XState in React Native

How I finally managed to keep my views lean and my logic clean

Dario Villanueva

`@radiodario`

a bit about me:

- @radiodario
- Bulding Anyone - a Voice Networking App
- ex Meta Reality Labs
- ex CTO at Feeld.co

Unstable of Contents

1. The problem with apps
2. What are Finite State Machines
3. How does XState model FSMs
4. How to use XState in your RN projects
5. Some Pros/Cons with XState

Before we start, a lil' game

WRONG!

This is a photo of a typical react native project

WRONG!

This is a photo of a typical redux store after a year of active development by 3 frontend developers.

WRONG!

This is just a photo of a bunch of cats sitting in boxes

```
!<--- pause for laughter --->
```

Face it, you make forms for a living.

And making forms shouldn't be that hard

Enter (Finite) State Machines

What do these two have in common?

what is a Finite State Machine?

he's literally reading out the Wikipedia definition right now smh

A litte bit about XState

- JS / TS finite state machines for modern apps
- by @davidkpiano et al.
- very active and lovely community (discord)
- https://stately.ai/editor <- visual editor also
- Really good VSCode Plugin
- Automatic typegen for machines

Modelling a my cat with XState

<iframe style="height: 100%;" src="https://stately.ai/viz/embed/040646b2-22db-4042-bc78-2a62225cc030?mode=viz&panel=code&showOriginalLink=1&readOnly=1&pan=0&zoom=0&controls=1" sandbox="allow-same-origin allow-scripts"></iframe>

<iframe style="height: 100%;" src="https://stately.ai/viz/embed/040646b2-22db-4042-bc78-2a62225cc030?mode=full&panel=code&showOriginalLink=1&readOnly=0&controls=1" sandbox="allow-same-origin allow-scripts"></iframe>

Anatomy of a Cat Machine

```
import { assign, createMachine } from "xstate";

const CatMachine = createMachine(
  {
    initial: "alive",
    context: {
      lives: 9, // the data that change accross "states"
    },
    states: {
     // the states we can be in
    },
  },
  { // these are the options - you don't need to define these here
    actions: {
      // -- named actions here*
    },
    services: {
      // -- services here
    },
    guards: {
      isDead: (ctx) => !ctx.lives,
    },
  }
);
```

States, events and transitions

```
states: {
  alive: {
    always: {
      target: "dead",
      cond: "isDead",
    },
    on: {
      DIE: {
        actions: ["diminishLives"],
      },
    },
  },
  dead: {
    type: "final",
  },
}
```

States can be "nested" (child states)

```
alive: {
  initial: "hungry",
  states: {
    hungry: {
      entry: 'meow',
      on: {
        FEED: "disappointed",
      },
    },
    asleep: {
      invoke: {
        src: 'sleep',
        onDone: 'hungry',
      }
    },
    disappointed: {
      on: {
        PET: "hungry",
      },
    },
  },
}
```

Actions for side effects 💥

```
actions: {
  meow: () => {
    console.log('meow');
  },
  diminishLives: assign({
    lives: (ctx) => ctx.lives - 1,
  }),
}
```

Machine Context

aka your *infinite* "states"

```
context: {
  lives: 9,
  mood: 'disappointed',
  awokenAt: new Date(),
},
```

`assign` is an action lets you assign values to the context:

```
actions: {
  diminishLives: assign({
    lives: (ctx, ev) => ctx.lives - 1,
  }),
}
```

Services (soon to be renamed to Actors)

~~Services~~ Actors model long running processes

```
{
  asleep: {
    invoke: {
      src: 'sleep',
    },
  },
}, {
  services: {
    sleep: async (ctx, ev) => {
      return await DigestiveSystem.digest(ctx.food);
    },
  }
}
```

Invoking ~~services~~ Actors 🧙‍♂️

```
asleep: {
  invoke: {
    id: 'sleep', // an id of your service
    src: 'sleep', // the name of your service in machine opts
    onDone: {
      // what to do when your service finishes
      // aka your promise returns
      target: 'hungry',
    },
    onError: {
      // error handling
      target: 'sick',
    },
  },
},
```

