ChatGPT

Modernes Audio-Streaming-Setup in Flutter

Für eine robuste Audio-Streaming- und Wiedergabe-Funktion in Flutter kombinieren wir **just_audio** für die Wiedergabe mit **audio_service** für Hintergrund- und Systemintegration. Dieses Setup unterstützt URL-Streams, Playlists und lokale Dateien sowie Play/Pause, Seek, Shuffle/Repeat und Steuerung über Sperrbildschirm und Benachrichtigungen (Android MediaSession, iOS NowPlayingInfo). Die Steuerung erfolgt über das **BLoC-Pattern** (`flutter_bloc`), eingebettet in eine **Clean-Architecture** und ist testbar (TDD). Folgende Abschnitte zeigen die Architektur, Best Practices und Beispielcode (inklusive Verzeichnisstruktur und Startpunkte).

1. Audio-Framework und Pakete

- **just_audio** (aktuell v0.10.x): Ein *feature-reicher* Audioplayer für Flutter, der Streams, Assets, Dateien, Playlists, Gapless Playback, Shuffle und Loop unterstützt [1] [2]. Allerdings kümmert sich **just_audio** *nur* um die Wiedergabe, nicht um Hintergrundbetrieb oder System-Steuerung.
- **audio_service** (v0.18.x): Ein Wrapper, der Ihren bestehenden Audiocode (z.B. **just_audio**) kapselt, um **Hintergrund-Audio** zu ermöglichen und Standard-Media-Aktionen wie Play/Pause/Seek/Shuffle/Repeat sowie Lock-Screen- und Benachrichtigungs-Steuerungen zu handhaben [3] [4]. Audio_Service erstellt für Android eine MediaSession und für iOS einen NowPlayingInfo-Channel. Beispiel: Durch `AudioService.init(builder: () => MyAudioHandler(), config: AudioServiceConfig(...))` registrieren Sie Ihren AudioHandler beim App-Start [5].
- **audio_session** (v0.2.x): Ergänzend konfiguriert dieses Plugin die *Audio-Session* von iOS/Android (Kategorie, Fokus, Ducking). Best Practice ist, nach allen anderen Audio-Plugins die Session zu konfigurieren (z.B. `await AudioSession.instance.configure(AudioSessionConfiguration.music())` für Musik oder `.speech()` für Podcasts) [6].
- **flutter_bloc**: Implementiert das BLoC/MVVM-Muster in Flutter. Mit `BlocProvider` / `RepositoryProvider` kann man Repositories und Blöcke einspeisen [7]. So ist der Code klar getrennt und leicht testbar.

Zusammen nehmen wir **just_audio** als Wiedergabe-Engine, umgeben von einem **AudioHandler** (audio_service), und steuern alles über BLoC-Events.

2. AudioHandler: Background-Audio mit audio_service

Mit **audio_service** kapseln Sie die Audio-Logik in einem `AudioHandler`. Dieser kommuniziert über standardisierte Methoden (`play`, `pause`, `seek`, `skipToNext`, `setShuffleMode` etc.) mit der App-UI, dem Sperrbildschirm und Zubehör (Headsets, Smartwatches). Ein typischer `AudioHandler`-Beispiel mit just_audio:

```
class MyAudioHandler extends BaseAudioHandler
    with QueueHandler, SeekHandler {
  final _player = AudioPlayer(); // just_audio

  MyAudioHandler() {
    // Hier könnten Sie beim Start vorladen oder Initial-States setzen.
```

```
  }

  @override
  Future<void> play() => _player.play();

  @override
  Future<void> pause() => _player.pause();

  @override
  Future<void> stop() => _player.stop();

  @override
  Future<void> seek(Duration position) => _player.seek(position);

  @override
  Future<void> skipToQueueItem(int index) =>
      _player.seek(Duration.zero, index: index);

  // Zusätzliche Methoden für Shuffle/Repeat:
  @override
  Future<void> setShuffleMode(AudioServiceShuffleMode mode) {
    // just_audio unterstützt Shuffle im Player:
    final enable = mode == AudioServiceShuffleMode.all;
    _player.setShuffleModeEnabled(enable);
    return super.setShuffleMode(mode);
  }

  @override
  Future<void> setRepeatMode(AudioServiceRepeatMode mode) {
    // Mappen von repeat/all/one auf just_audio LoopMode:
    LoopMode loopMode;
    if (mode == AudioServiceRepeatMode.one) loopMode = LoopMode.one;
    else if (mode == AudioServiceRepeatMode.all) loopMode = LoopMode.all;
    else loopMode = LoopMode.off;
    _player.setLoopMode(loopMode);
    return super.setRepeatMode(mode);
  }
}
```

Dieses Beispiel zeigt einen `BaseAudioHandler` mit Mixins `QueueHandler` und `SeekHandler` für Standard-Callbacks [2]. Die `play()`-, `pause()`- etc. Methoden rufen einfach den **just_audio**-Player auf. Shuffle/Repeat setzen wir zusätzlich auf dem Player. Beim Initialisieren rufen wir später `AudioService.init(builder: () => MyAudioHandler(), ...)` im `main()` auf [5], um den Handler zu starten.

