1 Data flow within Spot

1.1 Overview

Single source of truth. There is a single place that is considered the source of truth for anything that is related to the app state, and that is, well, the `AppState`. The app state aggregates the state of the UI, as well as the player state. This makes it easier to keep things in sync – when possible, anything state-related should be read from the app state over some local, possibly out-of-date state.

Centralized. That state is centralized and unique. This allows various parts of the application to access any part of it, and conversely makes it easy to perform state updates that affect various and sometimes unrelated parts of the application.

Controlled mutations. There is only one way to modify the app state, and that is by dispatching *actions* – plain structs that represent a mutation to the state. Updates to the state produce *events*, which `EventListeners` can use to update the UI.

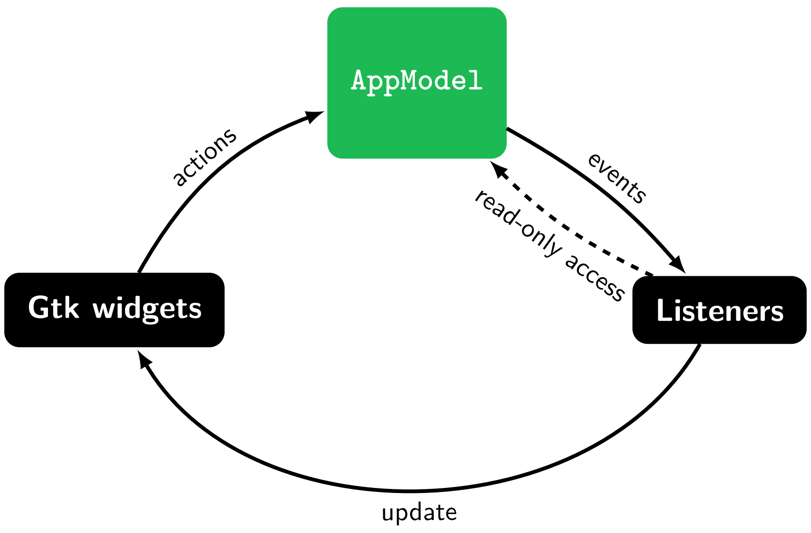

Figure 1: The data flow and and its relation to the UI – the `AppModel` enforces read-only access to the state.

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 How actions are handled

Here is the relevant part of the code[2] related to handling actions and notifying listeners:

```
let events = self.model.update_state(action);
```

[1]See https://facebook.github.io/flux/docs/in-depth-overview for instance
[2]Variables have been renamed for clarity...

```
for event in events.iter() {
    for listener in self.listeners.iter_mut() {
        listener.on_event(event);
    }
}
```

That first line is the only time that the app state is borrowed mutably – to apply actions.

On the technical side: all actions being dispatched, synchronous or not, are eventually sent through a `futures::channel::mpsc` channel. 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.

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.

1.3 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` (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.

