Componentizing Application State

Laying out your logic

Follow along here →

Nick Nisi

@nicknisi.com

SPECIAL THANKS TO ALL OUR AWESOME SPONSORS!

CLOUDFLARE

Unspecified

SOFTWARE CO

Join Us

WI 24

THAT® CONFERENCE

JULY 29TH - AUG. 1ST

Ahoy hoy!

- Software Engineer in Omaha, NE
- Panelist on JS Party
- NebraskaJS Organizer
- Former conference Emcee/Organizer
  - NEJS Conf (2015 - 2019)
  - TypeScript Conf US (2018 - 2021)

I like JavaScript and TypeScript a lot.
React is cool, too.

A few quick notes

- I use React in this talk, but React is not important — the concepts work with any front end framework, just like XState.
- I'm using XState 5 in this talk (released in November, 2023).
  - Some third party tools aren't updated to support it yet
  - `storybook-xstate-addon`

Definitely not yours

~~Your~~ application state is

Too complex

Application State vs. Business Logic

- **Application State** is **what** the current conditions of the app are.
- **Business logic** is **how** the app state got there and how it reacts to commands.

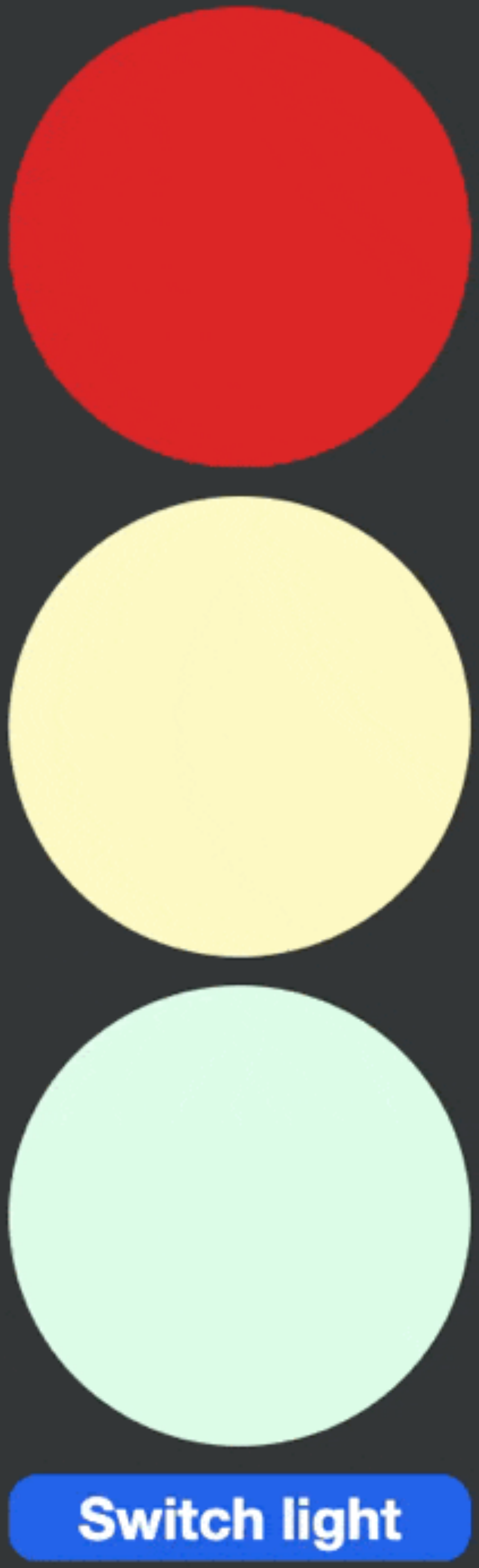
State complexity

- **Many sources:** local component state, application state, different contexts, etc.
  - Core business logic vs. local UI state, for example
- **Synchronization:** Keeping data consistent across the system
- **Concurrency:** Many operations access and modify state simultaneously
- **Temporal dependencies:** State depends on a previous sequence of events

How do you prevent **impossible** states?

A stop light

```
const c = (c: string) => cn('size-32 rounded-full', c);
return (
  <div className="m-16 flex flex-col gap-2 bg-[#323638] p-32">
    <div className={c(light === 'red' ? 'bg-red-600' : 'bg-red-100')} />
    <div className={c(light === 'yellow' ? 'bg-yellow-300' : 'bg-yellow-100')} />
    <div className={c(light === 'green' ? 'bg-green-600' : 'bg-green-100')} />
    <button
      className="max-w-32 rounded-lg bg-blue-600 font-bold text-white"
      onClick={switchLight}
    >
      Switch light
    </button>
  </div>
);
```

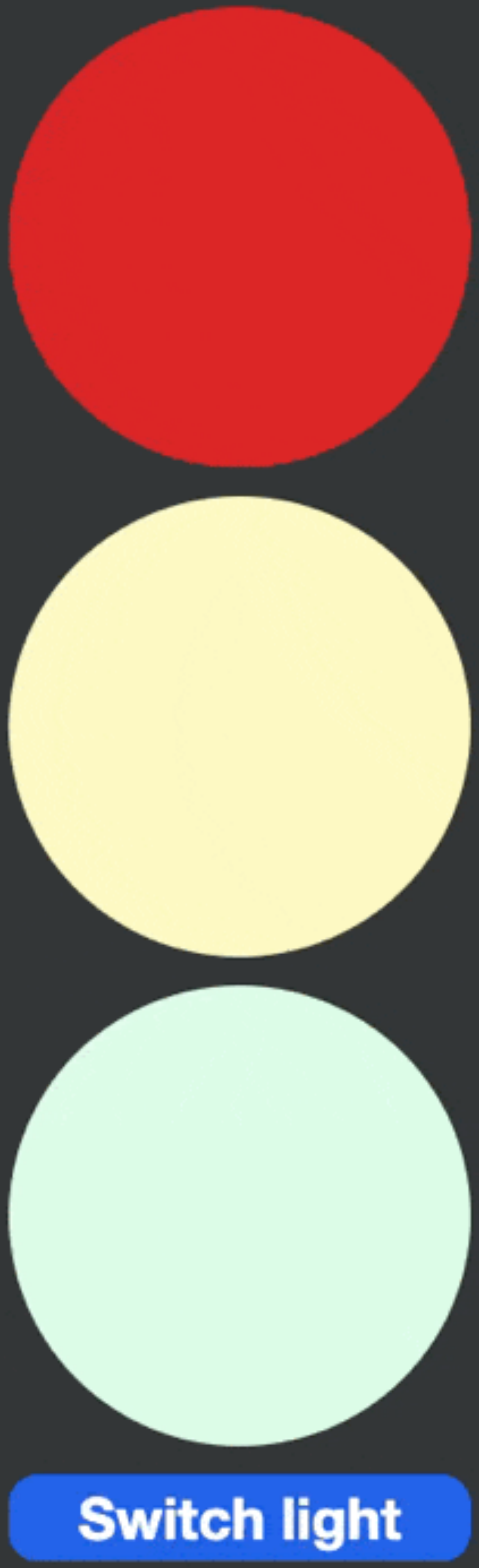


😱 Green → Red → Yellow → Green → Yellow

The problem

```
const [light, setLight] =
  useState<(typeof lights)[number]>('red');

const switchLight = () => {
  const randomLight =
    lights[Math.floor(Math.random() * lights.length)];
  setLight(randomLight);
};
```

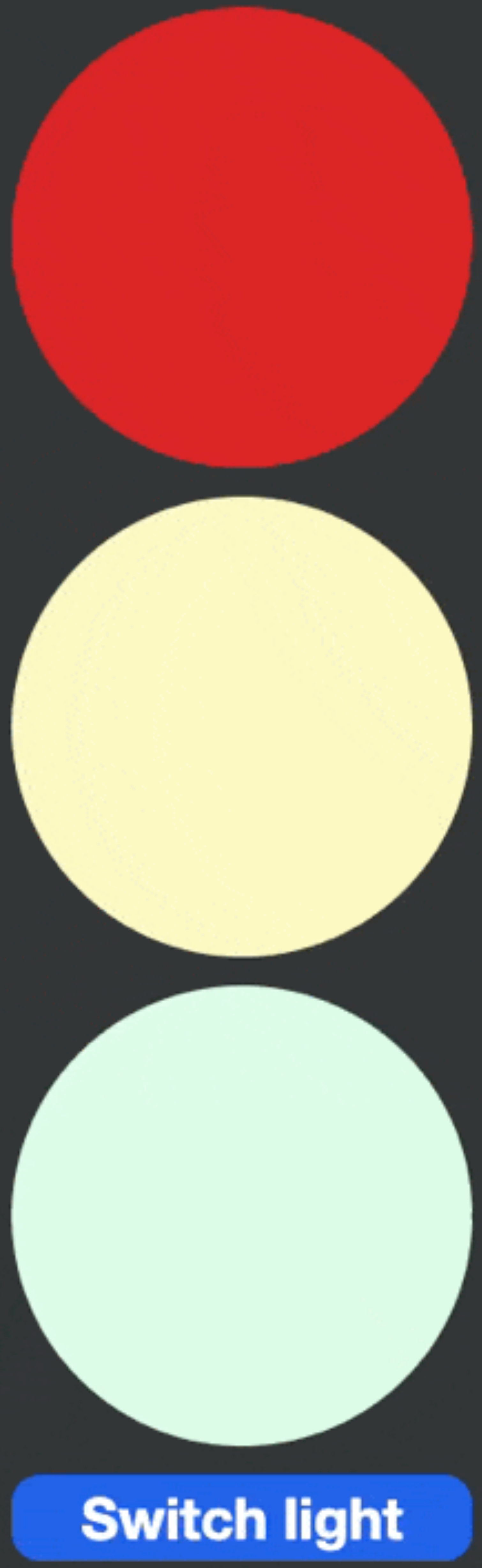


The solution

```
const [lightIndex, setLightIndex] = useState(0);

const switchLight = () => {
  setLightIndex((lightIndex + 1) % lights.length);
};

const light = lights[lightIndex];
```

