Why background services are restricted in newer android versions?

Background services are restricted in newer Android versions primarily to improve **battery life**, **performance**, and **user privacy**.

🔋 1. Battery Optimization

- **Background services** can drain battery by running continuously or waking the device frequently.
- Android introduced **Doze Mode** (from Android 6.0) and **App Standby** to limit background activity when the device is idle.
- Later versions (especially Android 8.0 and above) imposed stricter limits on background services to reduce unnecessary power consumption.

🚀 2. Performance Improvements

- Too many background services can slow down the system, especially on devices with limited RAM or processing power.
- By restricting background processes, Android ensures smoother performance and better resource allocation for foreground apps.

🔒 3. User Privacy and Security

- Background services can access sensitive data without the user's knowledge.
- Android now requires **foreground services** (with a visible notification) for tasks like location tracking, ensuring transparency.
- Apps must request **explicit permissions** and justify background access (e.g., background location access).

📱 4. Foreground Service Requirement

- From Android 8.0 (Oreo), apps can no longer start background services while in the background. Instead, they must use:
    - **Foreground services** (with a persistent notification)
    - **JobScheduler**, **WorkManager**, or **AlarmManager** for deferred or periodic tasks

🧰 **5. Encouraging Best Practices**

- Google encourages developers to use modern APIs like:
    - **WorkManager** for background tasks that need guaranteed execution
    - **JobScheduler** for system-optimized background jobs
    - **Broadcast receivers** for event-driven tasks

When should we use work manager and foreground service?

✅ **Use Work Manager when:**

1. **The task is deferrable** (doesn't need to run immediately).
2. **The task is expected to be long-running** or **guaranteed to execute**, even if the app exits or the device restarts.
3. You want to **respect system optimizations** like Doze Mode and Battery Saver.
4. Examples:
    - Uploading logs or analytics data
    - Syncing data with a server
    - Periodic background tasks (e.g., daily backup)
    - Sending notifications at scheduled times

📌 **Bonus**: Work Manager automatically chooses the best way to run the task (JobScheduler, AlarmManager, or FirebaseJobDispatcher) based on the device and API level.

✅ **Use Foreground Service when:**

1. The task is **time-sensitive** and must run **immediately**.
2. The task must **continue even if the user leaves the app**.

3. The task involves **user-visible work**, and you can show a **persistent notification**.
4. Examples:
    - Playing music or media
    - Tracking location in real-time (e.g., navigation, fitness apps)
    - Uploading or downloading large files
    - Bluetooth or USB communication

⚠️ From Android 10+, you must request the FOREGROUND_SERVICE permission and show a notification as soon as the service starts.

🔧 WorkManager's Internal Strategy

WorkManager uses a **backoff strategy** and **internal abstraction layer** to select the most appropriate scheduler. Here's how it works:

✅ 1. JobScheduler (API 23+)

- **Preferred on Android 6.0 (API 23) and above**
- Integrates well with Doze Mode and App Standby
- Handles constraints like network type, charging state, idle state, etc.
- **Most battery-efficient and reliable**

✅ 2. AlarmManager + BroadcastReceiver (API 14–22)

- Used on older devices where JobScheduler is not available
- Less efficient and more prone to being killed by the system
- WorkManager wraps this with retry logic and persistence

✅ 3. Firebase JobDispatcher (Deprecated)

- Was used for backward compatibility on older devices (API < 21)
- Now deprecated in favor of WorkManager itself

⚙️ How It Decides

WorkManager uses a **Scheduler API** internally. When you enqueue a task, it:

1. **Checks the device's API level**
2. **Checks available system services**
3. **Chooses the best available scheduler**:
    - JobSchedulerScheduler for API 23+
    - AlarmManagerScheduler for API 14–22
4. **Persists the task in a local database**, so it survives app restarts or crashes
5. **Executes the task** when constraints are met (e.g., network, charging)

🔁 Summary of How Work Manager Works Internally:

Android Version	Internal Scheduler Used by WorkManager
API 23+	JobScheduler
API 14–22	AlarmManager + BroadcastReceiver
(Previously)	FirebaseJobDispatcher (now deprecated)

(Important)

1. Unified API Across All Android Versions

- **Problem**: JobScheduler only works on API 21+ (Lollipop and above), and AlarmManager or FirebaseJobDispatcher had to be used for older versions.
- **Solution**: WorkManager abstracts this complexity. You write your task once, and it chooses the best available scheduler based on the device's API level.

