1 Data flow

1.1 Actions and events

The application state can only be modified by dispatching *actions*. Updates to the state produce *events*, which *listeners* can use to update the UI.

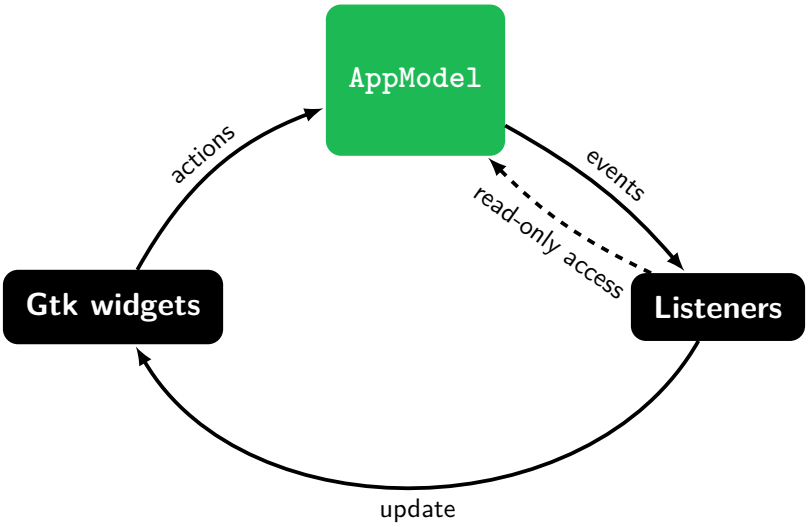

Figure 1: The data flow and and its relation to the UI

Events can often help to have an idea of the current state. However, the `AppState` should be the only source of truth for the current state of the application. The `AppModel` enforces read-only access to that state.

This draws heavy inspiration from the Flux architecture[1]; the one big difference here is that there is no way to automatically find out which portion of the UI should be updated. Instead, listeners are responsible for figuring out the updates to apply based on the events.

It should be noted that the app state is only readable from the main thread for simplicity.

1.2 A listener: the player subsystem

Any element that wishes to update the state or react to changes from the state has to follow that same pattern. For instance, the "player" part of Spot receives `Commands`

[1] *In-Depth Overview: Flux*. URL: `https://facebook.github.io/flux/docs/in-depth-overview`.

(mapped from events by a `PlayerNotifier`) to start playing music, and dispatches actions back the app through a `SpotifyPlayerDelegate` (see figure 2).

These two extra elements add some indirection so that the player is not too strongly coupled to the rest of the app (it does not and should not care about most events, afterall!). Moreover, those commands are handled in a separate thread where the player lives.

1.3 Everything is just actions and events!

In fact, widgets and their associated listeners (called *components* throughout the codebase) form another similar subsystem that only communicate with actions and events.

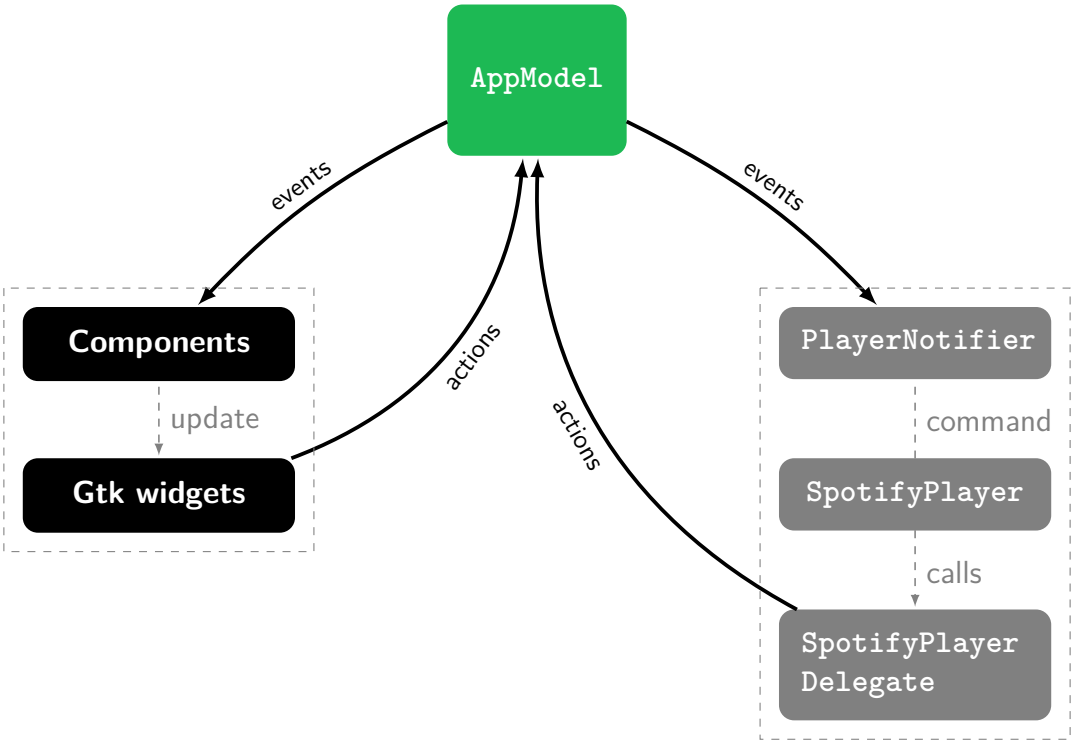

Figure 2: The player subsystem

1.4 Another listener: the MPRIS subsystem

Similarly, the MPRIS subsystem follows that same pattern. It spawns a small DBUS server that translates DBUS messages to actions, and an `AppPlaybackStateListener` listens to incoming events.

One major difference is that the MPRIS server has its own state here, since the app state cannot be accessed from outside the main thread. That state, however, is shared between the server thread (the MPRIS state read when a DBUS message asks for it) and the main thread (the MPRIS state is updated by the listener).

This is an alternative approach to the one used for the player subsystem, where yet another MPSC channel was used to bridge the gap between two threads.

To make sure this local state stays in sync, DBUS messages should not alter the local state directly – instead, we should wait for a roundtrip through the app and incoming events.

2 How actions are handled

All actions being dispatched, synchronous or not, are eventually sent through an MPSC channel[2]. The consumer on the other end of the channel is a future that will be executed by GLib. This allows Gtk to process *all actions* at its own pace, as part of its main loop.

This is the only time that the app state is borrowed mutably – to apply actions.

Note: futures are used a lot in the code to perform asynchronous operations such as calls to the Spotify API. To ease the use of futures, the dispatcher allows working with asynchronous actions, that is, futures that output one or more actions. Again, these futures are eventually handled in the main Gtk loop.

3 Conventions for components

Components are listeners dedicated to binding Gtk widgets so that they produce the right actions, and updating them when specific events occur.

The usual pattern adopted in the codebase is to have a model struct that defines accessors to the state and action-producing methods.

Because the state can only be borrowed immutably using the `AppModel`, there are sadly some hoops and loops to go through to work with `RefCells` if one wishes to avoid cloning data from the state.

[2] *Module futures::channel::mpsc*. URL: `https://docs.rs/futures/latest/futures/channel/mpsc/index.html`.