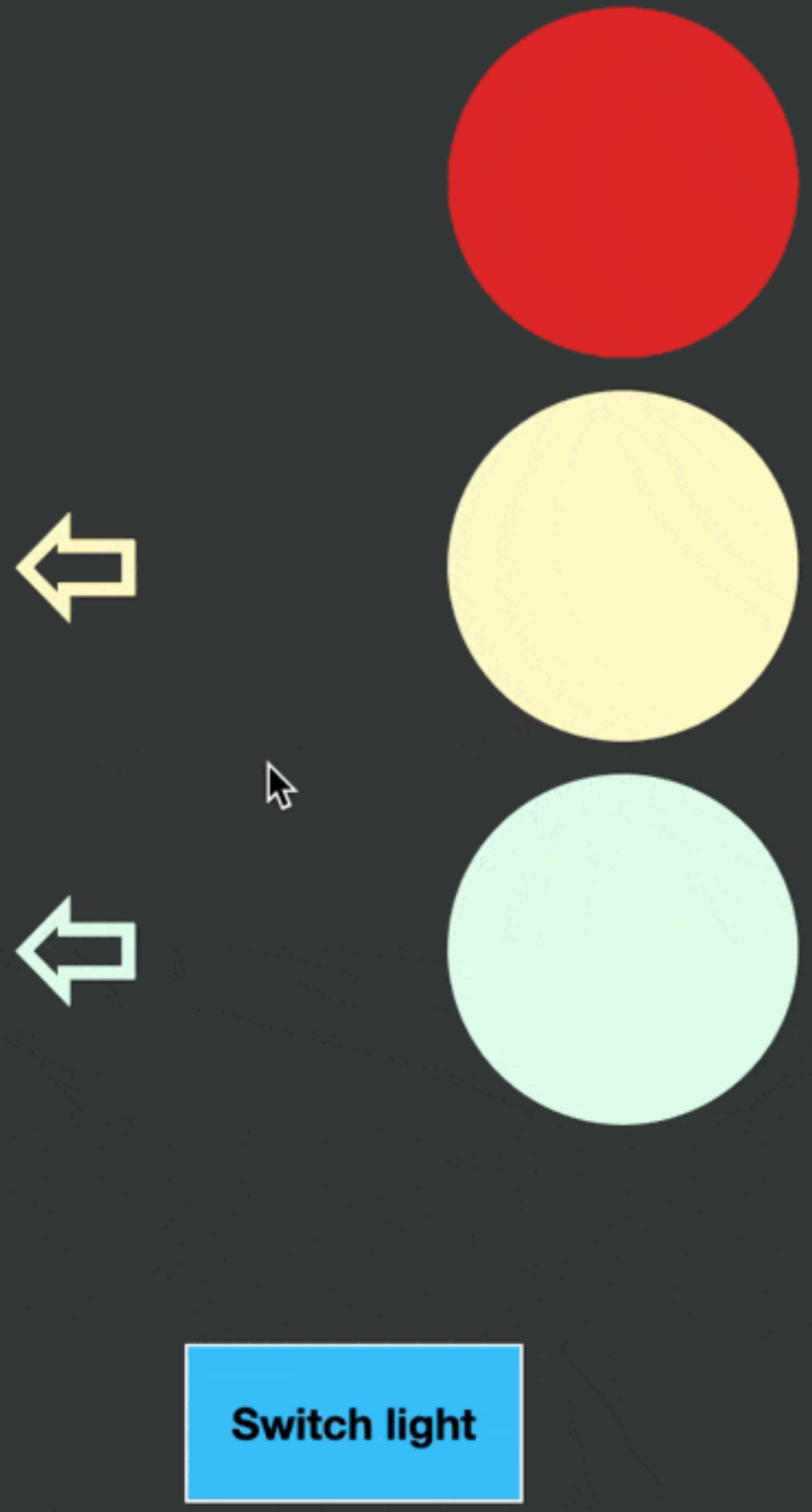


We can keep getting more

Complex 🥵

Stop light complexity

- Turn arrows
- Temporal factors
  - Time of day
  - Day of week



💸 Mo ~~Money~~ Variables Mo Problems*

```
const [light, setLight] = useState<(typeof lights)[number]>('red');
const [arrow, setArrow] = useState<'green' | 'yellow' | undefined>(undefined);

const [lightIndex, setLightIndex] = useState(0);
const switchLight = () => {
  const newIndex = (lightIndex + 1) % lights.length;
  setLightIndex(newIndex);
  setLight(lights[lightIndex]);
  setArrow((['green', 'yellow', undefined] as const)[Math.floor(Math.random() * 3)]);
};
```

*Mo money can also lead to mo problems

Another problem

Local / component-specific state

```
return (
  <div className="m-16 flex max-w-screen-sm flex-col justify-center bg-[#323638] p-32">
    <div className="m-16 flex space-x-8">
      <div className="flex flex-col justify-end space-y-3">
        <div className={c(arrow === 'yellow' ? 'text-yellow-300' : 'text-yellow-100')}>⇦</div>
        <div className={c(arrow === 'green' ? 'text-green-600' : 'text-green-100')}>⇦</div>
      </div>
      <div className="flex flex-col space-y-3">
        <div className={c(light === 'red' ? 'bg-red-600' : 'bg-red-100')} />
        <div className={c(light === 'yellow' ? 'bg-yellow-300' : 'bg-yellow-100')} />
        <div className={c(light === 'green' ? 'bg-green-600' : 'bg-green-100')} />
      </div>
    </div>
    <div className="px-32">
      <button className="mt-4 border border-white bg-sky-400 p-4 font-bold text-black"
onClick={switchLight}>
        Switch light
      </button>
    </div>
  </div>
);
```

Let’s talk about components

React Components

```
export const Game = () => (
  <div className="game">
    <Player name="nick" />
    {/* ... */}
  </div>
);

export const Player = ({ name }: Props) => (
  <div>
    <img src={`${name}.bmp`} />
    <marquee>{name}</marquee>
  </div>
);
```

React Components

Let’s talk about them

Inputs

```
export const Player = ({ name, score }: Props) => {

  const isVisible = useMemo(() => score !== 0, [score]);

  return (
    <div className={isVisible ? 'flex' : 'hidden'}>
      <img src={`${name}.bmp`} />
      <marquee>{name}</marquee>
    </div>
  );
};
```