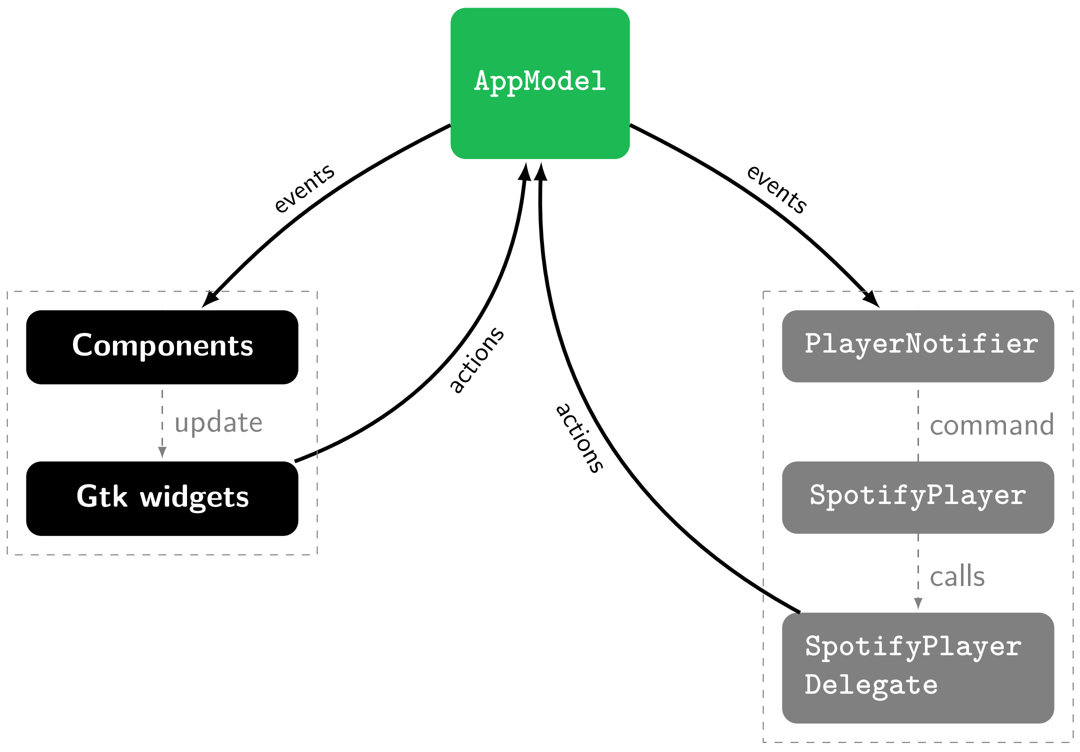

Figure 2: The player subsystem

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 maintains its own state here, since the app state cannot be accessed from outside the main thread. 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 Components

2.1 Overview

Components are thin wrappers around Gtk widgets, dedicated to binding them so that they produce the right actions, and updating them when specific events occur by conforming to `EventListener`.

2.2 Modeling interactions

Components should have some associated `struct` to model the interactions with the rest of the app. Let's consider the play/pause button as an example. Its behavior is defined in the `PlaybackModel`:

```rust
impl PlaybackModel {
    fn is_playing(&self) -> bool { /**/ }
    fn toggle_playback(&self) { /**/ }
}
```

What we need to make our button work is a way to know its current state (is a song playing?) and a way to change that state (toggling on activation). Note that it would be tempting to simply query the widget's state, which *should* be in sync with the actual playback state, but what we should really do instead is query the app state, which is the one source of truth for anything state-related.

Why do this? First, toggling the playback might fail (e.g. if no song is playing), but more importantly something else could alter the playback state (e.g. a DBUS query).

```rust
fn is_playing(&self) -> bool {
    self.app_model.get_state().playback.is_playing()
}
```

As for toggling the playback, remember that we can only mutate the state through actions (the `get_state` call above returns some `Deref<Target = AppState>`). In other words, we express what kind of action we want to perform, with no guarantee that it'll succeed.

```rust
fn toggle_playback(&self) {
    self.dispatcher.dispatch(PlaybackAction::TogglePlay.into());
}
```

2.3 Binding the widget

All that's left is binding the widget to our model. By wrapping our model in an `Rc`, it becomes easy to clone it into the kind of `'static` closure Gtk needs.

```
// model is an Rc<PlaybackModel>
widget.connect_play_pause(clone!(@weak model => move || model.toggle_playback()));
```

Finally, we need our component to listen to relevant events, and update our widget accordingly.

```
impl EventListener for PlaybackControl {
    fn on_event(&mut self, event: &AppEvent) {
        match event {
            AppEvent::PlaybackEvent(PlaybackEvent::PlaybackPaused)
            | AppEvent::PlaybackEvent(PlaybackEvent::PlaybackResumed) => {
                let is_playing = self.model.is_playing();
                self.widget.set_playing(is_playing);
            }
            /**/
        }
    }
}
```