Member-only story

Background audio in Flutter with Audio Service and Just Audio

Suragch · Follow
35 min read · Jul 17, 2021

Listen Share

A full step-by-step tutorial

Article updated on March 18, 2023.

If you're only playing short audio clips in your app, then generally a simple audio player plugin will work. However, for longer audio forms like music or podcasts, this is often insufficient. Let's say you start playing a podcast but then leave this app to check your fitness app. What does the podcast app do? It could stop the audio as soon as you leave, but you want to keep listening, so that's not a good option. On the other hand, if the app allows the podcast to keep playing in the background, imagine the second scenario: You're happily exercising and listening to the podcast when all of a sudden someone knocks on the door. Where's the pause button? How do you stop the podcast!? Knock, knock, knock! Argh! There's no stop button! You finally navigate back to the podcast app and find a way to stop the audio. How annoying!

I should know since I made an app like that once.

The Flutter **audio_service** plugin solves the problems described above by allowing users to listen to audio outside of the app and also by giving users access to system-wide audio control buttons. On Android these buttons are located in the notification drawer and the lock screen. On iOS they are located in the control center and notification center.

Audio controls in the Android Notification Drawer and iOS Control Center (image sources: Android Police and iOS docs)

Internally audio_service does all the work to interface with Android, iOS, and other platforms so that you don't need to worry about platform-related details (though some features are platform specific).

In this tutorial, I'll walk you through the steps of setting up an audio player that supports background audio. Audio Service isn't an audio player; it's just an interface to the system audio controls. For the audio player itself, you'll need another plugin. In this tutorial I'll use **just_audio**, which is also by Ryan Heise, the author of audio_service. If you aren't familiar with just_audio, I recommend you check out my previous tutorials on the topic, especially the third one:

- *Playing short audio clips in Flutter with Just Audio*
- *Streaming audio in Flutter with Just Audio*
- *Managing playlists in Flutter with Just Audio*

This tutorial will take the same basic app from the third article above and add Audio Service to it. Here is what the final result will look and sound like:

When I was first was learning how to use Audio Service, it was a non-trivial endeavor to get everything working. The documentation has improved since then, but this tutorial will go a little farther by taking you step by step through everything related to adding Audio Service and the Just Audio audio player to your app.

This article is *long*. It's not meant for casual reading. It's meant to guide you through creating the demo app yourself. If you're following along it'll probably take you at least an hour to finish. But by the end of that hour you'll know enough to start using background audio in your own apps.

This tutorial is up to date for:

```
audio_service 0.18.9
just_audio 0.9.31
Flutter 3.7
Dart 2.19
```

The demo project was tested and runs on:

```
Android API 31 emulator
iOS 15.5 simulator
macOS Monterey
Chrome web
```

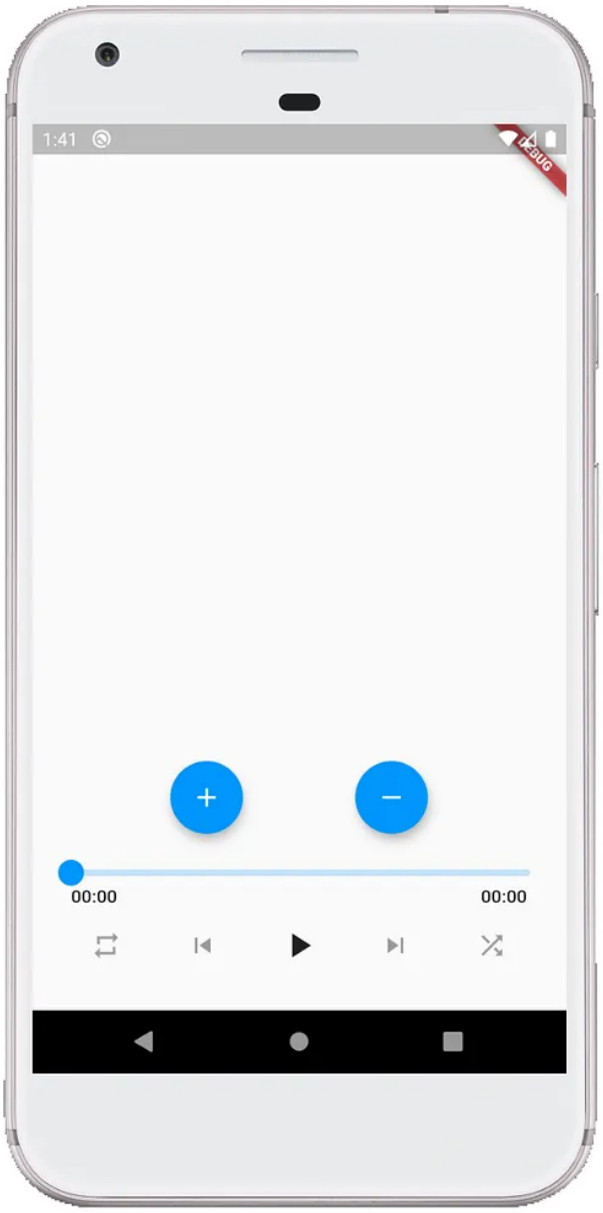
Setup

Rather than walking you through the entire UI and state management setup process, I'll provide you with a basic starter project.

Download or clone the GitHub repository for the project:

```
git clone https://github.com/suragch/flutter_audio_service_demo.git
```

Run the **starter** project and you should see the following:



Starter app

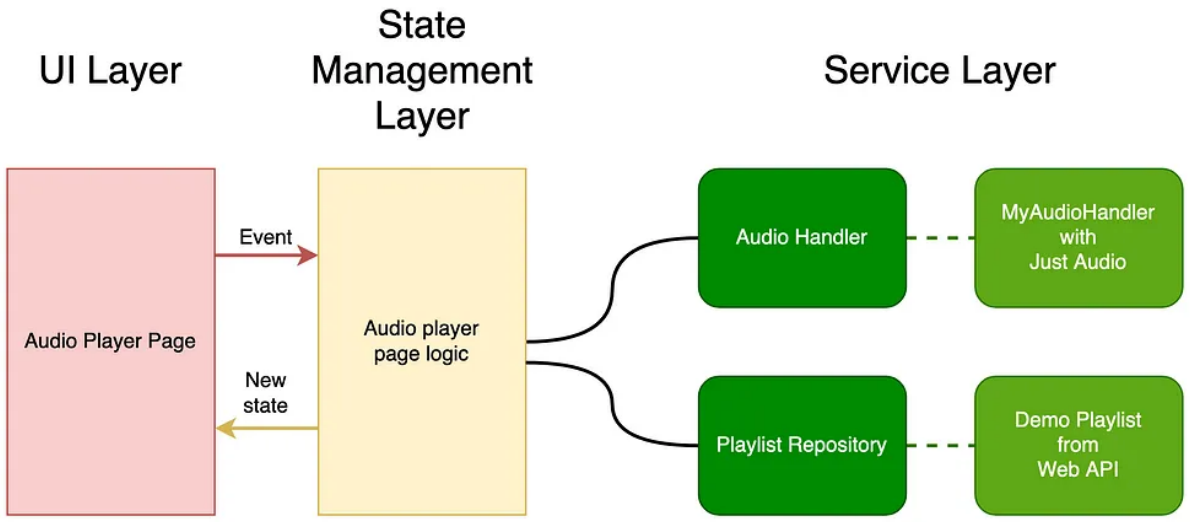
The UI is all ready to send events and react to state updates. However, since you haven't added the audio player yet, the app does nothing.

Most of the widgets are standard Flutter widgets except for the audio progress bar, which uses the audio_video_progress_bar package (my own). That custom widget is a single file so feel free to copy the source code to your project and modify it if you don't like the standard behavior.

Architecture

Even though this is just a tutorial, I sought to use a realistic architectural pattern that you could use in a production app. This is a simple state management solution I've described previously in *Flutter state management for minimalists*. You're welcome to switch it out with Provider or Bloc or something else.

Take a look at the basic app architecture:



Here are some high-level notes before we dive into the implementation details:

- The **UI layer** is already finished. That's not what this tutorial is about. But you should take a little time to browse the widgets. They use `ValueListenableBuilder` widgets to update the UI when there's a state change.
- The **state management layer** will communicate between the UI layer and the service layer. The `ValueNotifier` objects that the UI layer is listening to are already finished. However, there are a bunch of empty UI event methods that you'll fill in as you progress through the tutorial.
- The **service layer** uses the `GetIt` service locator to provide the state management layer with references to the app services. This is mostly already set up. Read *Flutter state management for minimalists* to learn more about how I use it.
- The **Audio Handler** belongs to the audio_service plugin. This (along with the state management layer) is where you'll do most of your work. You'll create a custom audio handler that will wrap a just_audio audio player.
- The **Playlist Repository** is already finished. It grabs songs from the web to play in the app. The implementation is fairly simple, so check it out to familiarize yourself with how it works.

Now it's time to get to work!

Adding the dependencies

Open **pubspec.yaml** and add the dependencies for just_audio and audio_service:

```
just_audio: ^0.9.31
audio_service: ^0.18.9
```

The audio player comes from just_audio, but if you'd like to use a different audio player in the future, that's fine. The audio_service plugin can wrap any audio player that doesn't interfere with the background service tasks that audio_service needs to perform.

Preparing the platforms

Audio Service and Just Audio handle most of the platform-related code for you, but some of it you need to do yourself. I'll include directions for each platform, but feel free to skip the directions for any platform that you're not supporting or don't care about right now.

Android

The directions in this section all relate to **AndroidManifest.xml**, which can be found in **android/app/src/main**.

Permissions

Open **AndroidManifest.xml** and add the following two permissions above the `application` section. These are required by audio_service:

```
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
```

Wake lock prevents the device from sleeping, which presumably you wouldn't want for long-playing audio. Foreground services are like background services that the user is aware of because they need to show some sort of notification icon, which audio_service does.

Since just_audio will stream songs from the internet, you also need to add the following permission:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Activity name

Next find the line that says `android:name=".MainActivity"` and replace it with the following:

```
android:name="com.ryanheise.audioservice.AudioServiceActivity"
```

This is required by the audio_service plugin.

***Note:** You could also provide your own custom Android activity as long as it performs the same basic tasks that the default one does. If you ever need to do that, you can check out the source code for `AudioServiceActivity` to see what you need to implement.*

Intents

After the end of the `</activity>` section and before the end of the `</application>` section, add the following service and receiver intents:

```
<service
    android:name="com.ryanheise.audioservice.AudioService"
    android:foregroundServiceType="mediaPlayback"
    android:exported="true"
    tools:ignore="Instantiatable">
  <intent-filter>
    <action android:name="android.media.browse.MediaBrowserService" />
  </intent-filter>
</service>

<receiver
    android:name="com.ryanheise.audioservice.MediaButtonReceiver"
    android:exported="true"
    tools:ignore="Instantiatable">
  <intent-filter>
    <action android:name="android.intent.action.MEDIA_BUTTON" />
  </intent-filter>
</receiver>
```

These are required by audio_service for the MediaBrowserService and handling media button input.