Internal state

Desired output

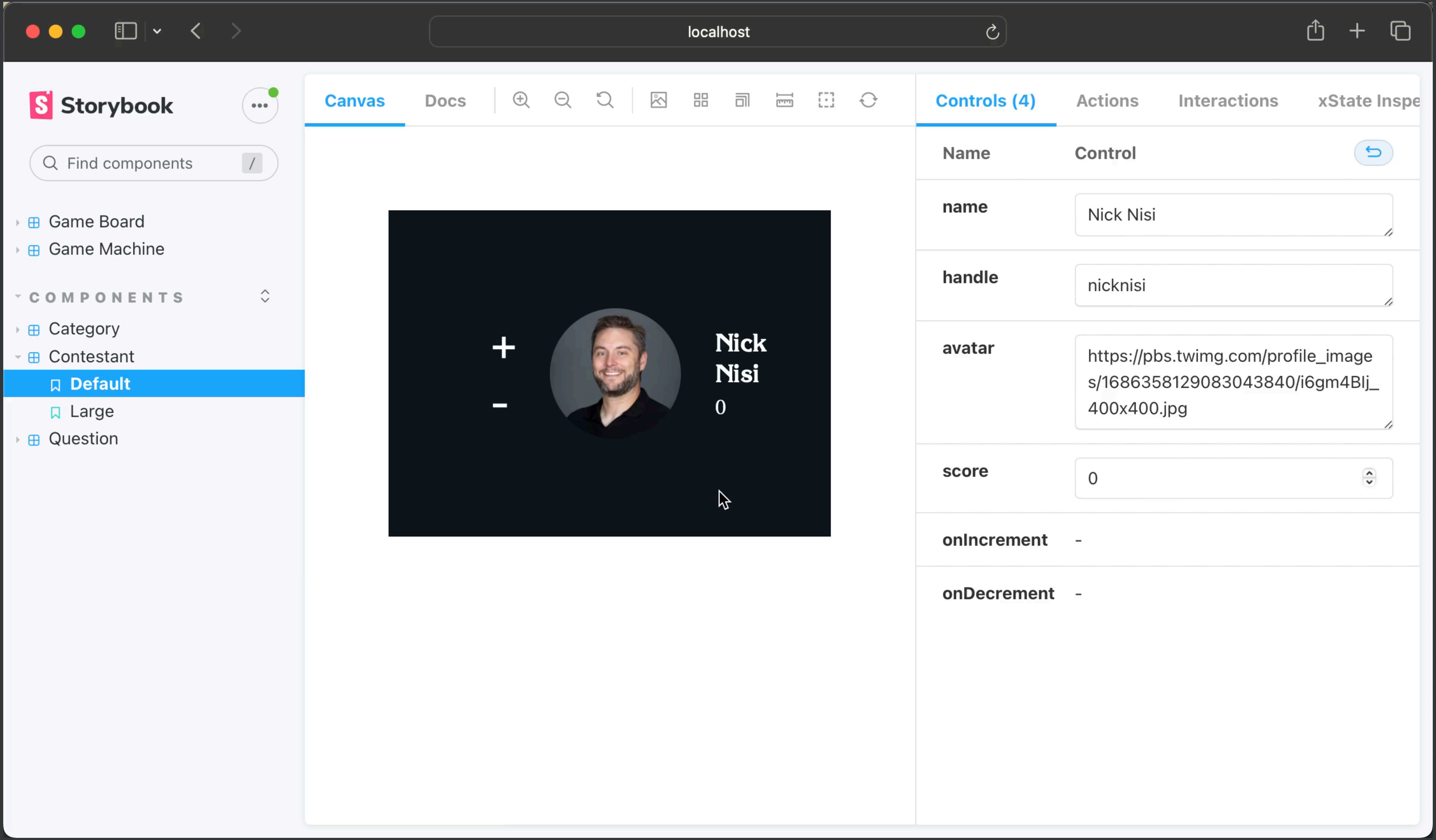
Component design benefits

- **Isolated state:** Internal, self-contained state specific to each individual use
  - Components can be built and tested independently
- **Predictable behavior:** Well-defined I/O leads to consistent behavior
- **Reusability:** Designed for specific functionality that can be used in more than one place
- **Modular:** Convenient to partition a large system into small, manageable modules
- **Fun:** Declarative UIs are fast and fun to build

Storybook

Independent component development

- Storybook helps build components faster
- Build components outside the app, in isolation
- Control inputs
- Streamlines UI development and testing



Name	Control
name	Nick Nisi
handle	nicknisi
avatar	https://pbs.twimg.com/profile_images/1686358129083043840/i6gm4BIj_400x400.jpg
score	0
onIncrement	-
onDecrement	-

#TMTOWTDI

There's more than one way to do it

More ways to handle state

And handle state across components / at an application level

The Context way

Share state but doesn’t provide prescribed way to work with it

```
export interface State {
  light: "red" | "green" | "yellow";
  arrow: "green" | "yellow" | undefined;
}

export const LightContext = createContext<State | null>(null);

export const LightProvider = ({ initialState, children }) => (
  <LightContext.Provider value={initialState}>{children}</
LightContext.Provider>
);
```

Redux

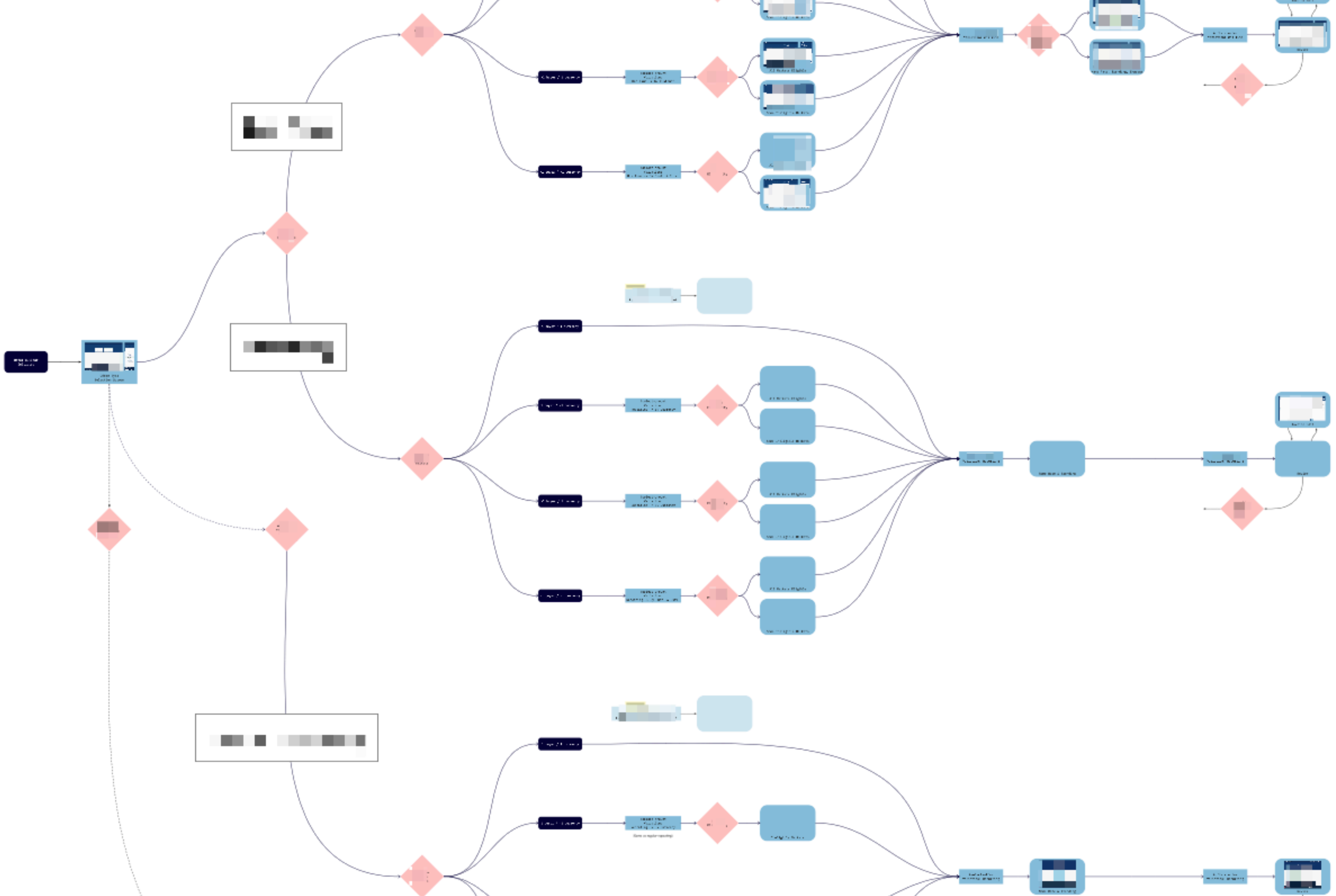
Provides more structure around the state and how it’s updated 👍

```
function lightReducer(state = { light: "red" }, action) {
  switch (action.type) {
    case "light/change":
      return { light: action.payload };
    default:
      return state;
  }
}

const store = createStore(lightReducer);
store.dispatch({ type: "light/change", payload: "yellow" });
```

What if we could solve our impossible state problem AND treat our state like a component? 🤔

🤫 We kind of want to, already.



Componentizing Application State

- Treat the app state as just another component
- Work on the state of the application and verify its flow **BEFORE** the UI exists
- Visualize and walk through the flow with non-technical stakeholders
- Components become dumb consumers of the state
  - Can easily check the current state
  - No need to handle **impossible-to-get-into** states