MediaItem & Metadaten: Für Sperrbildschirm und Benachrichtigungen ist es wichtig, ein aktuelles `MediaItem` zu setzen. Beispiel:

```
var item = MediaItem(
  id: 'https://example.com/track.mp3',
```

```
  title: 'Track Title',
  artist: 'Artist Name',
  album: 'Album Name',
  duration: Duration(minutes: 3, seconds: 25),
  artUri: Uri.parse('https://example.com/cover.jpg'),
);
_audioHandler.playMediaItem(item);
```

AudioService sendet diese Informationen automatisch an die Systemoberfläche (Android MediaSession bzw. iOS NowPlaying) [8]. Auf ähnliche Weise verwaltet `audio_service` eine Warteschlange (`queue`) und Broadcasts über Streams.

Hintergrundbetrieb: Einmal initialisiert, läuft Ihr AudioHandler in einem Hintergrund-Isolat. Er reagiert auf Medien-Tasten (Play/Pause auf Kopfhörern, Android Auto, Siri/Google-Befehle) automatisch über `audio_service`. Sie müssen nur die Methoden im Handler implementieren. Die Tabelle in der audio_service-Dokumentation zeigt alle unterstützten Features (Lock-Screen, Headset-Buttons, Shuffle, Repeat, Remote Controls) mit Häkchen für Android und iOS [9].

3. AudioService Initialisierung und Android/iOS Setup

Beim App-Start initialisieren Sie `audio_service` typischerweise so:

```
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  // AudioService-Initialisierung (speichern Sie _audioHandler global oder
via DI)
  _audioHandler = await AudioService.init(
    builder: () => MyAudioHandler(),
    config: AudioServiceConfig(
      androidNotificationChannelId: 'com.example.app.audio',
      androidNotificationChannelName: 'Audio Playback',
      androidNotificationOngoing: true,
    ),
  );
  runApp(MyApp());
}
```

Sie können weitere Parameter angeben (Icons, Verbosity, etc.). Wichtig ist hier vor allem die Android-Notification-Channel-Konfiguration [5]. In **AndroidManifest.xml** müssen Sie außerdem Berechtigungen (`WAKE_LOCK`, `FOREGROUND_SERVICE`, `INTERNET`) und `<service>`-Einträge hinzufügen (siehe audio_service-Doku). Für **iOS** fügen Sie in `Info.plist` unter `UIBackgroundModes` den Eintrag `audio` hinzu (erlaubt Hintergrundwiedergabe).

> **Tipp:** AudioService bietet auch eine `JustAudioHandler`-Implementation (im just_audio_handlers Paket), die einige Schritte vereinfacht. Für volle Kontrolle empfiehlt sich jedoch ein eigener Handler wie oben.

Android MediaSession & iOS NowPlaying

AudioService richtet automatisch eine Android **MediaSession** ein und füllt sie mit Ihrem `MediaItem` sowie Status (Play/Pause, Position). Auf iOS verwendet es das **MPNowPlayingInfoCenter** für Sperrbildschirm-Infos. In der Praxis müssen Sie dafür nichts extra tun, außer in Ihrem `MyAudioHandler` regelmäßig `playbackState.add(...)` und `mediaItem.add(...)` zu schicken, damit System und UI aktuelle Informationen haben [10] [11]. Beispiel für Status-Updates im Handler:

```
// Wenn Sie z.B. _player.play() aufrufen, senden Sie vorher:
playbackState.add(playbackState.value.copyWith(
  playing: true,
  controls: [MediaControl.pause],
  processingState: AudioProcessingState.ready,
));
// Bei Pause:
playbackState.add(playbackState.value.copyWith(
  playing: false,
  controls: [MediaControl.play],
));
```

AudioService überträgt diese Status-Streams an Benachrichtigung und Sperrbildschirm [12]. Ihre Flutter-UI kann sich über `_audioHandler.playbackState` subscriben (z.B. via StreamBuilder oder Bloc) und so Icons aktualisieren [13].

4. BLoC für Player-Steuerung und Zustand

Um Interaktion und Logik zu entkoppeln, setzen wir das **BLoC-Pattern** (flutter_bloc) ein. Eine übliche Struktur:

- **Events (PlayerEvent):** `PlayEvent`, `PauseEvent`, `SeekEvent`, `NextTrackEvent`, etc.
- **State (PlayerState):** Enthält aktuellen Status, z.B. `isPlaying`, `position`, `duration`, `currentTrack`, `shuffleOn`, `repeatMode`, etc.
- **Bloc (PlayerBloc):** Nimmt Events entgegen, nutzt ein *AudioRepository* oder direkt den `AudioHandler`, ruft Methoden (`_audioHandler.play()`) auf und emittiert neue States. Zusätzlich hört es auf Streams des AudioHandler, z.B. `playbackState` oder Positions-Updates, um den State fortlaufend zu aktualisieren.

