить повторную попытку или другое поведение.

Примеры использования WorkManager

Подводя итог, WorkManager особенно хорош для работ, которые могут быть выполнены асинхронно, с гибкими требованиями к времени запуска, но которые должны быть выполнены гарантированно.
Он берет на себя заботы о повторных запусках, учете системных ограничений совместимости с разными версиями Android – разработчику нужно лишь описать что делать и при каких условиях.

Сервис vs WorkManager

Когда лучше подойдет Service: если фоновая задача требует немедленного начала, непрерывного выполнения или тесного взаимодействия с пользователем/интерфейсом приложения.

- Стриминг медиа, воспроизведение музыки, аудио/видео звонки;
- Отслеживание местоположения в реальном времени;
- Долгие фоновые операции, требующие общения с UI;
- Критичные задачи, зависящие от действий пользователя прямо сейчас;
- Межпроцессное взаимодействие (IPC).

Сервис vs WorkManager

Когда лучше подойдёт WorkManager: если фоновые задачи могут быть отложены, планируются на определённое время или должны выполняться при определённых условиях, а также когда требуются гарантии их завершения (даже при перезапуске приложения).

- Периодические фоновые задачи без строгой привязки ко времени;
- Откладываемые несрочные задачи;
- Задачи, требующие условий (интернет, зарядка и пр.);
- Последовательные сложные операции.

Практика

Задача 1. Запуск StartedService

Создайте сервис, который пишет в Logcat текущую дату и время каждые 5 секунд.
Запустите сервис по нажатию кнопки в Activity.
Остановите сервис по второй кнопке или автоматически через 20 секунд после запуска.

Задача 2. BoundService

Реализуйте BoundService, возвращающий случайное число методом getRandomNumber().

Привяжитесь к сервису из Activity и покажите число в TextView по нажатию кнопки.

Задача 3. Простая задача в WorkManager

Создайте Worker, который пишет «Привет, я Worker!» в Logcat.
Запустите Worker по нажатию кнопки из Activity.

Домашнее задание

Задача 1. Сервис для скачивания файла

Реализуйте ForegroundService, который скачивает картинку из интернета (используйте любую ссылку на изображение).

Во время скачивания отображайте уведомление с прогрессом.

После завершения скачивания покажите уведомление о завершении с возможностью открыть скачанный файл в галерее устройства.

Задача 2. Синхронизация данных с помощью WorkManager

Создайте задачу в WorkManager, которая раз в сутки загружает JSON-файл с сервера (можете использовать любой тестовый API, например, https://jsonplaceholder.typicode.com/posts/1).

После загрузки сохраните полученные данные в локальные SharedPreferences или в файл.
Запускайте задачу с ограничениями: только Wi-Fi и подключенная зарядка.

Q&A

Ваши вопросы

Спасибо

<TeachMeSkills/>