XState

XState
Application State
Business Logic
imgflip.com

XState

A simple light machine

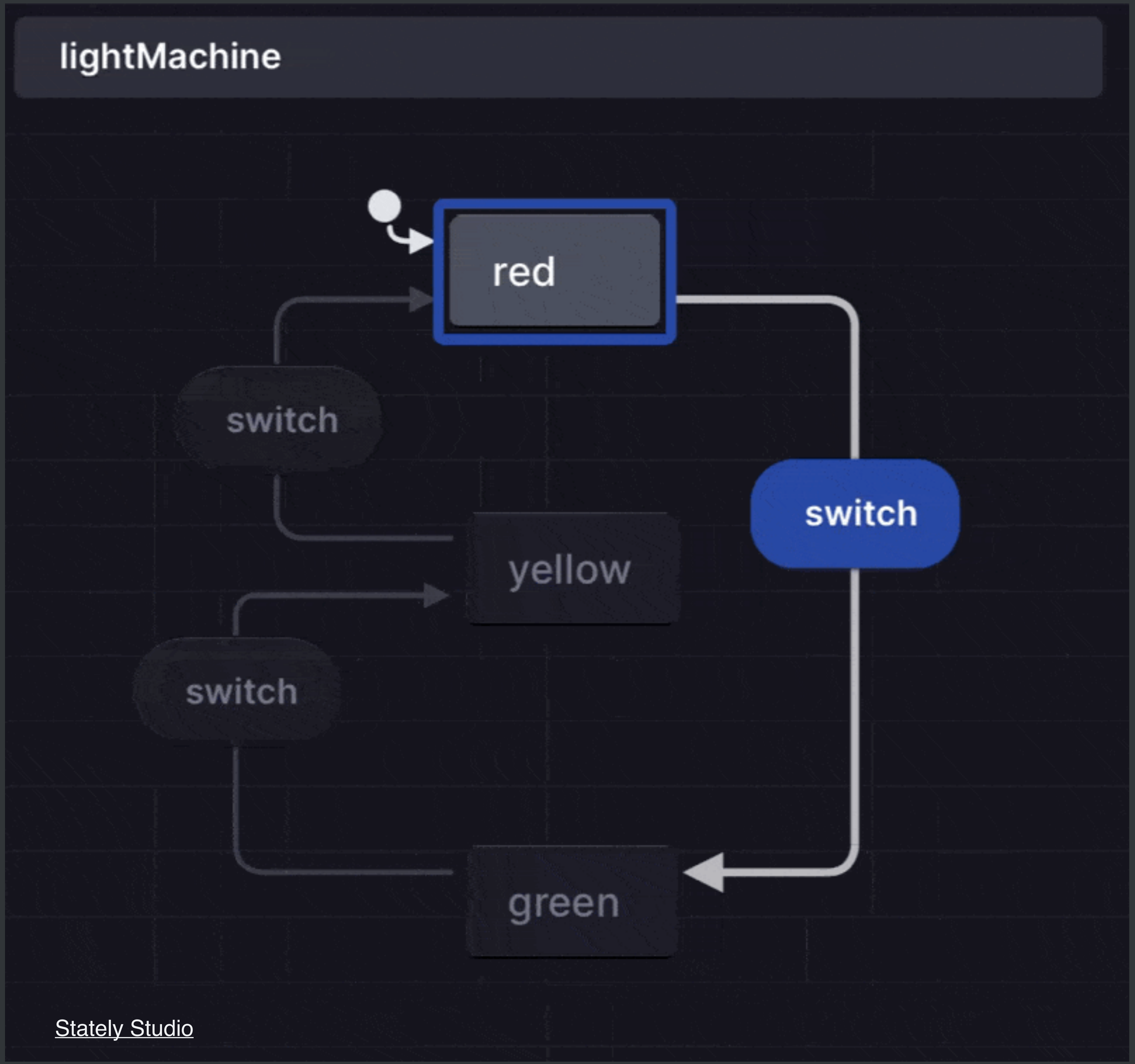
- Finite states
- Infinite states handled as private context
- Side-effects declarative and explicit
- Framework-agnostic
- Transitions defined per-state

```
import { createMachine } from 'xstate';

export const machine = createMachine({
  id: 'lightMachine',
  initial: 'red',
  states: {
    red: {
      on: {
        switch: { target: 'green' },
      },
    },
    green: {
      on: {
        switch: { target: 'yellow' },
      },
    },
    yellow: {
      on: {
        switch: { target: 'red' },
      },
    },
  },
  types: { events: {} as { type: 'switch' } },
});
```

State charts

Visualize the logic



```
import { createMachine } from 'xstate';

export const machine = createMachine({
  id: 'lightMachine',
  initial: 'red',
  states: {
    red: {
      on: {
        switch: { target: 'green' },
      },
    },
    green: {
      on: {
        switch: { target: 'yellow' },
      },
    },
    yellow: {
      on: {
        switch: { target: 'red' },
      },
    },
  },
  types: { events: {} as { type: 'switch' } },
});
```

Your state becomes [like] a component 🤯

Visualize and test independently of the app's UI

- Render state charts directly from the actual application flow
- Walk through the state and verify all possible transitions from one state to another
- Walk through the application before the UI exists
- Simulate the whole thing from Stately Studio
- You can even render them to Storybook*

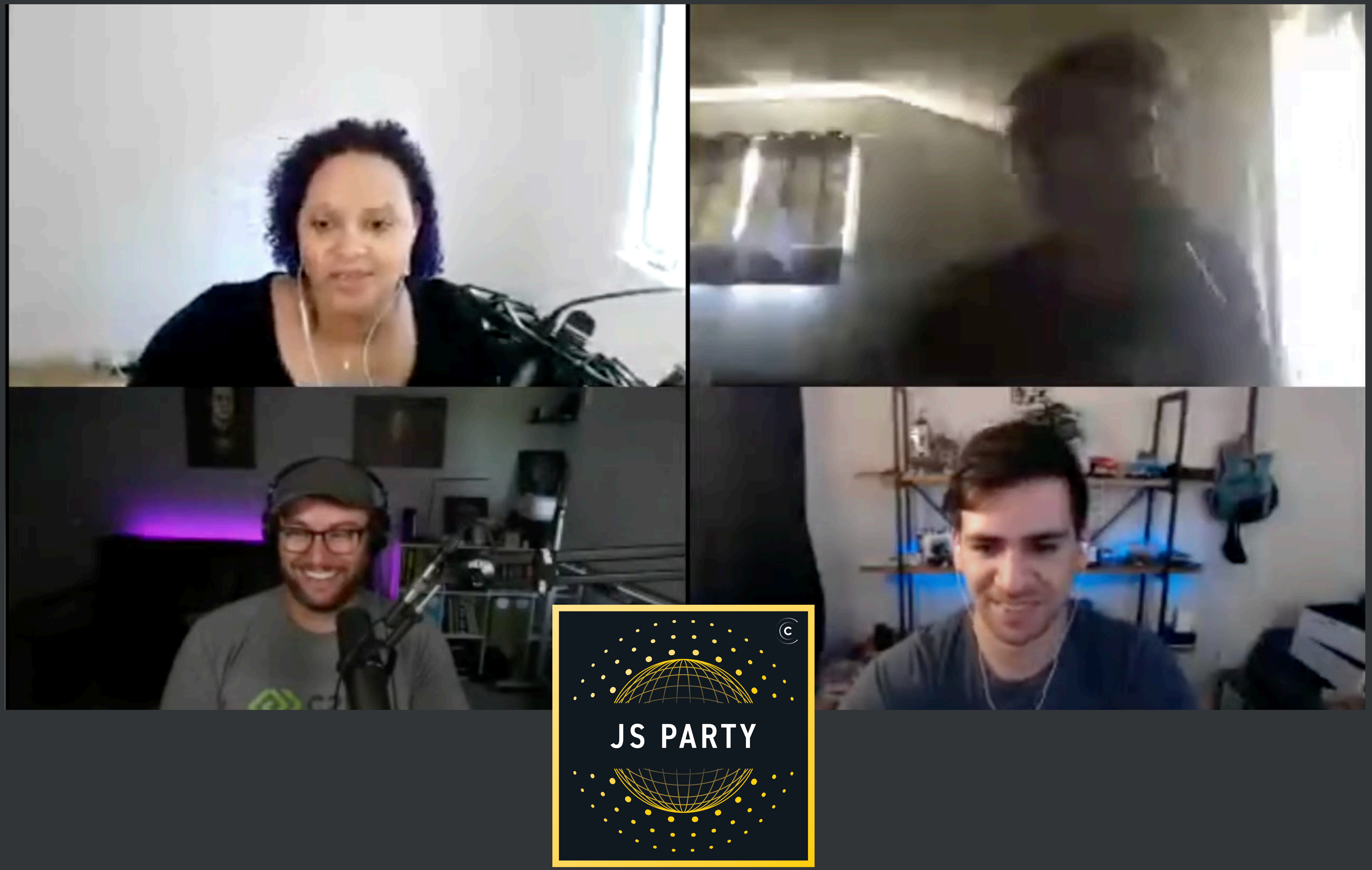
```
npm install storybook-xstate-addon
```