Beispiel (verkürzt):

```
// Events
abstract class AudioEvent {}
class PlayPauseToggle extends AudioEvent {}
class SeekPosition extends AudioEvent {
  final Duration position;
  SeekPosition(this.position);
}

// State
```

```dart
class AudioState {
  final bool isPlaying;
  final Duration position;
  final Duration duration;
  AudioState({this.isPlaying=false, this.position=Duration.zero,
this.duration=Duration.zero});
}

// BLoC
class AudioBloc extends Bloc<AudioEvent, AudioState> {
  final AudioHandler _audioHandler;
  StreamSubscription<PlaybackState>? _stateSub;
  StreamSubscription<MediaItem?>? _itemSub;

  AudioBloc(this._audioHandler) : super(AudioState()) {
    // Event-Handler
    on<PlayPauseToggle>((event, emit) async {
      final playing = state.isPlaying;
      if (playing) await _audioHandler.pause();
      else await _audioHandler.play();
      // Zustand wird später durch StreamListener aktualisiert.
    });
    on<SeekPosition>((event, emit) {
      _audioHandler.seek(event.position);
    });
    // ... weitere Events

    // Hör auf Hintergrund-Streams
    _stateSub = _audioHandler.playbackState.listen((playbackState) {
      final isPlaying = playbackState.playing;
      final position = playbackState.updatePosition;
      final duration = playbackState.processingState ==
AudioProcessingState.ready
        ? playbackState.duration ?? Duration.zero
        : state.duration;
      add(_PlaybackChanged(isPlaying, position, duration)); // internes Event
    });
  }

  @override
  Future<void> close() {
    _stateSub?.cancel();
    _itemSub?.cancel();
    return super.close();
  }

  // Internes Event zum Updaten
  void _onPlaybackChanged(_PlaybackChanged e, Emitter<AudioState> emit) {
    emit(AudioState(isPlaying: e.isPlaying, position: e.position, duration:
e.duration));
```

```
  }
}
```

Wichtig: Über `RepositoryProvider` / `BlocProvider` injizieren Sie den `AudioHandler` oder ein `AudioRepository`, was Tests und Austausch erleichtert [7]. In Tests können Sie dann den Handler mocken und das Bloc-Verhalten prüfen.

5. Beispiel UI mit BLoC

Im UI-Bereich verwenden wir Widgets wie `BlocBuilder` oder `BlocConsumer` aus `flutter_bloc`, um auf Zustandsänderungen zu reagieren. Beispiel für ein Play/Pause-Widget:

```
class PlayerControls extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocBuilder<AudioBloc, AudioState>(
      builder: (context, state) {
        return Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            IconButton(
              icon: Icon(Icons.skip_previous),
              onPressed: () =>
context.read<AudioBloc>().add(PreviousTrackEvent()),
            ),
            IconButton(
              icon: Icon(state.isPlaying ? Icons.pause : Icons.play_arrow),
              onPressed: () =>
context.read<AudioBloc>().add(PlayPauseToggle()),
            ),
            IconButton(
              icon: Icon(Icons.skip_next),
              onPressed: () =>
context.read<AudioBloc>().add(NextTrackEvent()),
            ),
          ],
        );
      },
    );
  }
}
```

Für die Fortschrittsanzeige (Seekbar) könnten wir `audio_video_progress_bar` verwenden oder einen `Slider`, der die `position` / `duration` aus dem State anzeigt. Ein `Timer` oder StreamBuilder kann periodisch `SeekPosition`-Events feuern, oder der Bloc selbst sendet regelmäßig seine Position (siehe oben).

6. Clean Architecture & Verzeichnisstruktur

Für saubere Architektur strukturieren wir den Code entlang der Schichten (Domain, Data, Presentation). Beispiel:

```
lib/
├── core/                  // Allgemeine Utilities, z.B. AudioModels,
Usecases-Interfaces
│   └── models/
│       └── media_item.dart // Eigene Mediendaten (optional)
├── data/
│   └── audio/
│       ├── audio_repository_impl.dart  // Implementierung, nutzt
AudioHandler
│       └── audio_datasource.dart      // z.B. AudioService-Schnittstelle
├── domain/
│   └── audio/
│       ├── entities/       // Domain-Modelle (MediaItem, Track)
│       ├── repositories/   // Abstrakte Interfaces (AudioRepository)
│       └── usecases/       // Anwendungsfälle (PlayUseCase, PauseUseCase
usw.)
├── presentation/
│   └── blocs/             // Alle Blöcke (AudioBloc, ProgressBloc etc.)
│   └── pages/             // Bildschirme (AudioPlayerPage)
│   └── widgets/           // Wiederverwendbare Widgets (Controls, TrackInfo)
├── main.dart              // Startpunkt: DI (BlocProvider,
RepositoryProvider) und MyApp
```

