Here's a crash course in the Antares Protocol!

The Antares Protocol grew out of the need to improve the behavior of front-end applications that are based on React and Redux, and some sort of async processing middleware such as Redux Thunk Redux Observable and Redux Saga.

(Q)

In this frequently used architecture, you'll find a central store is mainly responsible for 2 things: Maintaining a single source of truth for data, called the state tree, and for managing the consequences of async calls.

Consequences are things in the real world that you can't take back, such as sending real electricity to a wire, or posting to an API. When you hear about side-effects, or 'effects' as you'll hear them called some times, these are what I'm referring to as consequences.

(Sierra consequence aside)

The Antares Protocol describes an architecture that decouples the handling of consequences from the handling of data. The AP considers storing data itself a consequence.

So the most valuable service an Antares agent provides to your application is a consequence and timing engine. You can't do 'realtime' without a good engine designed to waste as little of your time, and the computers', as is possible.

Because... (there are problems)

There are problems...

with the current architecture.

So recall that in current architecture, the store manages both of these things - data, and consequences.

Problem 1 - Coupling

I think almost every application has effects, or consequences, but not every application requires a store, or has a UI.

So Redux-middleware based solutions, while powerful, are not applicable to entire categories of apps. If you're working on NodeJS Servers, a Command Line apps, and IOT device-controlling application, you may not want to manage your consequences through a store.

Problem 2 - Complicated UI Components

Components have too much responsibility, and have grown too complex. This causes issues when we let our UI layer handle the consequences of our application.

Raise your hand if you've ever had (We've all had) a component update too frequently, triggering API calls or other bad things at the wrong times, or because an object identity was different when we didn't expect it.

Components allow low-level concepts like the object identity of props to control important, possibly expensive things, like API calls. That's like parents letting children drive the car!

Frankly, it's very hard to get the right combination of component lifecycle events to control our consequences tightly and adequately.

Simplify your components by reducing their responsibility back to where it started with React:

View is a Function of Props.

All the components need to do is originate actions describing what they want an agent to do on their behalf, which they get by calling a function, like `dispatch`, called `process`.

This produces simpler applications, which means you can do more with less of your scarcest resource: time.

What is the Protocol?

The agent has 2 ways of getting actions into it, and 2 ways of assigning consequences from it - that's all. You get actions into an agent by calling process. Or `subscribe`, but that is just like calling process in a loop. You would process a request to add an item to a cart. You would subscribe to a data feed - it's like calling `process` many times in a loop, over time - the actions follow the same code path whichever one you use.

When an agent processes an action, the processing can take place along either of 2 paths, one synchronous, on asynchronous. The synchronous one is called `filter` and the async one is called the `render` path

The difference between the two possible paths can be understood in terms of exceptions, and their visibility when the app calls `agent.process`

The synchronous code path - the filters - are run through first. A great use of a filter is for validation - If an action is not syntactically or grammatically valid, throwing an exception in a filter will require the component that processed the action to handle it right away, as an exception. Any errors thrown in filters will prevent the asynchronous code path, the renderers, from running.

The asynchronous code path - the renderers - are run through after all filters have run. They run independently of each other, and are always async. They are the place you will write to an API, change the voltage on a pin - etc.. Whatever you would use Sagas or Thunks for.

Renderers will typically return Observables to give the agent the ability to - know about the renderer's completion, error, and progress - queue up renderings - cancel something in progress

Renderers take a function, and run it in one of several ways, based upon configuration options. For example, parallel or serial. These are called concurrency modes. The function you write can stay decoupled from those details, buying you the flexibility to change them later as required, such as for performance reasons.

And the most interesting, Inception like configuration option is that, like in Redux Saga or Redux thunk, these renderings can be the source of future actions - such as communicating the return value of an API in a `http/post/complete` action payload.

This will be the architecture of any app built on the Antares Protocol. What will change between apps is which filters you have, which renderers you have attached, and the business logic inside of them. But your architecture is that of mini-programs (don't call them micro-services!) that respond to certain patterns of actions, and whose consequences can be turned into other actions and fed back through.

997 words (8 minutes)