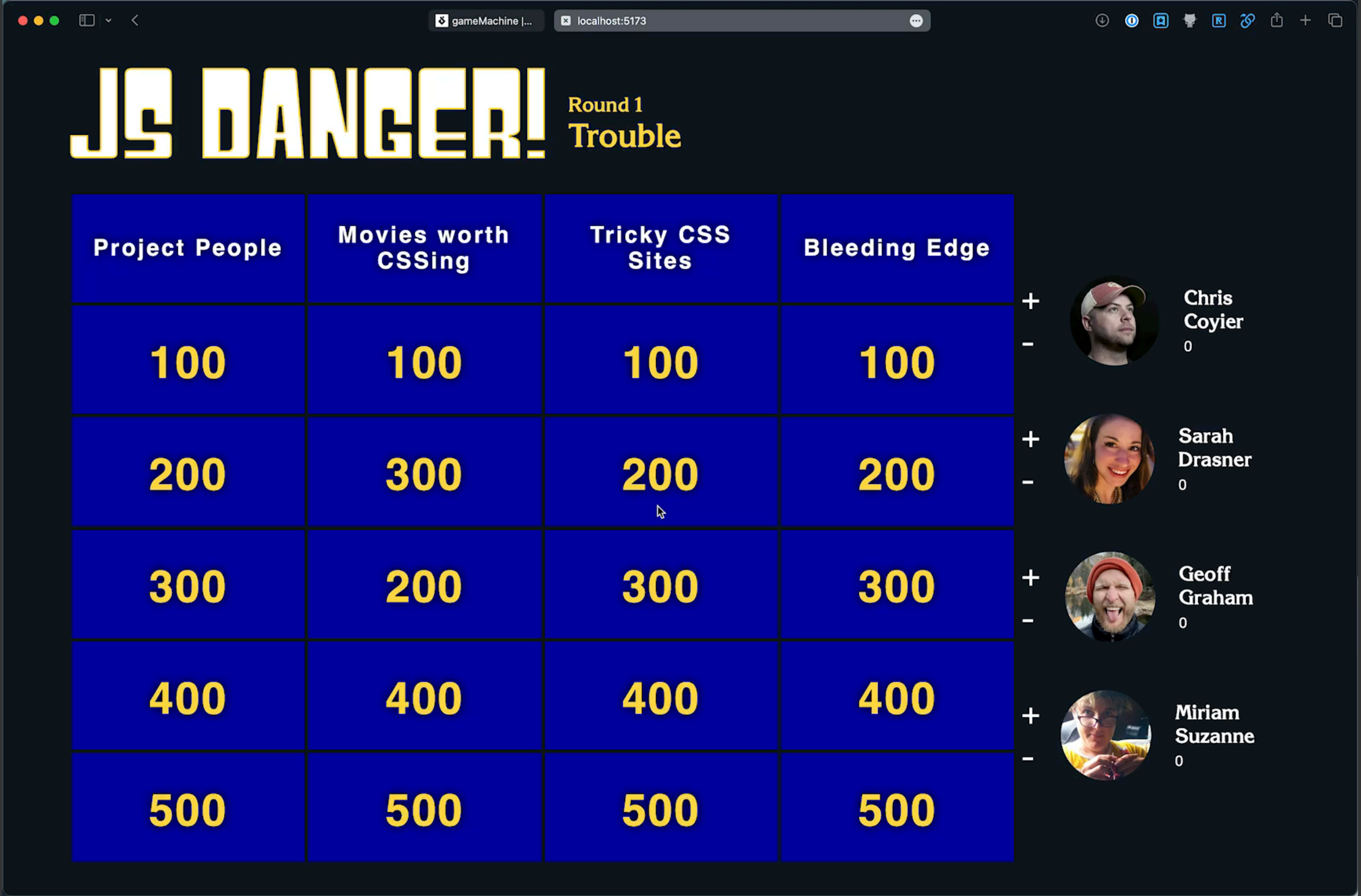
← *Currently, this only supports XState 4

How I discovered state machines

State machines, so hot right now

- Twitter
- Had David Khourshid on JS Party
- Built a demo app with XState
  - JS Danger





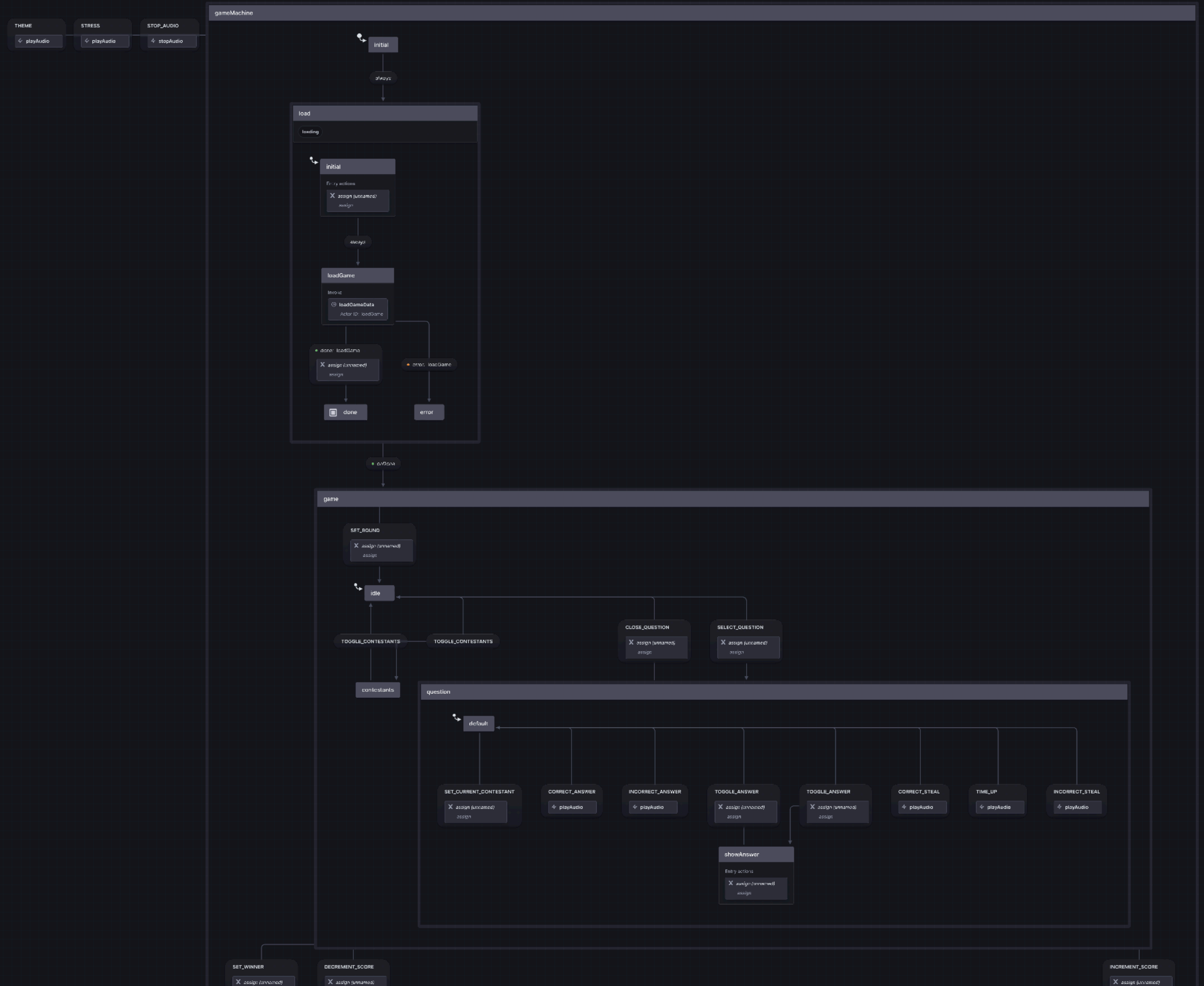
JS DANGER!

Round 1

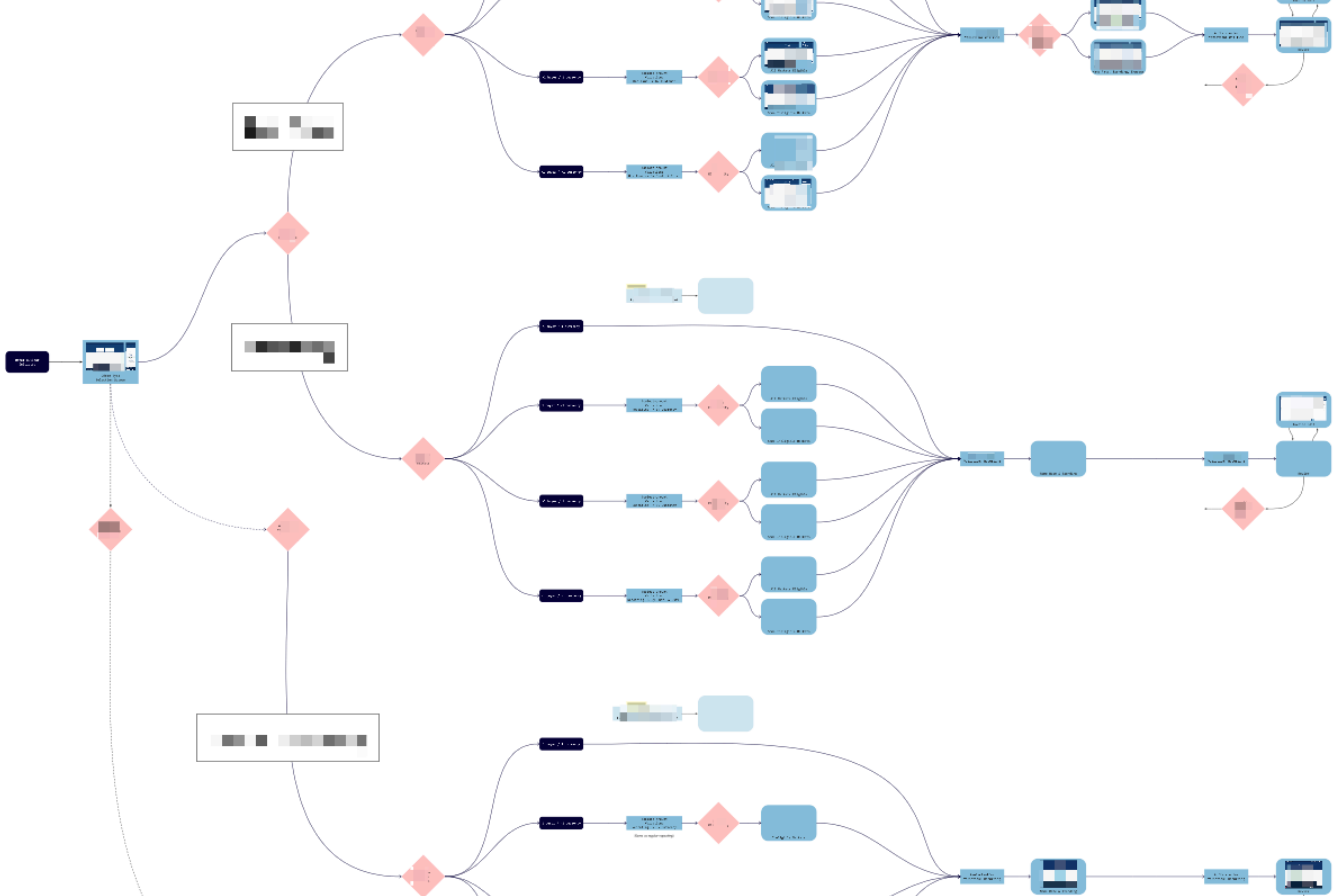
Trouble

Project People	Movies worth CSSing	Tricky CSS Sites	Bleeding Edge
100	100	100	100
200	300	200	200
300	200	300	300
400	400	400	400
500	500	500	500