- **Repositories** abstrahieren die Audiologik. Ein `AudioRepositoryImpl` kann z.B. Methoden wie `play()`, `pause()`, `loadPlaylist(List<MediaItem>)` bereitstellen und intern den `AudioHandler` ansprechen. So sind Business-Logik und Storage/Service entkoppelt – ideal für Tests.
- **Use Cases** können konkret definieren, welche Operation ausgeführt wird (z.B. `PlayTrack`, `ShufflePlaylist`).
- **Blocs** werden über `RepositoryProvider` instanziiert und können so in Tests durch Mock-Repositories ersetzt werden [7].
- **Widgets/Pages** kommunizieren nur über BLoC (Dispatch von Events, Lesen von States), und kennen keine Audio-Details.

7. Tests und Best Practices

- **Modultestbarkeit:** Durch die Trennung über Interfaces sind Audio-Logik und UI leicht testbar. Man kann etwa den `AudioHandler` mocken (z.B. mit `mocktail`) und den Bloc gegen erwartete States prüfen.
- **Player-State-Handling:** Verwenden Sie die Streams von `AudioHandler` (insbesondere `playbackState` und `mediaItem`) als einzige Quelle der Wahrheit für den aktuellen Status. BLoC oder Widgets reagieren darauf, statt über lokale Flags zu verwalten [14] [13].
- **Lifecycle:** audio_service hält die Wiedergabe auch bei App-Hintergrund aktiv. Trotzdem sollten Sie Ressourcen freigeben: Im `onStop()` (bei der alten BackgroundAPI) oder beim Beenden der

App `AudioService.disconnect()` aufrufen. Achten Sie darauf, `AudioHandler`-Instanz nicht mehrfach zu initialisieren.
- **Energieverwaltung:** Fügen Sie in Android `WAKE_LOCK` ein und in iOS das `audio`-Flag, damit die Wiedergabe auch bei ausgeschaltetem Bildschirm fortgesetzt wird.
- **TDD:** Schreiben Sie zunächst Tests für Ihre Blöcke und Use Cases, indem Sie das `AudioHandler`-Interface oder -Repository ersetzen. Beispiel: Testen, dass auf `PlayEvent` tatsächlich `_audioHandler.play()` aufgerufen wird.

8. Ressourcen und Beispielprojekte

- **Dokumentation:** Die offiziellen Pub-Seiten bieten ausführliche Beispiele (Just Audio: Features [1]; Audio Service: Grundlagen und Tutorials [2] [8]).
- **Tutorials:** Surag Chincholi hat ein Schritt-für-Schritt-Tutorial (und begleitendes GitHub-Repo) erstellt, das die AudioService-Integration erklärt [15] [16].
- **GitHub-Boilerplates:** Als Startpunkt kann das Repository suragch/flutter_audio_service_demo dienen, das AudioService + just_audio kombiniert. Auch die offizielle AudioService-Beispiel-App (im Repository von ryanheise) zeigt viele Use-Cases.
- **Verlinkte Pakete:** flutter_bloc-Doku (z.B. RepositoryProvider für DI [7]) und audio_session-Beispiel für Audio-Kategorien [6].

Mit dieser Architektur erhalten Sie eine moderne, erweiterbare Audio-Streaming-Lösung für Flutter (3.x+), die alle geforderten Funktionen (Streams, Playlists, lokale Dateien, Shuffle/Repeat, Sperrbildschirm-Steuerung) sauber abbildet und leicht testbar bleibt.

Quellen: Offizielle Dokumentationen und Beispiele zu just_audio und audio_service [1] [17] [2], Einführungen und Tutorials [15] [6].

---

[1] just_audio | Flutter package
https://pub.dev/packages/just_audio

[2] [3] [4] [5] [8] [9] [10] [15] [17] audio_service | Flutter package
https://pub.dev/packages/audio_service

[6] audio_session | Flutter package
https://pub.dev/packages/audio_session

[7] flutter_bloc | Flutter package
https://pub.dev/packages/flutter_bloc

[11] [12] [13] [14] Tutorial · ryanheise/audio_service Wiki · GitHub
https://github.com/ryanheise/audio_service/wiki/Tutorial

[16] GitHub - suragch/flutter_audio_service_demo: Companion project for Flutter audio_service tutorial
https://github.com/suragch/flutter_audio_service_demo