Guards - for conditional Transitions

```
guards: {
  isDead: (ctx: CatContext) => ctx.lives === 0,
  isHungry: () => true,
}
```

READ THE DOCS

👉 here 👈

Integrating with React Native

Wrap your flow in a context

```
const CatMachineContext = React.createContext<InterpreterFrom<typeof CatMachine>>();

function MyCat = () => {

  const catMachine = useInterpreter(CatMachine, {
    actions: {
      meow: (ctx, ev) => console.log("Meow");
    },
    services: {
      ...
    }
  })

  return (
    <CatMachineContext.Provider value={catMachine}>
      ...
    </CatMachineContext.Provider>
  )
}
```

Add a navigator

```
const CatFlowStack = createNativeStackNavigator<CatFlowParamList>();

function CatFlow = () => {

  const catMachine = useInterpreter(CatMachine, {...});

  return (
     <CatMachineContext.Provider value={catMachine}>
      <CatFlowStack.Navigator>
        <CatFlowStack.Screen name="Hungry" component={HungryScreen} />
        <CatFlowStack.Screen name="Asleep" component={AsleepScreen} />
        <CatFlowStack.Screen name="Disappointed" component={DisappointedScreen} />
        <CatFlowStack.Screen name="Dead" component={DeadScreen} />
      </CatFlowStack.Navigator>
    </CatMachineContext.Provider>
  )
}
```

Grab your cat by the context:

```
const HungryScreen = () => {

  const catService = useContext(CatMachineContext);

  // subscribe only to what you need
  const livesRemaining = useSelector(catService,
    current => current.context.lives,
  );

  return (
    <View>
      <Text>{livesRemaining} lives</Text>
      <Image src="@assets/hungry-cat" />
      <Button onPress={() => cat.send('FEED')} />
    </View>
  )
};
```

Use a hook to navigate by subscribing to state

```
const useHandleNavigation = () => {
  const catService = useContext(CatMachineContext)
  const navigation = useNavigation();

  useEffect(() => {
    const subscription = catService.subscribe((state) => {
      if (state.matches("alive.asleep")) {
        navigation.navigate('Asleep');
      }
      if (state.matches("alive.disappointed")) {
        navigation.navigate('Disappointed');
      }
      // ... etc
    });
    return subscription.unsubscribe;
  }, [catService, navigation]);
}
```

You get many things for free

```
const catService = useContext(MyStateMachineContext)
const [currentState, send] = useActor(catService)

// `can` will evaluate to true if the current state has any
// valid transitions in the current state for that event

<Button disabled={currentState.can('FEED')} />



// you can show a spinner for long running services
const isAsleep = useSelector(catService,
  current => current.matches('alive.asleep')
);

return (isAsleep && <Spinner>)
```

Why do this:

- Separates your logic and services from your view layer.
- Keeps views short and clean
- Reusable services / actions - swap out logic for mocks in tests
- Do Model Based Testing (it's *mental*)
- Rearranging flows / moving things around is easy - just modify the transitions
- Complex flows and logic doesn't mean complex views.
- Flipper Plugin `react-native-flipper-xstate`

Some things to keep in mind

- Autorefresh doesn't work if you change the state machine code `;_;`
- Name your actions, trust me.
- Keep state / context lean
- If you listen to the state on views that are mounted, the views will refresh
- prefer `useSelector` to `useActor` and grab only what you need from your context / state
- nativeStackNavigator -> `freezeOnBlur` screenOption

A real life specimen of Anyone Post-call review machine

References and further reading

- Finite State Machines (pdf) - David Wright
- State machines are wonderful tools - Chris Wellons
- Rage Against the Finite-State Machines
- Integrating XState with React Native and React Navigation - Simone D'Avico

Question Time

slides 👇