+ −	Chris Coyier	0
+ −	Sarah Drasner	0
+ −	Geoff Graham	0
+ −	Miriam Suzanne	0


THEME
playAudio
STRESS
playAudio
STOP_AUDIO
stopAudio
gameMachine
initial
always
load
loading
initial
Entry actions
assign (unnamed)
assign
always
loadGame
Invoke
loadGameData
Actor ID: loadGame
done: loadGame
assign (unnamed)
assign
error: loadGame
done
error
onDone
game
SET_ROUND
assign (unnamed)
assign
idle
TOGGLE_CONTESTANTS
TOGGLE_CONTESTANTS
contestants
CLOSE_QUESTION
assign (unnamed)
assign
SELECT_QUESTION
assign (unnamed)
assign
question
default
SET_CURRENT_CONTESTANT
assign (unnamed)
assign
CORRECT_ANSWER
playAudio
INCORRECT_ANSWER
playAudio
TOGGLE_ANSWER
assign (unnamed)
assign
TOGGLE_ANSWER
assign (unnamed)
assign
CORRECT_STEAL
playAudio
TIME_UP
playAudio
INCORRECT_STEAL
playAudio
showAnswer
Entry actions
assign (unnamed)
assign
SET_WINNER
assign (unnamed)
DECREMENT_SCORE
assign (unnamed)
INCREMENT_SCORE
assign (unnamed)