If your AndroidManifest is giving you an error about `tools`, make sure you add it the the beginning of the file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    ...>
```

And if you get an error about Multidex, check this solution out.

iOS

Open **Info.plist**, which can be found in **ios/Runner**, and the following key-value pair:

```
<key>UIBackgroundModes</key>
<array>
  <string>audio</string>
</array>
```

This tells iOS that you want permission to use the audio background mode, which audio_service needs.

Under the hood Just Audio uses `AVAudioSession` to play audio, but this includes some recording APIs. Since our app doesn't actually record anything, you should strip out those APIs. Otherwise iOS will require you to declare it with a message telling the user why you need microphone permission. To strip out the unneeded APIs, open **Podfile** in the **ios/Runner** folder and scroll down to the bottom to find the **post_install** block. Then add the following code to the end of the innermost loop:

```
target.build_configurations.each do |config|
  config.build_settings['GCC_PREPROCESSOR_DEFINITIONS'] ||= [
    '$(inherited)',
    'AUDIO_SESSION_MICROPHONE=0'
  ]
end
```

The **post_install** block should look like this now:

```
post_install do |installer|
  installer.pods_project.targets.each do |target|
    flutter_additional_ios_build_settings(target)
    # Remove microphone APIs
    target.build_configurations.each do |config|
      config.build_settings['GCC_PREPROCESSOR_DEFINITIONS'] ||= [
        '$(inherited)',
        'AUDIO_SESSION_MICROPHONE=0'
      ]
    end
  end
end
```

That's all you need to do for iOS.

macOS

The minimum version that audio_service supports is macOS 10.12.2, so open **macos/Podfile** and make sure the first line is no lower than the following:

```
platform :osx, '10.12.2'
```

If it was already higher (like `10.14`), it should be fine. No need to change that.

You also need to ask for permission to access the internet for the audio files that just_audio will be playing. Add the following key-value pair to both **DebugProfile.entitlements** and **Release.entitlements**, which you can find in **macos/Runner**.

```
<key>com.apple.security.network.client</key>
<true/>
```

Other platforms

No special setup is needed for the web. Some browsers support media notifications, which audio_service gives you, so this can be useful on a mobile browser.

Windows and Linux are also supported but I haven't tested those platforms, so check the documentation for setup instructions.

Creating an audio handler

The audio_service plugin contains an abstract class called `AudioHandler`, which defines a bunch of standard methods like `play`, `pause`, `seek`, etc. You'll implement your own logic for these methods in a class called `MyAudioHandler`, but the outside world (i.e., the rest of the app) will only know about `AudioHandler`. This allows you to completely encapsulate your audio functionality in a single location. In this tutorial your audio player will be Just Audio, but if you wanted to switch it out with a different audio player later, you would only need to make the changes in one place, not all throughout your app.

Create a new file in **lib/services** called **audio_handler.dart**. Then paste in the following code:

```
import 'package:audio_service/audio_service.dart';

Future<AudioHandler> initAudioService() async {
  return await AudioService.init(
    builder: () => MyAudioHandler(),
    config: const AudioServiceConfig(
      androidNotificationChannelId: 'com.mycompany.myapp.audio',
      androidNotificationChannelName: 'Audio Service Demo',
      androidNotificationOngoing: true,
      androidStopForegroundOnPause: true,
    ),
  );
}

class MyAudioHandler extends BaseAudioHandler {
  // TODO: Override needed methods
}
```

Notes:

- The method `initAudioService()` is where you create the audio handler for your app. You'll call this method in the next section.
- You can pass in some configuration settings with `AudioServiceConfig`. Mostly this is for Android related settings. You should read the source code for that file to see what all of the options are.
- The `androidNotificationChannelId` is used in the phone settings and the audio_service documentation recommends that you set this value. If you change it later the old value will still appear in the phone settings.
- The `androidNotificationChannelName` parameter is used for the display name in the Android notification settings, so you should use whatever name you want to display

there.

- Personally I find it annoying when I can't swipe away the audio controls in the notification drawer when the audio is paused, so to make the notifications dismissable, I set both `androidNotificationOngoing` and `androidStopForegroundOnPause` to `true`. The downside is that this makes it easier for the system to dismiss your app when paused as well.
- One more config option you should consider setting for a production app (that we don't do here) is `androidNotificationIcon`. This sets the little notification icon in the status bar. If you don't set it, the launcher icon will be used, which works but probably wouldn't look as good that small as a custom icon would.

Notification icons (Image source: Android docs)

Initializing audio service

In the last section you defined an `initAudioService()` method that will create your audio handler, but you need to actually call that method now. The audio service docs have you initialize the audio handler in `main()` before `runApp`, but since we are already setting up our service locator at that location, you can just call `initAudioService()` in the service locator setup method.

For the reader's reference, this is what `main()` currently looks like in **main.dart**:

```
void main() async {
  await setupServiceLocator();
  runApp(MyApp());
}
```

Now open **service_locator.dart** in the **lib/services** folder and add the following two imports:

```
import 'package:audio_service/audio_service.dart';
import 'audio_handler.dart';
```

Then add the following line at the beginning of `setupServiceLocator()`:

```
getIt.registerSingleton<AudioHandler>(await initAudioService());
```

This will call `initAudioService()`, which will give you an `AudioHandler` instance, which in turn will be registered as a singleton with `GetIt` so that you can access it from anywhere in your app. That will be useful next.

Getting a reference to AudioHandler

Open **page_manager.dart** and have a look at all of the notifiers and methods.

```
class PageManager {
  // Listeners: Updates going to the UI
  final currentSongTitleNotifier = ValueNotifier<String>('');
  final playlistNotifier = ValueNotifier<List<String>>([]);
  final progressNotifier = ProgressNotifier();
  final repeatButtonNotifier = RepeatButtonNotifier();
  final isFirstSongNotifier = ValueNotifier<bool>(true);
  final playButtonNotifier = PlayButtonNotifier();
  final isLastSongNotifier = ValueNotifier<bool>(true);
  final isShuffleModeEnabledNotifier = ValueNotifier<bool>(false);

  // Events: Calls coming from the UI
  void init() {}
  void play() {}
  void pause() {}
  void seek(Duration position) {}
  void previous() {}
  void next() {}
  void repeat() {}
  void shuffle() {}
  void add() {}
  void remove() {}
  void dispose() {}
}
```

The event methods are empty currently and you'll implement them all through the course of the tutorial. For most of them, all you need to do is pass the method call on to the next layer, that is, on to `AudioHandler`. For that you need to get a reference to `AudioHandler`.

Add the following two imports at the top of the file:

```
import 'package:audio_service/audio_service.dart';
import 'services/service_locator.dart';
```

And then add the following member variable to the `PageManager` class:

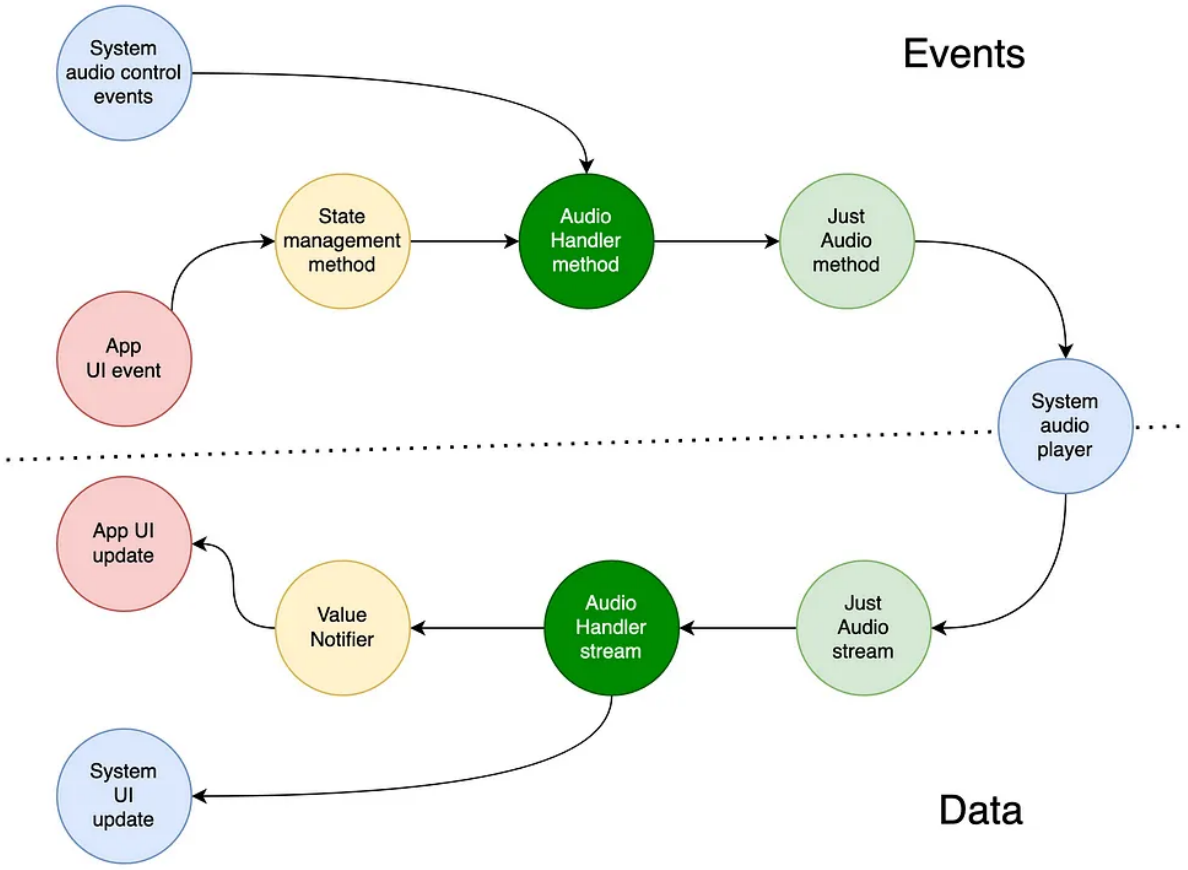
```
final _audioHandler = getIt<AudioHandler>();
```

This uses your service locator GetIt to get a reference to your `AudioHandler` implementation, which behind the scenes is `MyAudioHandler`, but your state management class doesn't know that. It only knows that the `_audioHandler` object is of type `AudioHandler`.

A little more background

You're going to be jumping into the implementation details of all of the audio functions in just a minute, but before you get to that I think it would be helpful to present a little more background on the architecture of how events and data will flow throughout your app.

Take a look at the following diagram:



Event and data flow

When some event like pressing a button happens in the app, this event is passed through the layers as successive method calls until it finally reaches the system audio player. Since the event also causes a change in state, data about the state change is passed back through the layers in the form of a stream or notifiers. Finally the data makes it back to the UI where it's used to update how the UI looks.

Let's take the specific example of pressing the play button in your app or in the system audio controls. Pressing the play button, which is in the UI layer, will call the `play()` method in the state management class. From there it's your job to call the `play()` method in the `AudioHandler` class. In your `MyAudioHandler` implementation you need to call the `play()` method in Just Audio's `AudioPlayer` class. Just Audio will in turn tell the system audio player to start playing the audio. The playing state has changed now and Just Audio will provide this data as a `PlayerState` stream. `MyAudioHandler` will listen to that stream and return its own `PlaybackState` stream. (Actually we're going to do it a little differently, but you could do it this way.) Your state management layer listens to the data stream coming from the audio handler and then notifies your app UI. The app UI responds by changing the play button into a pause button, ready and waiting for you to press it and trigger the next UI event round.

In summary, for every UI event in your app, you'll need to implement the code to pass that event on to the audio handler, and from the audio handler to Just Audio. In addition to that, you'll need to listen to data streams from Just Audio and AudioHandler so that you can update the value notifiers that the UI layer is listening to.

It's kind of a lot of running around, but I wanted to try to give you a high-level view first so that hopefully you won't be so lost when we get to the implementation details. Or maybe you're already lost, in which case I hope the implementation details will make things a little more clear for you.

At any rate, let's dive in.

Adding Just Audio to AudioHandler

Before you start implementing the various audio functions, you'll add Just Audio to your audio handler class.

Open **audio_handler.dart** in the **lib/services** folder and add the just_audio import at the top of the file:

```
import 'package:just_audio/just_audio.dart';
```

Then replace the empty `MyAudioHandler` class with the following code:

```
class MyAudioHandler extends BaseAudioHandler {
  final _player = AudioPlayer();
  final _playlist = ConcatenatingAudioSource(children: []);

  MyAudioHandler() {
```

```
    _loadEmptyPlaylist();
  }

  Future<void> _loadEmptyPlaylist() async {
    try {
      await _player.setAudioSource(_playlist);
    } catch (e) {
      print("Error: $e");
    }
  }
}
```

Notes:

- You'll use `_player` and `_playlist` throughout the class to manage Just Audio. If you aren't familiar with them, review my previous article *Managing playlists in Flutter with Just Audio*.
- I've broken `_loadEmptyPlaylist()` out into its own method to keep the `MyAudioHandler` constructor clean. You'll be adding more methods to the constructor as the tutorial progresses.

Loading an initial playlist

The first UI event to happen is loading a playlist. In this case the event is triggered simply by starting the app.

Notifying AudioHandler about a new playlist

Open **page_manager.dart** and find the empty `init()` method. I've set up the UI layer so that it calls this method when the app first starts, so this is a good place to load the songs you want to add to your playlist.

Replace `init()` with the following code:

```
void init() async {
  await _loadPlaylist();
}

Future<void> _loadPlaylist() async {
  final songRepository = getIt<PlaylistRepository>();
  final playlist = await songRepository.fetchInitialPlaylist();
  final mediaItems = playlist
      .map((song) => MediaItem(
            id: song['id'] ?? '',
            album: song['album'] ?? '',
            title: song['title'] ?? '',
            extras: {'url': song['url']},
          ))
      .toList();
  _audioHandler.addQueueItems(mediaItems);
}
```

I'm breaking `_loadPlaylist()` out into its own method because you're going to add some other methods to `init()` later.

Here are a few things to pay attention to about `_loadPlaylist()`:

- I implemented `fetchInitialPlaylist()` to return a list of three maps, where each map represents one song. Generally in a repository like that, you would query a web API and get JSON back, which you would then convert into a Dart map before passing it back here. Since that isn't the main focus of this tutorial, I just returned a map directly. Check out the source code if you're curious about it.
- `AudioHandler` requires you to put metadata about songs in `MediaItem` objects, so that means you have to convert your list of maps into a list of `MediaItem` objects.
- Playlists in `AudioHandler` are called **queues**, so when you add songs to the playlist you should call `addQueueItems()` to let `AudioHandler` know about the change.

You called `addQueueItems()` above, but this method doesn't actually do anything yet. You need to implement it yourself, which you'll do next.

Handling new queue items in AudioHandler

`AudioHandler` has two main jobs. One is to manage your audio player, which in our case is Just Audio. The other job is to notify the system about audio events. Each time you come to `AudioHandler` you need to remember these two tasks. It's your job to implement them.

`MyAudioHandler` extends `BaseAudioHandler`, which contains a default implementation of `addQueueItems()`. However, the default implementation does nothing, so you need to implement the logic yourself.

***Note**: There is a `QueueHandler` mixin with an `addQueueItems()` implementation that you can use if you like. However, I find it easier to just implement the logic myself because the mixin still doesn't help you with Just Audio and it's confusing to have to call `super`. You can always browse the code in `QueueHandler` and copy it to your own implementation when needed.*

Add the following code inside `MyAudioHandler`:

```
@override
Future<void> addQueueItems(List<MediaItem> mediaItems) async {
  // manage Just Audio
  final audioSource = mediaItems.map(_createAudioSource);
  _playlist.addAll(audioSource.toList());

  // notify system
  final newQueue = queue.value..addAll(mediaItems);
  queue.add(newQueue);
}
```

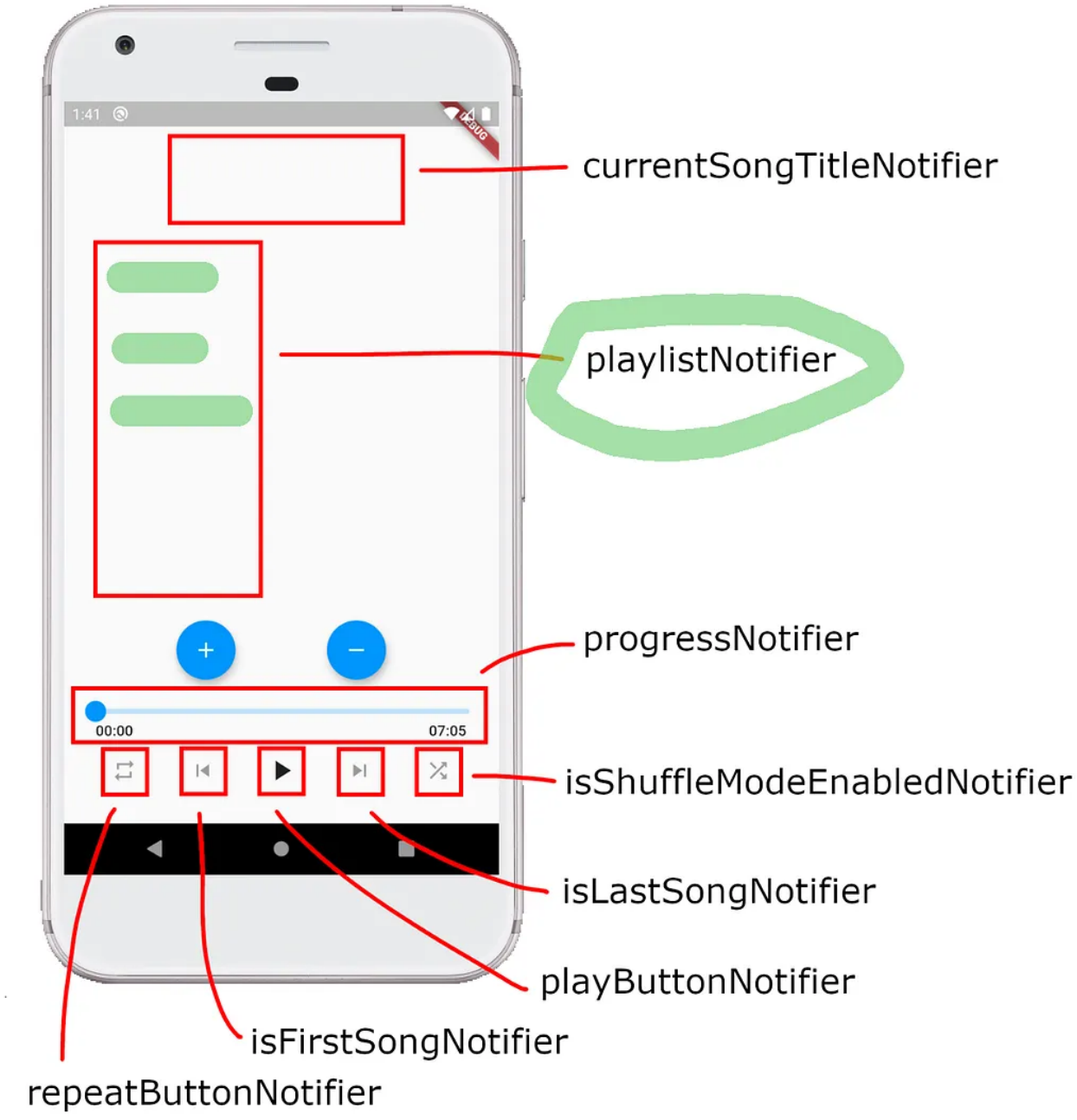
```
UriAudioSource _createAudioSource(MediaItem mediaItem) {
  return AudioSource.uri(
    Uri.parse(mediaItem.extras!['url'] as String),
    tag: mediaItem,
  );
}
```

Remember that you have two tasks to accomplish every time you come to your `AudioHandler` implementation: manage your audio player and notify the system about changes. Here are some more notes about each of those:

- `AudioHandler` uses `MediaItem` objects, but Just Audio needs `AudioSource` objects. That means you have to do yet another conversion before you can add the songs to the playlist. `AudioSource` objects do have a `tag` parameter, though, which can store any object, so you'll use this to keep a copy of your `MediaItem` objects. This will be useful later when you implement shuffling. If you don't need shuffling support then you can probably skip adding the `tag` parameter.
- The way to notify AudioHandler about changes to the playlist is to call `queue.add()`. The `queue` is a `BehaviorSubject` from RxDart and is basically just a glorified stream. When you add a new playlist, any listeners to the `queue` stream are notified. The audio_service plugin will take care of notifying the underlying platform and you can listen to `queue` yourself to update the UI, which you'll do next.

Listening for playlist updates from AudioHandler

What you want to do now is listen to any changes in the playlist queue from `AudioHandler`. When those happen you want to update the UI with the current playlist by updating `playlistNotifier`. See the green highlights in the following image for clarification:



`playlistNotifier` tells the UI when it should show a new list of songs.

Open **page_manager.dart** and add the following method to `PageManager`:

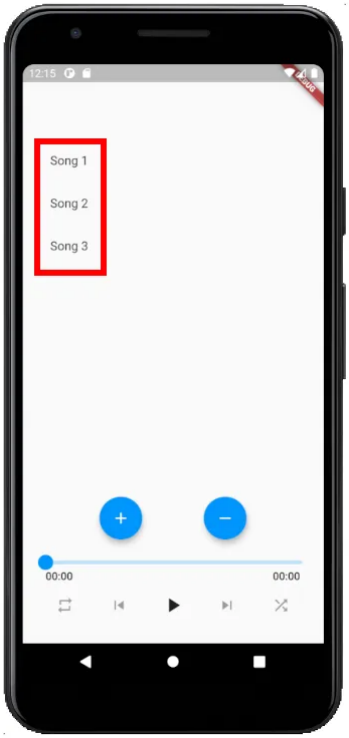
```
void _listenToChangesInPlaylist() {
  _audioHandler.queue.listen((playlist) {
    if (playlist.isEmpty) return;
    final newList = playlist.map((item) => item.title).toList();
    playlistNotifier.value = newList;
  });
}
```

The UI wants a list of song titles, which is what `playlistNotifier` provides. You get this by listening to the audio handler `queue` stream and extracting the title from each `MediaItem` object.

Now call the method you just created at the end of the `init()` method:

```
_listenToChangesInPlaylist();
```

Run the app now to see what you have:



There you go! The UI shows the song list.

One round down. A few more to go. You're going to be a pro by the time you finish this tutorial. Just keep repeating this pattern for every audio feature you want to add:

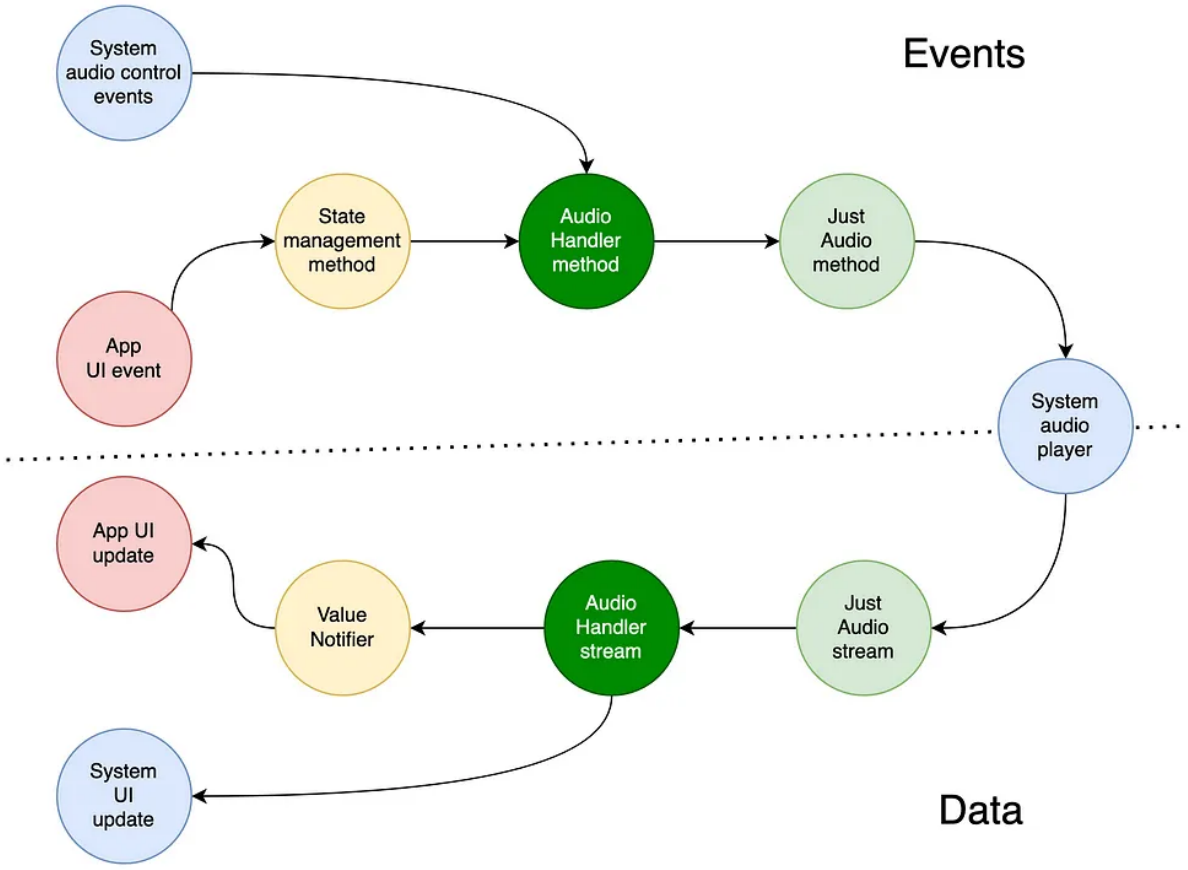
1. notify `AudioHandler` and Just Audio about UI events and then
2. listen for audio data updates.

Next let's implement the play/pause button.

Playing and pausing audio

You've already loaded an initial playlist, so when a user presses the play button, you want the audio to start playing. Also, the play button should turn into a pause button that's ready for the user to pause the audio.

Take another look at the data flow diagram again for a reminder of where we are. You need to take the button press events coming from the UI to the state management layer and pass them on to the audio handler layer. Then you need to listen to the data stream coming back from the audio player.



Event and data flow

Passing on events from state management to AudioHandler

Open **page_manager.dart** and find the empty `play()` and `pause()` methods. The UI is already set up to call them when the user presses the play/pause button.

Since you're only forwarding the calls, there isn't much to do. Replace `play()` and `pause()` with the following:

```
void play() => _audioHandler.play();
void pause() => _audioHandler.pause();
```

Passing on events in AudioHandler

Now you're basically going to just repeat that at the AudioHandler layer. Open **audio_handler.dart** and add the following two overrides:

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

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

The `_player` is the Just Audio `AudioPlayer`.

As you recall, though, you have two jobs that you need to perform for every event that you give to `AudioHandler`. In addition to managing the audio player, you also need to tell `AudioHandler` to notify the system.

When the playlist changed, you notified the system by adding the new playlist to `queue`. However, with play and pause, you're not changing the playlist so you don't need to notify `queue`. What you are doing is changing the playback state, and there's a special `playbackState` stream for that.

Now, you *could* add the new playback state to the `playbackState` stream directly in the `play` and `pause` methods like so:

```dart
@override
Future<void> play() async {
  playbackState.add(playbackState.value.copyWith(
    playing: true,
    controls: [MediaControl.pause],
  ));
  await _player.play();
}

@override
Future<void> pause() async {
  playbackState.add(playbackState.value.copyWith(
    playing: false,
    controls: [MediaControl.play],
  ));
  await _player.pause();
}
```

This is what the official tutorial shows and is also similar to how we updated `queue` in `addQueueItems`. However, we aren't going to do that. Instead we'll take a little different approach that will help us knock out a whole bunch of updates types all at once.

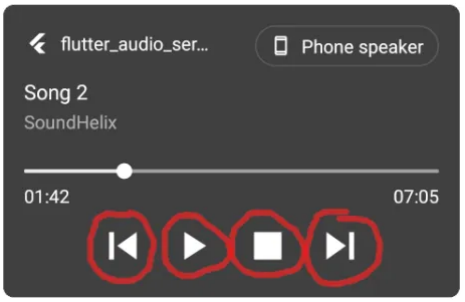
The Just Audio audio player has a playback event stream that we can listen to and pass on to the audio handler. Add the following method to `MyAudioHandler`:

```dart
void _notifyAudioHandlerAboutPlaybackEvents() {
  _player.playbackEventStream.listen((PlaybackEvent event) {
    final playing = _player.playing;
    playbackState.add(playbackState.value.copyWith(
      controls: [
        MediaControl.skipToPrevious,
        if (playing) MediaControl.pause else MediaControl.play,
        MediaControl.stop,
        MediaControl.skipToNext,
      ],
      systemActions: const {
        MediaAction.seek,
      },
      androidCompactActionIndices: const [0, 1, 3],
      processingState: const {
        ProcessingState.idle: AudioProcessingState.idle,
        ProcessingState.loading: AudioProcessingState.loading,
        ProcessingState.buffering: AudioProcessingState.buffering,
        ProcessingState.ready: AudioProcessingState.ready,
        ProcessingState.completed: AudioProcessingState.completed,
      }[_player.processingState]!,
      playing: playing,
      updatePosition: _player.position,
      bufferedPosition: _player.bufferedPosition,
      speed: _player.speed,
      queueIndex: event.currentIndex,
    ));
  });
}
```

Rather than just listening to the playing state, you're listening to all kinds of different state changes that are encapsulated in Just Audio's `PlaybackEvent`. You then take all that state and pass it on to `AudioHandler`'s `playbackState` stream. Doing it all at once like this will save you from having to update each one individually for every audio feature you want to add later.

A few of those parameters are a little confusing, so let's go over their meaning:

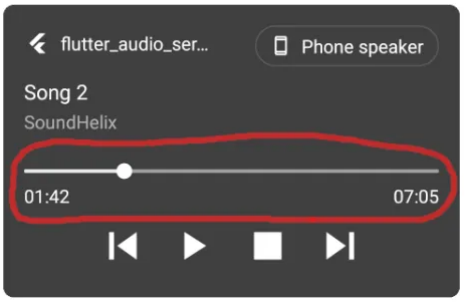
- **controls**: These are the audio buttons that you want displayed in the notification panel.



Audio controls (skipToPrevious, play, stop, skipToNext)

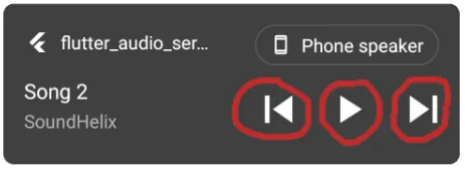
- **systemActions**: This seems like it could have been included with `controls`, but these are actions that don't have a specific button themselves. For example,

`MediaAction.seek` shows a seek bar.



Media action (seek)

- **androidCompactActionIndices**: These refer to items in the control list that you want to show in Android's compact notification view. Since there are four controls listed above (`skipToPrevious` , `play/pause` , `stop` , `skipToNext`), `[0, 1, 3]` refers to `skipToPrevious` , `play` (or `pause`), and `skipToNext` . The `stop` control is at index `2` and is excluded here.



Compact notification view on Android

- **processingState**: Here you are just doing a direct mapping from the Just Audio enum to the Audio Service enum. The UI in the starter project is already set up to show a circular progress indicator when the processing state is loading or buffering.

***Note**: In this tutorial, we aren't adding functionality to change the playback speed, so you could remove the line `speed: _player.speed,` if you like. The idea is that you only need to update the UI with the playback events that matter to your app. Later on we'll be adding one more for `repeatMode` . It's currently missing here but is a function that we'll be supporting.*

Add the method you just created to the `MyAudioHandler()` constructor body (right after `_loadEmptyPlaylist`):

```
_notifyAudioHandlerAboutPlaybackEvents();
```

Now your app will start listening to playback events as soon as `AudioHandler` is initialized.

Updating the UI in response to playback state changes

Next, in the state management layer you need to listen to that playback state data stream. Open **page_manager.dart** and add the following method:

```
void _listenToPlaybackState() {
  _audioHandler.playbackState.listen((playbackState) {
    final isPlaying = playbackState.playing;
    final processingState = playbackState.processingState;
    if (processingState == AudioProcessingState.loading ||
        processingState == AudioProcessingState.buffering) {
      playButtonNotifier.value = ButtonState.loading;
    } else if (!isPlaying) {
      playButtonNotifier.value = ButtonState.paused;
    } else if (processingState != AudioProcessingState.completed) {
      playButtonNotifier.value = ButtonState.playing;
    } else {
      _audioHandler.seek(Duration.zero);
      _audioHandler.pause();
    }
  });
}
```

This just converts the `AudioProcessingState` enum from Audio Service to a custom enum I made called `ButtonState` . (I didn't reuse `AudioProcessingState` because I wanted to keep Audio Service out of my UI code.) When the playlist is finished (i.e., `AudioProcessingState.completed`), the audio handler will seek to the beginning and pause. This is getting a little ahead of ourselves, though, because we haven't implemented the `seek` method yet. That will come soon.

Add the method you just created to the bottom of `init()` :

```
_listenToPlaybackState();
```

Now restart the app and notice the circular progress indicator showing the loading and buffering state. Press the **play/pause** button to confirm that it works.

Play/pause button working

Hooking up the progress bar

The progress bar is used for showing the current playing time and also for seeking to a new location. Seeking is a UI event that you need to pass on to `AudioHandler` and the audio player. The audio progress, buffering, and song length are data that you need to get from the audio player. As usual you'll start by handling the UI event and finish by listening for data updates.

Passing seek events from state management to AudioHandler

Open **page_manager.dart** and find the empty `seek()` method. Replace it with the following:

```
void seek(Duration position) => _audioHandler.seek(position);
```

That's all you need to do here.

Passing seek events from AudioHandler to Just Audio

Open **audio_handler.dart** and add the following override:

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

That's all you need to do to tell the audio player to start playing at a new position.

For most events you also need to tell `AudioHandler` to update the system as well, but for seeking you don't. There's an `AudioService` class that produces continuous audio position updates and it apparently doesn't need our help.

Listening for duration changes from Just Audio

In order to hook up the progress bar, you'll need the total length of the audio. The problem is, though, that when you first load the audio you often don't know how long it is. That means the `duration` property of `MediaItem` is `null`. You can't get the duration until Just Audio has had a chance to load the file.

Once you know the duration from Just Audio, you need to update the `MediaItem` with the duration. And if you change a `MediaItem`, you also need to update the playlist `queue`. It's kind of a pain, but here is how you do that:

Still in **audio_handler.dart**, add the following method:

```
void _listenForDurationChanges() {
  _player.durationStream.listen((duration) {
    final index = _player.currentIndex;
    final newQueue = queue.value;
    if (index == null || newQueue.isEmpty) return;
    final oldMediaItem = newQueue[index];
    final newMediaItem = oldMediaItem.copyWith(duration: duration);
    newQueue[index] = newMediaItem;
    queue.add(newQueue);
    mediaItem.add(newMediaItem);
  });
}
```

You've seen how to add a new queue to the `queue` stream before, but now you are also adding a new `MediaItem` to the `mediaItem` stream. The `mediaItem` stream updates `AudioHandler` with the current media item.

Now add that method to the bottom of the `MyAudioHandler()` constructor body:

```
_listenForDurationChanges();
```

Updating the UI progress bar

The UI progress bar needs to know the current audio position, the buffered position, and the total audio duration of the current song. That means there are three different things to listen to in the state management layer.

Open **page_manager.dart**.

To update the progress bar with the **current audio position**, add the following method to `PageManager`:

```
void _listenToCurrentPosition() {
  AudioService.position.listen((position) {
    final oldState = progressNotifier.value;
    progressNotifier.value = ProgressBarState(
      current: position,
      buffered: oldState.buffered,
      total: oldState.total,
    );
  });
}
```

Usually most of your interaction is with `AudioHandler`. However, when you want to get the current streaming audio position that is updated multiple times a second, you need to listen to the `position` stream from `AudioService`.

To update the **buffered position**, add the following method to `PageManager`:

```
void _listenToBufferedPosition() {
  _audioHandler.playbackState.listen((playbackState) {
    final oldState = progressNotifier.value;
    progressNotifier.value = ProgressBarState(
      current: oldState.current,
      buffered: playbackState.bufferedPosition,
      total: oldState.total,
    );
  });
}
```

You might not recall, but back when you implemented the play/pause notification for `AudioHander`, you also set the buffered position. Look back at `_notifyAudioHandlerAboutPlaybackEvents()` and you'll see the `bufferedPosition` parameter that you set when updating the playback state. You're retrieving that value now.

Still in **page_manager.dart**, add the following method `PageManager` to update the progress bar notifier with the total duration:

```
void _listenToTotalDuration() {
  _audioHandler.mediaItem.listen((mediaItem) {
    final oldState = progressNotifier.value;
    progressNotifier.value = ProgressBarState(
      current: oldState.current,
      buffered: oldState.buffered,
      total: mediaItem?.duration ?? Duration.zero,
    );
  });
}
```

Since you update the media item in the audio handler layer when the duration changes, you can also listen to the `mediaItem` stream here in the state management layer to get the new duration.

Now use the three methods you just created by adding them to the bottom of the `init()` method:

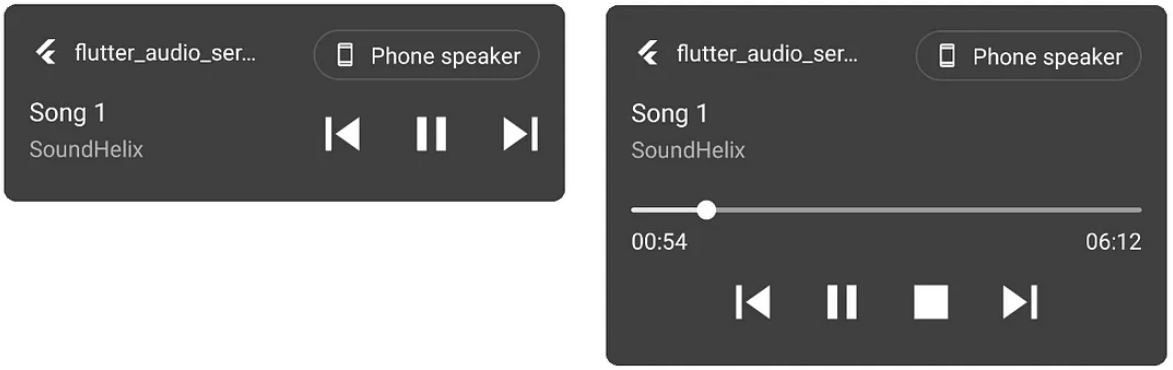
```
_listenToCurrentPosition();
_listenToBufferedPosition();
_listenToTotalDuration();
```

Note: *You can use RxDart to combine these three streams into one if you prefer.*

Run the app and you'll see the progress bar update with the current position, buffered position, and total duration:

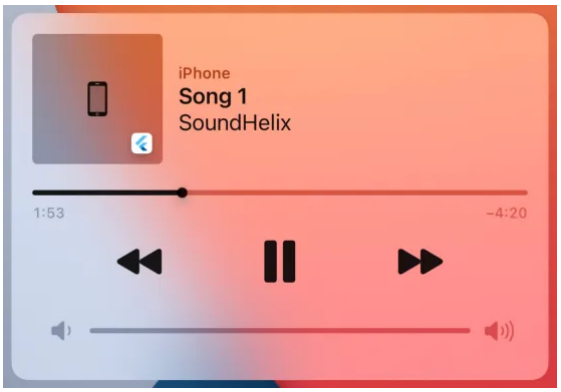
In-app view

Here is what it looks like in the Android notification drawer. You might have to expand the notification to see the seek bar:



Android system notification views (compact and expanded)

And here is the view from the iOS notification center:



iOS notification center view

Note: You can replace that big ugly grey square around the Flutter logo with an image if you add a URL to the `artUri` parameter of `MediaItem`.

Try seeking to a new position both within the app and using the system controls to confirm that you can.

Skipping to a different song in the playlist

If you try to press the skip-to-next or skip-to-previous buttons in the app or in the system controls, they won't work. Let's implement that next and also show the current song in the app UI.

Passing skip events from state management to AudioHandler

The UI is already set up to send an event to the state management layer by calling `previous()` or `next()`. Now you just need to pass those calls on to the `AudioHandler`.

Open **page_manager.dart** and replace `previous()` and `next()` with the following:

```
void previous() => _audioHandler.skipToPrevious();
void next() => _audioHandler.skipToNext();
```

Managing skip events in AudioHandler

The audio handler logic is just as easy. Open **audio_handler.dart** and add the following methods to `MyAudioHandler`:

```
@override
Future<void> skipToNext() => _player.seekToNext();

@override
Future<void> skipToPrevious() => _player.seekToPrevious();
```

In order to make sure that AudioHandler knows when there's a new song, you can listen to the current index stream in `JustAudio`. Add the following method to `MyAudioHandler`:

```
void _listenForCurrentSongIndexChanges() {
  _player.currentIndexStream.listen((index) {
    final playlist = queue.value;
    if (index == null || playlist.isEmpty) return;
    mediaItem.add(playlist[index]);
  });
}
```

Whenever the current song changes, you can use the index to look up the new `MediaItem` in your audio handler `queue`. Adding this to the `mediaItem` stream will let

`AudioHandler` know the current song.

Now add that method to the `MyAudioHandler()` constructor body:

```
  _listenForCurrentSongIndexChanges();
```

So `AudioHandler` knows the current song, but your UI doesn't what it is yet. Let's fix that.

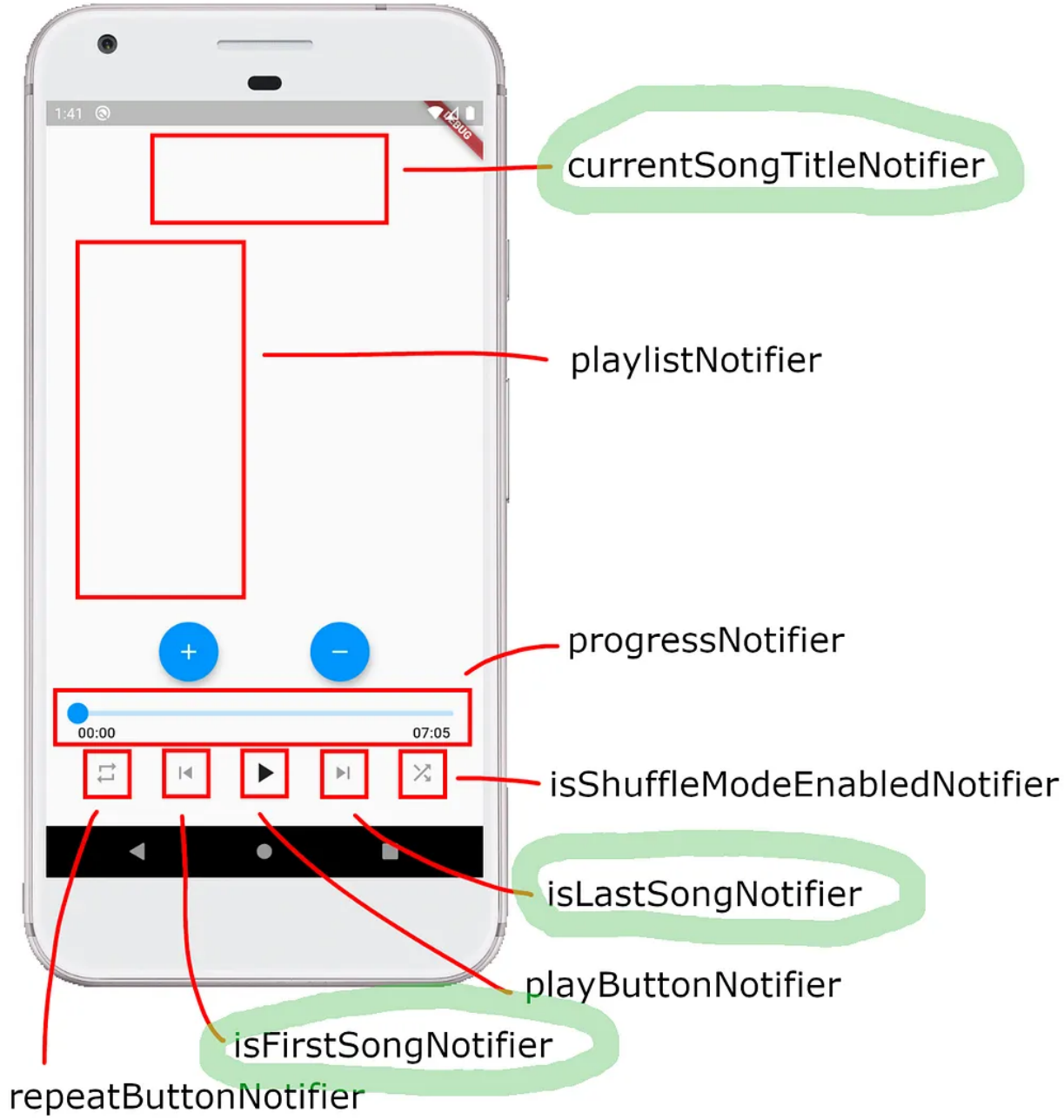
Updating the app UI with the current song

You need to listen to a data stream coming from `AudioHandler` to get notified of the current song. Go to **page_manager.dart** and add the following methods to `PageManager`:

```
void _listenToChangesInSong() {
  _audioHandler.mediaItem.listen((mediaItem) {
    currentSongTitleNotifier.value = mediaItem?.title ?? '';
    _updateSkipButtons();
  });
}

void _updateSkipButtons() {
  final mediaItem = _audioHandler.mediaItem.value;
  final playlist = _audioHandler.queue.value;
  if (playlist.length < 2 || mediaItem == null) {
    isFirstSongNotifier.value = true;
    isLastSongNotifier.value = true;
  } else {
    isFirstSongNotifier.value = playlist.first == mediaItem;
    isLastSongNotifier.value = playlist.last == mediaItem;
  }
}
```

The `mediaItem` stream tells you the current media item, and you can use this information to set three different value notifiers that the UI layer is listening to. I've factored out the code to update the skip-to-previous and skip-to-next buttons because you'll call this method from another place later.

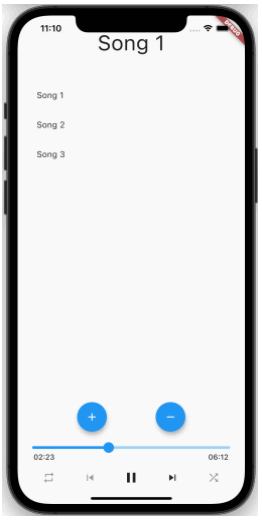


Now add `_listenToChangesInSong()` at the end of the `init()` method:

```
_listenToChangesInSong();
```

Note: You're actually already listening to the `mediaItem` stream in `_listenToTotalDuration()`. You may consider refactoring at this point and combining both methods into one.

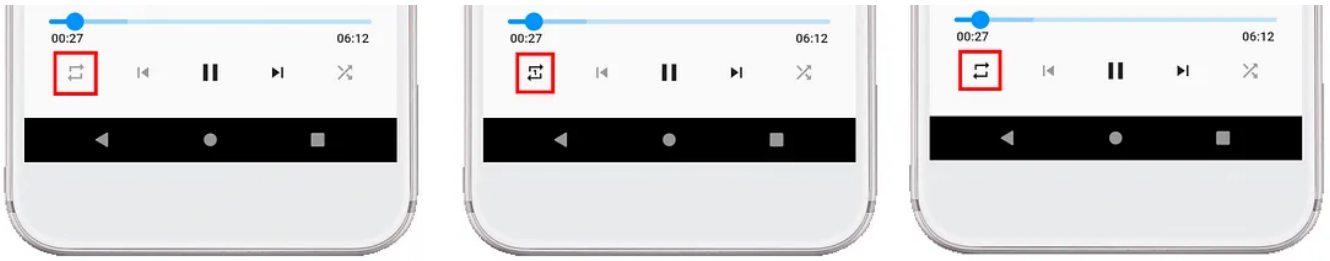
Run the app and note that the song title gets updates and that the skip-to-next and skip-to-previous buttons work in the app and in the system controls.



Skipping demo on iOS

Setting the repeat mode

Right now the playlist will play through once and then it will stop. However, the UI is set up to allow repeating the song or playlist. Let's implement that next.



repeat off (left), repeat song (middle), repeat playlist (right)

Passing repeat mode events to AudioHandler

The app, Audio Service, and Just Audio all have different enums for repeat mode so you have to convert between them at every level.

```dart
// app
enum RepeatState {
  off,
  repeatSong,
  repeatPlaylist,
}
```

```dart
// Audio Service
enum AudioServiceRepeatMode {
  none,
  one,
  all,
  group,
}
```

```dart
// Just Audio
enum LoopMode {
  off,
  one,
  all,
}
```

enums for the same thing

Note: `AudioServiceRepeatMode.group` *is not implemented yet.*

Open **page_manager.dart** and replace the empty `repeat()` method with the following code:

```dart
void repeat() {
  repeatButtonNotifier.nextState();
  final repeatMode = repeatButtonNotifier.value;
  switch (repeatMode) {
    case RepeatState.off:
      _audioHandler.setRepeatMode(AudioServiceRepeatMode.none);
      break;
    case RepeatState.repeatSong:
      _audioHandler.setRepeatMode(AudioServiceRepeatMode.one);
      break;
    case RepeatState.repeatPlaylist:
      _audioHandler.setRepeatMode(AudioServiceRepeatMode.all);
      break;
  }
}
```

Notes:

- The app's `repeatButtonNotifier` already contains the logic to loop through the `RepeatState` enums. (`RepeatState` is an enum I defined myself.)

- The only other task you accomplished here was to map `RepeatState` to `AudioServiceRepeatMode` on the `AudioHandler`.

Passing repeat mode events to Just Audio and the system

The audio handler's `setRepeatMode` isn't implemented yet, so open **audio_handler.dart** and add the following override to `MyAudioHandler`:

```
@override
Future<void> setRepeatMode(AudioServiceRepeatMode repeatMode) async {
  switch (repeatMode) {
    case AudioServiceRepeatMode.none:
      _player.setLoopMode(LoopMode.off);
      break;
    case AudioServiceRepeatMode.one:
      _player.setLoopMode(LoopMode.one);
      break;
    case AudioServiceRepeatMode.group:
    case AudioServiceRepeatMode.all:
      _player.setLoopMode(LoopMode.all);
      break;
  }
}
```

Again, you are just mapping the enum from one layer to another layer, this time from Audio Service to Just Audio.

Recall what I said earlier about your `AudioHandler` implementation needing to update both the audio player and the system. You've already notified Just Audio when you called `_player.setLoopMode`, but you haven't notified the system yet. If you look back at the new `playbackState` in `_notifyAudioHandlerAboutPlaybackEvents()`, you'll notice that there's no entry for repeat mode. Let's add it in over there.

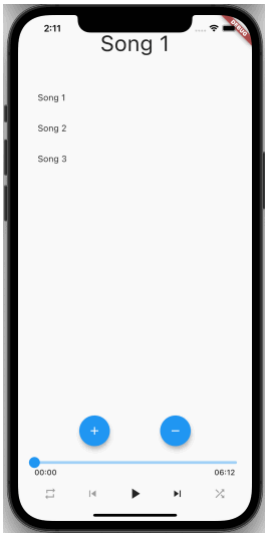
Go to `_notifyAudioHandlerAboutPlaybackEvents()` and add the following parameter to the `copyWith()` method on the `PlaybackState` value that's already there:

```
repeatMode: const {
  LoopMode.off: AudioServiceRepeatMode.none,
  LoopMode.one: AudioServiceRepeatMode.one,
  LoopMode.all: AudioServiceRepeatMode.all,
}[_player.loopMode]!,
```

This is just a compact way to map Just Audio's enum back to Audio Service's enum. Now that AudioHandler's `playbackState` is aware of the repeat mode, the Audio Service will make sure that the system is too.

Note: It's perhaps unnecessary to update `playbackState` with the repeat mode because our app's value notifier handles its own repeat mode state. We're also not showing a repeat mode button in the system audio controls. (There isn't even an option for that.) However, it still seems to me like a good idea to keep everyone up to date with the correct audio state.

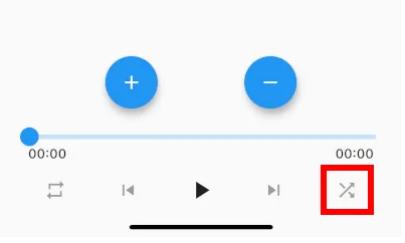
Run the app and press the repeat mode button. Check that the repeat mode behaves as expected.



Repeat mode working

Randomly shuffling the songs in a playlist

You might have a favorite playlist but sometimes it's nice to get a little variety instead of always listening to songs in the same order. That is where shuffle mode comes in.



When you press the shuffle button you want the app to randomize the play order. Pressing it again should return the playlist to the original order.

Passing shuffle mode events to AudioHandler

Open **page_manager.dart** and replace the empty `shuffle()` method with the following code:

```
void shuffle() {
  final enable = !isShuffleModeEnabledNotifier.value;
  isShuffleModeEnabledNotifier.value = enable;
  if (enable) {
    _audioHandler.setShuffleMode(AudioServiceShuffleMode.all);
  } else {
    _audioHandler.setShuffleMode(AudioServiceShuffleMode.none);
  }
}
```

The method updates the value notifier for the UI and also calls `setShuffleMode` on `AudioHandler`, which you'll implement next.

Passing shuffle mode events to Just Audio and the system

Open **audio_handler.dart** and add the following override to `MyAudioHandler`:

```
@override
Future<void> setShuffleMode(AudioServiceShuffleMode shuffleMode) async {
  if (shuffleMode == AudioServiceShuffleMode.none) {
    _player.setShuffleModeEnabled(false);
  } else {
    await _player.shuffle();
    _player.setShuffleModeEnabled(true);
  }
}
```

Shuffling involves not only calling `shuffle()`, but also setting the shuffle mode Boolean. (Setting it to `true` should be done after calling `shuffle()`.) Every time you call `shuffle()` on the audio player you will get a new random order for the playlist.

Updating the system

You've updated the order of the songs in the playlist for the audio player, but you haven't told the system about that change yet. This means updating the audio handler `queue` when the sequence of songs changes. To find out when there's a change, listen to `sequenceStateStream` on the audio player.

Add the following method to `MyAudioHandler`:

```
void _listenForSequenceStateChanges() {
  _player.sequenceStateStream.listen((SequenceState? sequenceState) {
    final sequence = sequenceState?.effectiveSequence;
    if (sequence == null || sequence.isEmpty) return;
    final items = sequence.map((source) => source.tag as MediaItem);
    queue.add(items.toList());
  });
}
```

This is the place I mentioned earlier where it would be useful to have a copy of the `MediaItem` object in Just Audio's audio source tag. That makes it relatively painless to recreate a list of media items for the `queue`.

And add this method to the `MyAudioHandler()` constructor body so that it starts listening when `MyAudioHandler` is initialized:

```
_listenForSequenceStateChanges();
```

Also, like you did for repeat mode, you can set the shuffle mode for `playbackState`. Go to `_notifyAudioHandlerAboutPlaybackEvents()` and add the following parameter to the `copyWith()` method on the `PlaybackState` value that's already there:

```
shuffleMode: (_player.shuffleModeEnabled)
    ? AudioServiceShuffleMode.all
    : AudioServiceShuffleMode.none,
```

Refactoring a few old methods to handle shuffling

Unfortunately, you also need to update some of the code you wrote earlier to accommodate the new shuffled indexes when shuffle mode is on.

Find the `_listenForDurationChanges()` method and replace it with the following. The bolded code is the part that changed.

```
void _listenForDurationChanges() {
  _player.durationStream.listen((duration) {
    var index = _player.currentIndex;
    final newQueue = queue.value;
    if (index == null || newQueue.isEmpty) return;
    if (_player.shuffleModeEnabled) {
```

```
      index = _player.shuffleIndices!.indexOf(index);
    }
    final oldMediaItem = newQueue[index];
    final newMediaItem = oldMediaItem.copyWith(duration: duration);
    newQueue[index] = newMediaItem;
    queue.add(newQueue);
    mediaItem.add(newMediaItem);
  });
}
```

The `shuffleIndices` is a list (like `[0, 2, 1]`) that tells what the song index order is when in shuffle mode. You can use this list to convert `_player.currentIndex` to the the `queue` index.

Next find `_listenForCurrentSongIndexChanges()` and replace that method with the following:

```
void _listenForCurrentSongIndexChanges() {
  _player.currentIndexStream.listen((index) {
    final playlist = queue.value;
    if (index == null || playlist.isEmpty) return;
    if (_player.shuffleModeEnabled) {
      index = _player.shuffleIndices!.indexOf(index);
    }
    mediaItem.add(playlist[index]);
  });
}
```

Again, the bolded text is what you added to handle shuffling.

Since the state management layer is already listening for changes in the queue, there is nothing that you need to add there.

Note: A big thanks to M-Shinoda who fixed a bug in this tutorial that was driving me crazy trying to get the shuffling to work.

Optional: Challenge 1 help

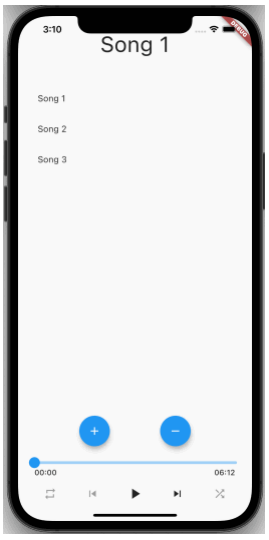
If you decide to implement seeking to a specific song (which you haven't done yet but you'll need to if you decide to try Challenge 1 at the end of the article), then you should add the following method to `AudioHandler`:

```
@override
Future<void> skipToQueueItem(int index) async {
  if (index < 0 || index >= queue.value.length) return;
  if (_player.shuffleModeEnabled) {
    index = _player.shuffleIndices!.indexOf(index);
  }
  _player.seek(Duration.zero, index: index);
}
```

Skipping to an item in the queue obviously needs to take into account whether or not the playlist is shuffled or not, which is what the `shuffleModeEnabled` check handles for you.

Testing it out

Run the app and press the shuffle button several times. Observe the song titles change order.

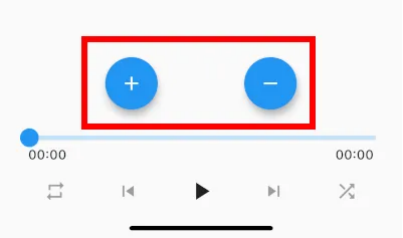


Shuffling the playlist order

Note: Since there are only three songs and the current one won't change, you might have to try a few times before you actually get a different song order. Shuffling will be more obvious when you add more songs to the playlist in just a bit.

Editing the playlist

So far the playlist has been a constant three songs. The app has some buttons to add and remove songs, though, so let's implement them.



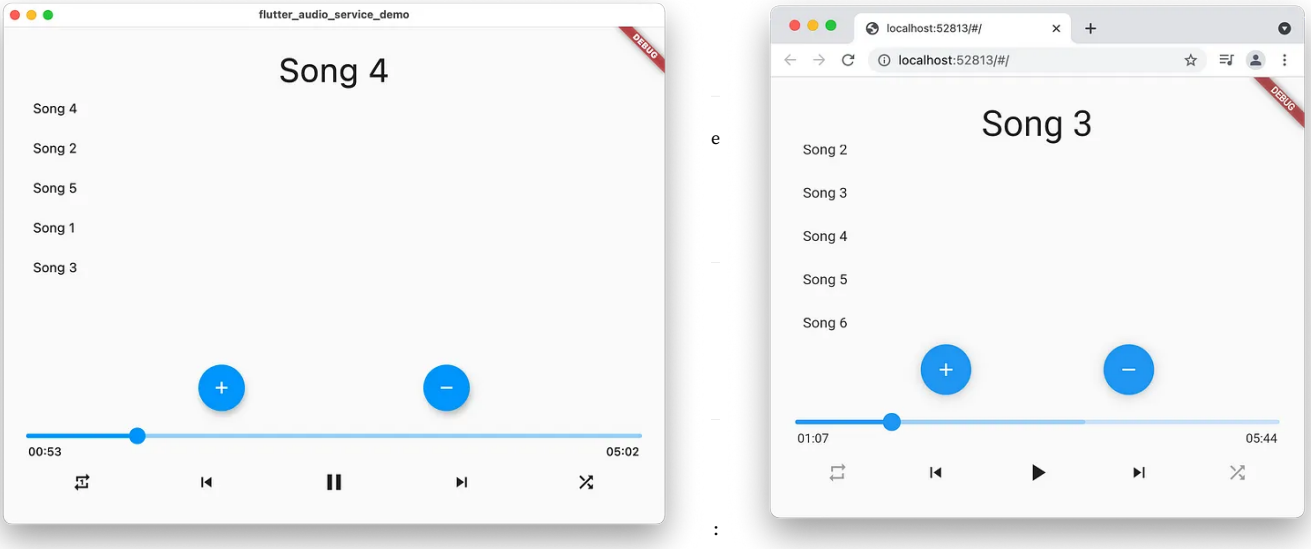
Add and remove buttons

A real app is not likely to have buttons like this to add and remove songs, but they provide a convenient way to demonstrate editing a playlist.

Passing playlist editing events to AudioHandler

Open **page_manager.dart** and replace the empty `add()` method with the following code:

```
void add() async {
  final songRepository = getIt<PlaylistRepository>();
  final song = await songRepository.fetchAnotherSong();
  final mediaItem = MediaItem(
    id: song['id'] ?? '',
    album: song['album'] ?? '',
```



Application running on macOS (left) and Chrome web (right)

```
  @override
  Future<void> addQueueItem(MediaItem mediaItem) async {
    // manage Just Audio
    final audioSource = _createAudioSource(mediaItem);
    _playlist.add(audioSource);
    // notify system
    final newQueue = queue.value..add(mediaItem);
    queue.add(newQueue);
  }
```

Notes:

- If you recall, `_createAudioSource()` is a method you added earlier to help you convert the `MediaItem` type to the `AudioSource` type that Just Audio needs.
- As when you made `addQueueItems` earlier (note the plural form), you need to both add the new item to Just Audio and also have the `AudioHandler` queue notify the system about the new playlist.
- Since you're already listening to the sequence state stream and updating the queue in `_listenForSequenceStateChanges()`, you technically don't need to update `queue` here. However, for those not implementing shuffling and listening to the `sequenceStateStream`, I'm leaving it here. Feel free to remove it yourself though.

Now add an override for `removeQueueItemAt()`:

```
  @override
  Future<void> removeQueueItemAt(int index) async {
    // manage Just Audio
    _playlist.removeAt(index);
    // notify system
    final newQueue = queue.value..removeAt(index);
    queue.add(newQueue);
  }
```

Source code

The **starter** and **final** projects for this tutorial are available on GitHub. Here are some quick links from the final project:

Challenges

As much ground as we covered in the tutorial today, there's still a lot more that you could do with this app and with Audio Service and Just Audio. Here are a few challenges for you if you'd like to do a little more:

- **Challenge 1:** Enable pressing an item in the playlist to start playing that song.
- **Challenge 2:** Drag an item in the playlist to change the default play order.
- **Challenge 3:** Use the `SeekHandler` mixin to add fast forward and rewind functionality.
- **Challenge 4:** Add two sliders to the UI to control volume and playback speed.

Going on

Check out the following links to learn more:

- Audio Service documentation
- Audio Service official tutorial
- Audio Service official example

Also have a look at the just_audio_background plugin by the same author as audio_service. I only discovered it after writing this tutorial and I haven't tried it yet, so it might make it completely unnecessary to use audio_service. The docs say:

> *This package plugs into just_audio to add background playback support and remote controls (notification, lock screen, headset buttons, smart watches, Android Auto and CarPlay). It supports the simple use case where an app has a single AudioPlayer instance.*
>
> *If your app has more complex requirements, it is recommended that you instead use the audio_service package directly.*

Notes:

- main.dart (UI layer)
- page_manager.dart (State management layer)
- audio_handler.dart (Service layer for Audio handler and Just Audio)

Open an issue or leave a comment below if you have any questions or things to add.

Support

If you've found audio_service or just_audio useful in your projects, consider supporting Ryan Heise, the primary developer of these two open source plugins. His GitHub

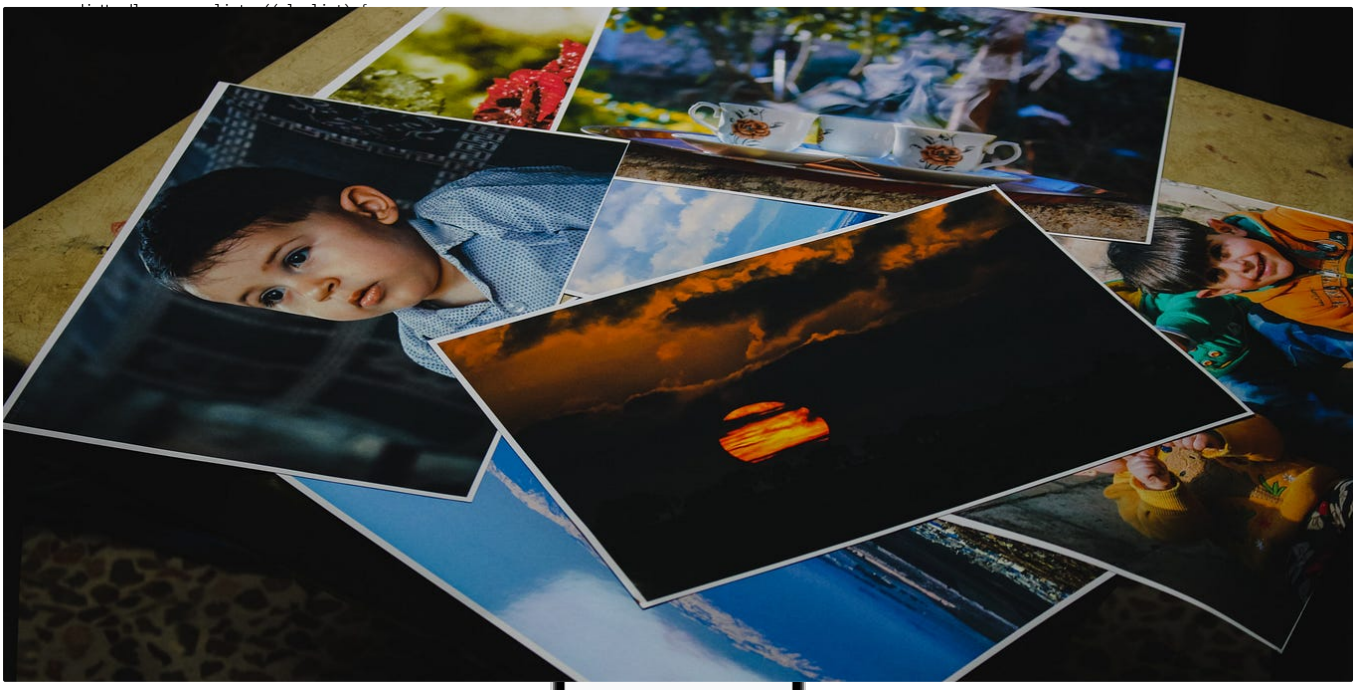
Flutter Dartlang Programming Audio Mobile App Development

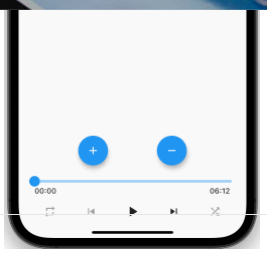
Open **page_manager.dart** and find the `_listenToChangesInPlaylist()` method that you wrote earlier:

```
void _listenToChangesInPlaylist() {
  _audioHandler.queue.listen((playlist) {
    if (playlist.isEmpty) return;
    final newList = playlist.map((item) => item.title).toList();
    playlistNotifier.value = newList;
  });
}
```


At the time, that was fine, but now that you can add and remove songs from the playlist, you should handle a few more edge cases. Replace the entire `_listenToChangesInPlaylist()` method with the following:


```
void _listenToChangesInPlaylist() {
```



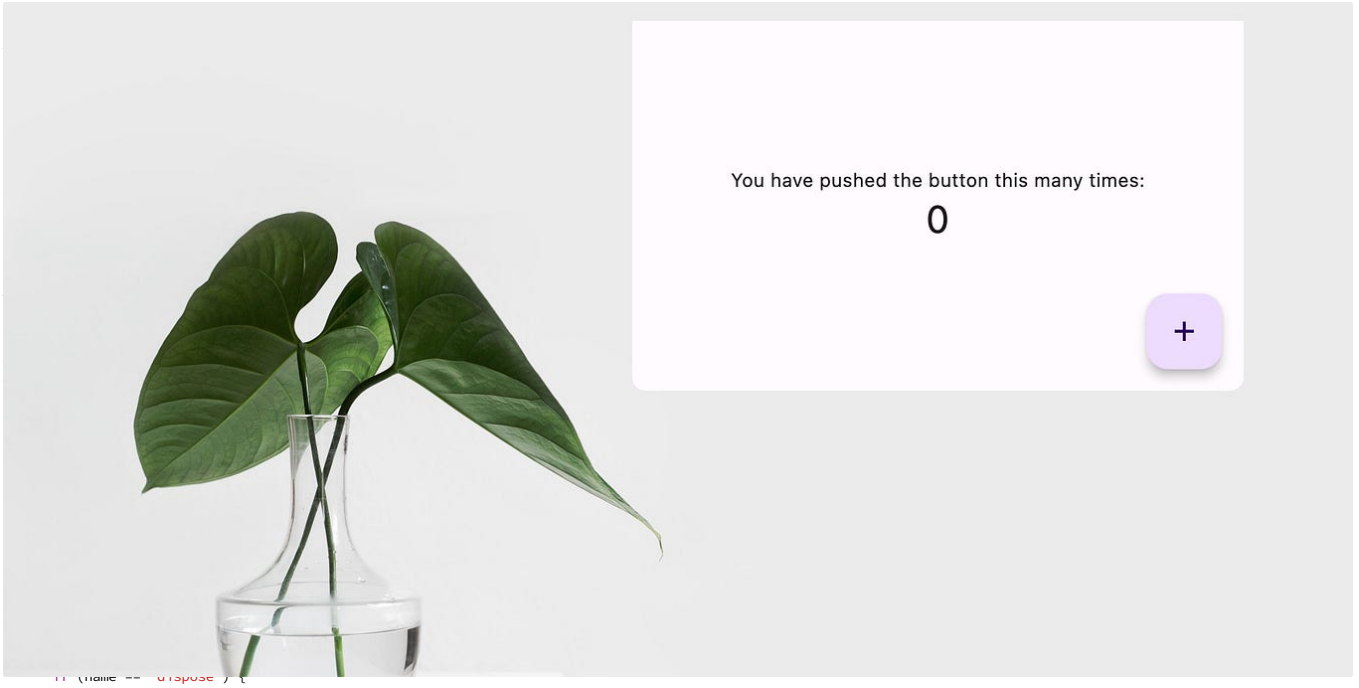


Disposing the audio resources

You're almost done, but before you finish the tutorial, you should dispose your `AudioHandler` and `AudioPlayer`.

Open **page_manager.dart** and replace the empty `dispose()` method with the following code:

```
void dispose() {
  _audioHandler.stop();
}
```




```
    await _player.dispose();
    super.stop();
  }
```

Suragch

Flutter minimalist state management: Counter app

An additional example to clarify the process

· 5 min read · Oct 6

`AudioHandler`'s `customAction` method allows you to implement any logic that isn't supported by default. You just use string matching to catch the method name. In this case it's `'dispose'`, but you could also use `customAction` for supporting something like `'setVolume'` and pass in the volume setting in `extras`.

63 1

Now update the `stop()` method of `MyAudioHandler` to stop the audio player rather than dispose it:

Congratulations! This was a long tutorial but you made it through. Here is what the final app looks and sounds like:

Suragch

Simple SQFlite database example in Flutter

Ten minutes to get a working app

· 2 min read · Jan 8, 2019

1.8K 6

Recommended from Medium

Vijay R in vijaycreations

Isolates in Flutter | Dart Isolate Tutorial — Run tasks in background using Isolates

In this article we will discuss about when, why and how to use isolates in Flutter apps.

4 min read · May 29

7 1

Free quota per day	Price beyond the free quota (per unit)
50,000	$0.06
20,000	$0.18
20,000	$0.02
1 GiB storage	$0.18

Karandeep Singh in Better Programming

Firebase Firestore: Cut Costs by Reducing Reads

Before we start, I should explain that the sample code here is in Swift and SwiftUI but you can implement the logic behind it in other...

6 min read · May 1

70

Lists

General Coding Knowledge
20 stories · 470 saves

It's never too late or early to start something
15 stories · 174 saves

Coding & Development
11 stories · 228 saves

Stories to Help You Grow as a Software Developer
19 stories · 484 saves

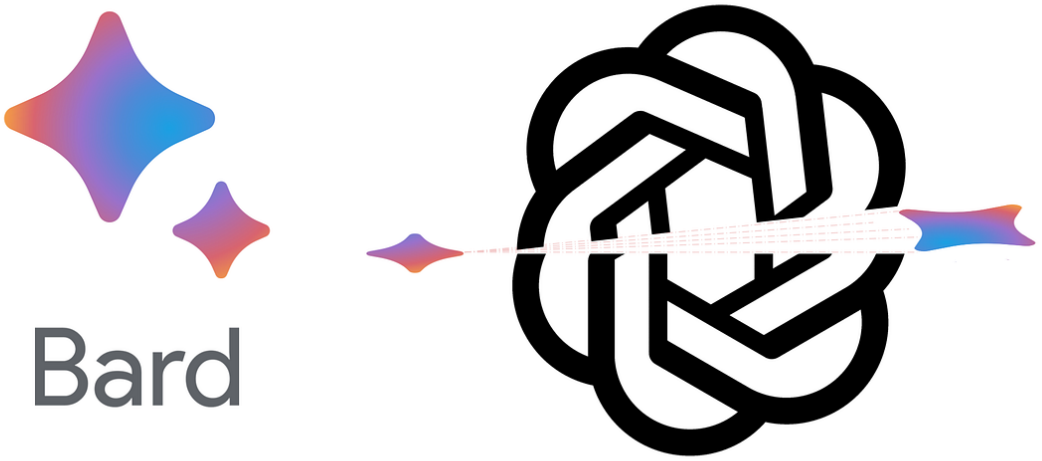
Matatias Situmorang

Invoke Child Method from Parent Class in Flutter & Vice Versa

How to invoke a method in the child widget from the parent widget or invoke a parent method from the child widget.

6 min read · Jun 21

21 1



AL Anany

The ChatGPT Hype Is Over — Now Watch How Google Will Kill ChatGPT.

It never happens instantly. The business game is longer than you know.

· 6 min read · Sep 1

15.7K 477

Eman Yaqoob

Map in Dart/Flutter

In general, a map is an object that associates keys and values. Both keys and values can be any type of object. Each key occurs only once...

5 min read · Jun 16

2

Lumei Lin

Using SQLite In Flutter

Use the SQLite plugin to save data offline and create a database in a Flutter app

8 min read · Jun 18

8

See more recommendations