2. Guaranteed Execution

- WorkManager ensures that your task **will be executed**, even if the app is killed or the device restarts (with constraints like network availability, charging, etc.).
- AlarmManager and JobScheduler don't always guarantee execution under Doze Mode or App Standby.

3. Constraint Support

- WorkManager supports constraints like:
    - Only run on Wi-Fi
    - Only when charging
    - Only when the device is idle
- These are easier to define and manage compared to setting them manually with JobScheduler.

4. Chaining and Complex Workfl ows

- WorkManager allows **chaining tasks**, running them in sequence or in parallel, and handling dependencies between them.
- This is not natively supported by JobScheduler or AlarmManager.

5. Lifecycle Awareness

- WorkManager is **lifecycle-aware**, meaning it can be used safely with LiveData, ViewModel, and other Jetpack components.

How retries work in workmanager?

In WorkManager, retries are handled using a built-in mechanism that allows you to specify how and when a failed task should be retried.

🔁 **Retry Behavior in WorkManager**

1. **Automatic Retry on Failure**:
    - If a Worker returns Result.retry(), WorkManager will automatically retry the task.

- You can control the **backoff policy** and **delay** between retries.

2.Backoff Policy: You can specify how long WorkManager should wait before retrying a failed task using:

- setBackoffCriteria(BackoffPolicy policy, long delay, TimeUnit unit)
- BackoffPolicy.LINEAR or BackoffPolicy.EXPONENTIAL

3.Retry Limit:

- WorkManager automatically retries up to **10 times**.
- After 10 failed attempts, the work is marked as **FAILED**.
- **Maximum Backoff Delay**:
  - The maximum backoff delay is **5 hours** (as of the latest WorkManager versions).
  - After this, retries will not be delayed further.
-

4.Manual Retry Trigger:

- Inside your Worker, you can return Result.retry() to indicate a retry is needed

🧠 When to Use Faster or Custom Retry Logic

If your task is **time-sensitive** (e.g., needs to retry within seconds), WorkManager might not be the best fit. You could:

- Use **coroutines** or **RxJava** with custom retry logic.
- Use **foreground services** for urgent tasks.
- Combine WorkManager with **constraints** (e.g., only run when network is available) to reduce failure chances.

🧠 Notes

- The **minimum interval** for periodic work is **15 minutes**.
- Use enqueueUniquePeriodicWork() to avoid duplicate scheduling.
- Use ExistingPeriodicWorkPolicy.KEEP to keep the existing work or REPLACE to overwrite it.

⚠️ Limitations of PeriodicWorkRequest

1. ⏱️ **Minimum Interval is 15 Minutes**
   - You cannot schedule periodic work more frequently than **every 15 minutes**.
   - This is enforced by the system to optimize battery and performance.
2. 🔁 **No Exact Timing**
   - WorkManager does **not guarantee exact execution time**.
   - The system may delay execution based on battery, Doze mode, or app standby.
3. 📴 **Not Ideal for Real-Time Tasks**
   - It's not suitable for tasks that require **precise or real-time scheduling**, like alarms or reminders.
4. 🧠 **Limited Control Over Execution**
   - You can't specify exact times (e.g., "run at 2:00 PM every day").
   - For that, you'd need to use AlarmManager.
5. 🔄 **No Built-in Support for Cron-like Schedules**
   - You can't define complex schedules like "every Monday and Friday at 9 AM".
6. 🧪 **Difficult to Test**
   - Testing periodic work can be tricky due to the long intervals and system optimizations.
7. 📱 **Affected by Battery Optimizations**
   - On some devices, periodic work may be delayed or skipped due to aggressive battery-saving features.

☑️ android.permission.RECEIVE_BOOT_COMPLETED

- **Purpose**: Allows WorkManager to **reschedule work after a device reboot**.

- **When Required**: If you're using **persistent work** like OneTimeWorkRequest or PeriodicWorkRequest that should survive reboots.