Then I introduced it at work



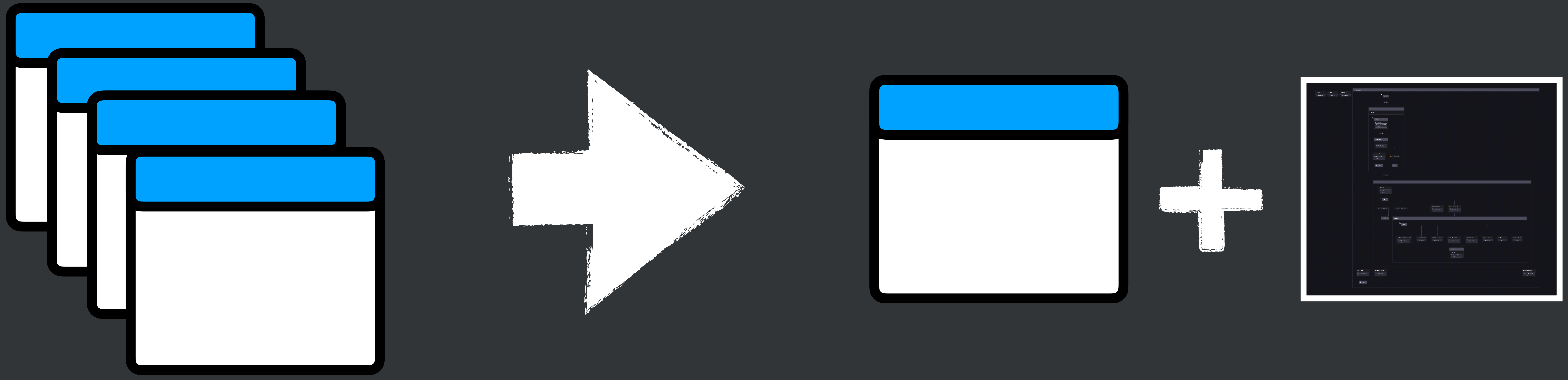
Real-life use

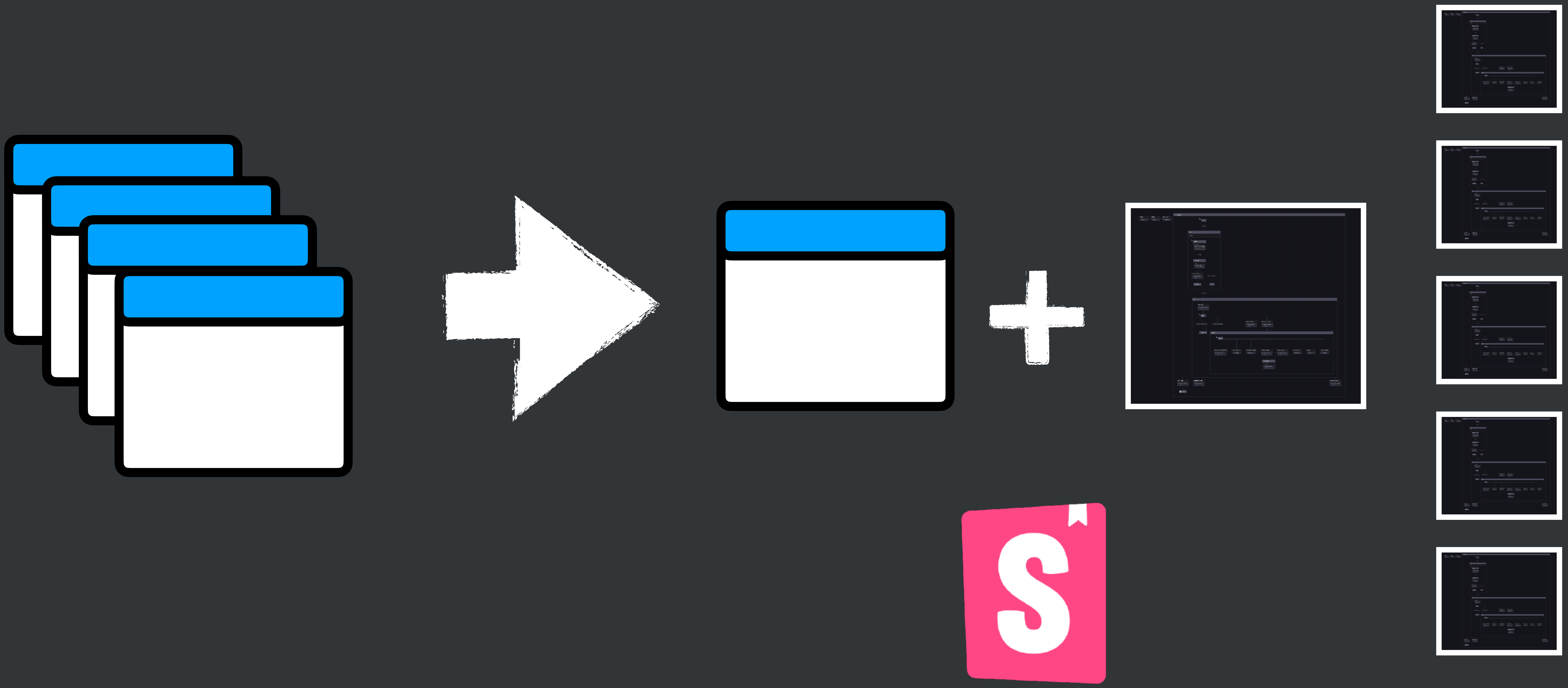
Offer builder flow

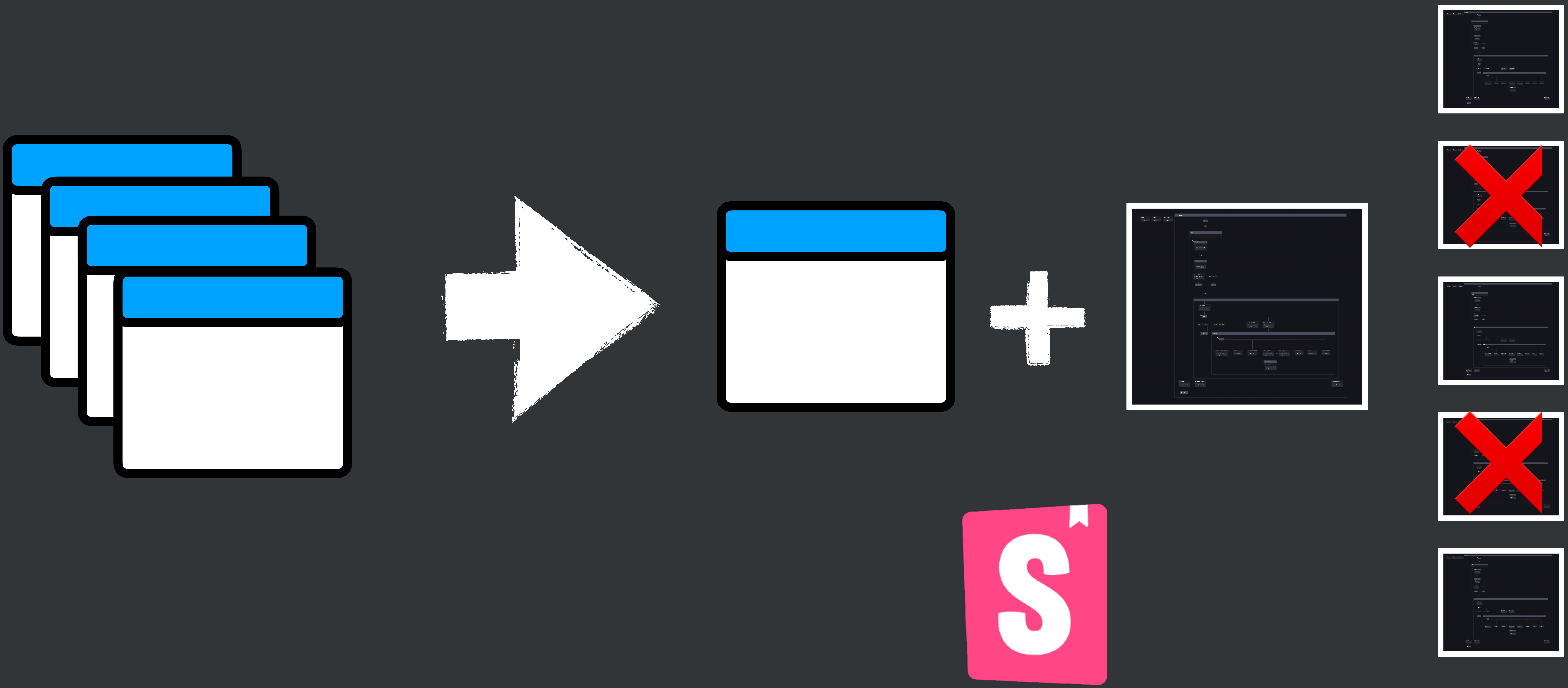
- Made up of several possible flows that depend on various factors
- Existing offer builder had completely separate code for every flow

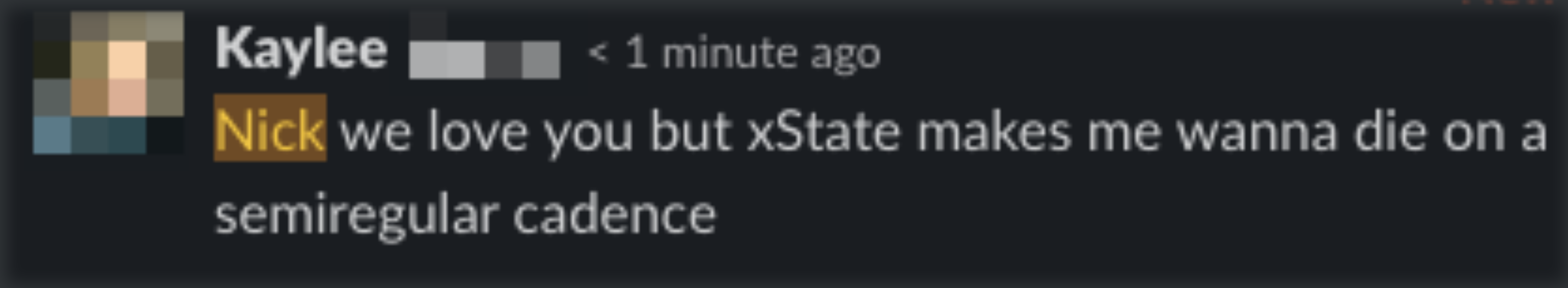
Good on us for not being too DRY!

  - Making updates meant changing code in 4+ places, and we were adding more
  - We were adding more flows while revamping 😱









Kaylee [redacted] < 1 minute ago

Nick we love you but xState makes me wanna die on a semiregular cadence

What went wrong?

Nothing! It’s still used in production

- 😰 Working on a large ‘JSON object’ can be tedious
- 😮‍💨 Terminology (actor vs. interpreter, cond vs. guard, services)
- 💀 We went a bit overkill in some places
- ⚛️ Can be difficult to use with React
    - A lot of business logic is in hooks, but our machines were not in a hook context

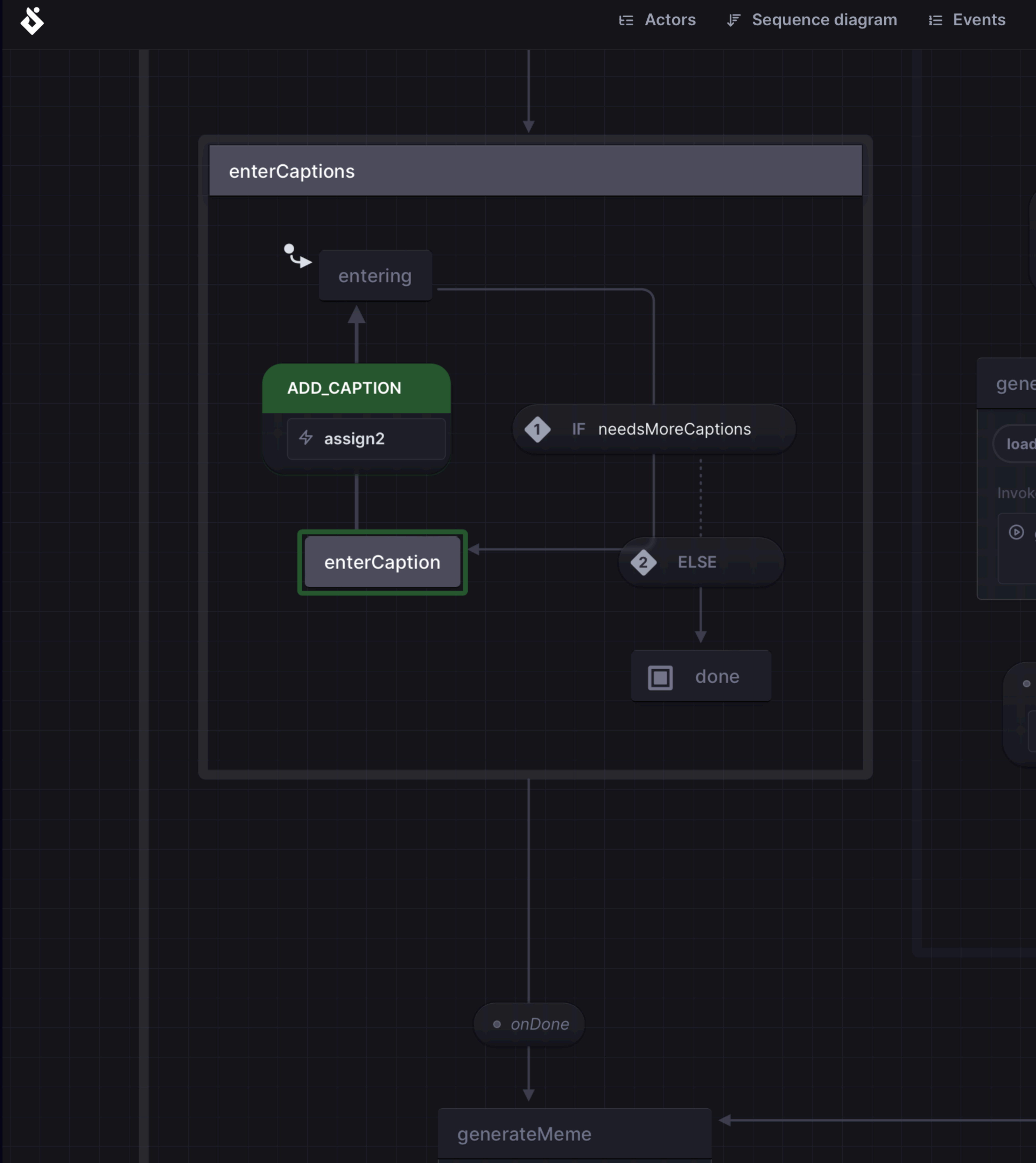
XState 5 fixes a lot of this!

So, Lets build a state machine!

XSTATE-MEME

CAPTION 1 / 2

Vim

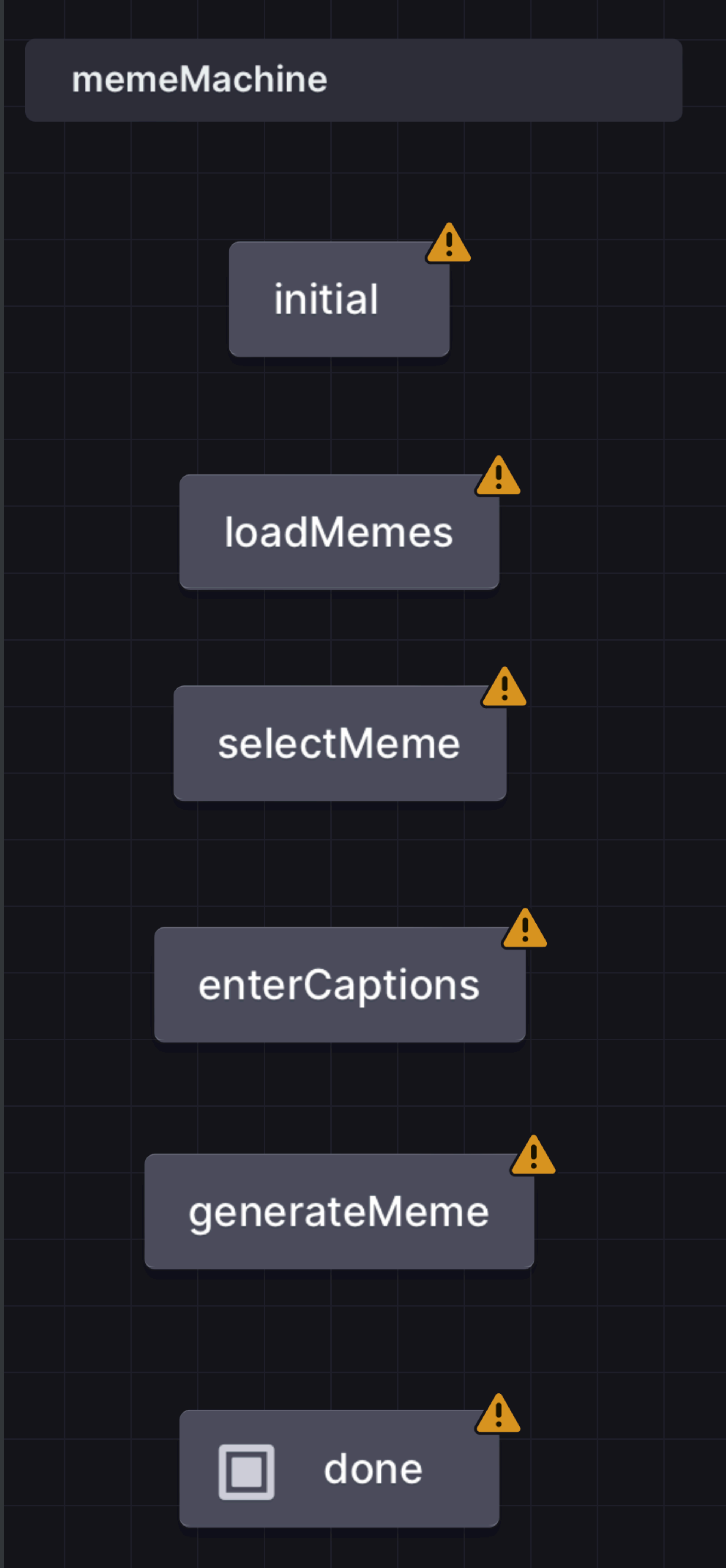


A literal meme machine

Caption a random meme

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

export const memeMachine = createMachine({
  id: "memeMachine",
  states: {
    initial: {}, // starting state
    loadMemes: {}, // fetch popular memes
    selectMeme: {}, // randomly select
    enterCaptions: {}, // enter captions
    generateMeme: {}, // generate meme
    done: { type: "final" }, // show meme
  },
});
```



Context

The infinite state

```
export interface MemeMachineContext {
  memes: Meme[];
  selectedMeme: Meme | null;
  captions: string[];
  clue: string | null;
  generatedMemeUrl: string | null;
  prompt: string | null;
}
```

- The data you'd like the state machine to store
  - General/supplemental data about the state machine
  - The data that cannot be codified into the machine itself
- The list of memes, the selected meme, the captions

The States

The finite part 😉

- Represents all possible states the machine can be in
- `done` is the final state
  - The state machine ends in this state
- Define the starting state with `initial`

```
initial: 'initial',
 states: {
   initial: {
     /* ... */
   },
   loadMemes: {
     /* ... */
   },
   selectMeme: {
     /* ... */
   },
   enterCaptions: {
     /* ... */
   },
   generateMeme: {
     /* ... */
   },
   done: { type: 'final' },
 }
```

Events

- All possible actions that can occur while in a state
- Events are quietly ignored if not defined
- Full control of how transitions from one state to another can happen

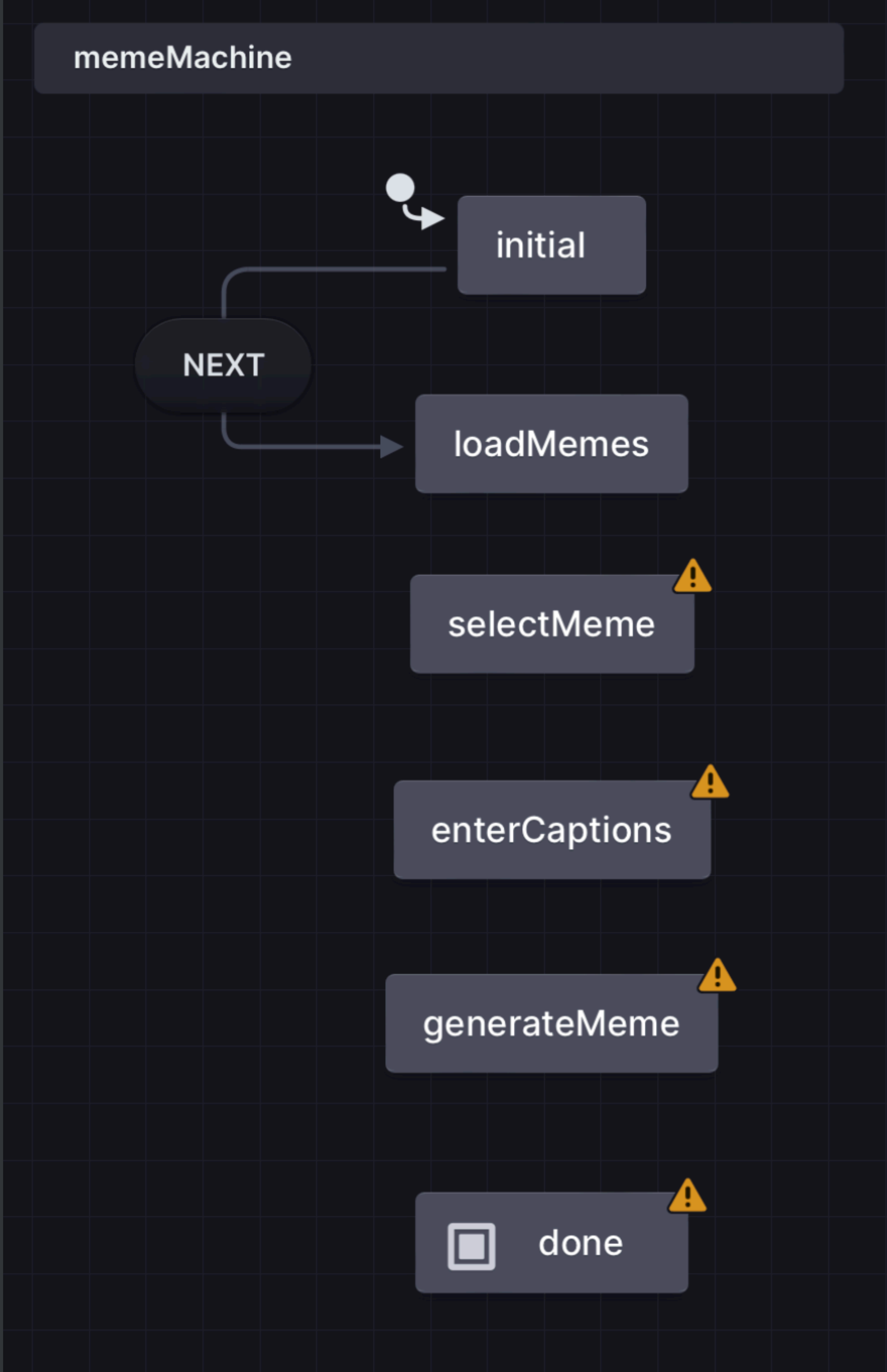
Meme events

```
export type MemeMachineEvent =
  | { type: 'ADD_CAPTION' | 'ADD_PROMPT'; value: string }
  | { type: 'START' | 'NEXT' | 'ENTER_PROMPT' | 'ENTER_CAPTIONS' | 'RETRY' };
```

- **START**, **NEXT** move to the next state (when defined)
- **ADD_CAPTION** provide a value which will be stored in the machine's context
- **ADD_PROMPT** provide a prompt to be stored in the machine's context
- **ENTER_PROMPT** and **ENTER_CAPTIONS** choose a different path through the machine
- **RETRY** moves back to a previously visited state (when defined)

Transitioning to loadMemes

```
export const memeMachine = createMachine({
  id: "memeMachine",
  initial: "initial",
  states: {
    initial: {
      on: {
        NEXT: "loadMemes",
      },
    },
    loadMemes: {
      /* ... */
    },
    // ...
  },
});
```

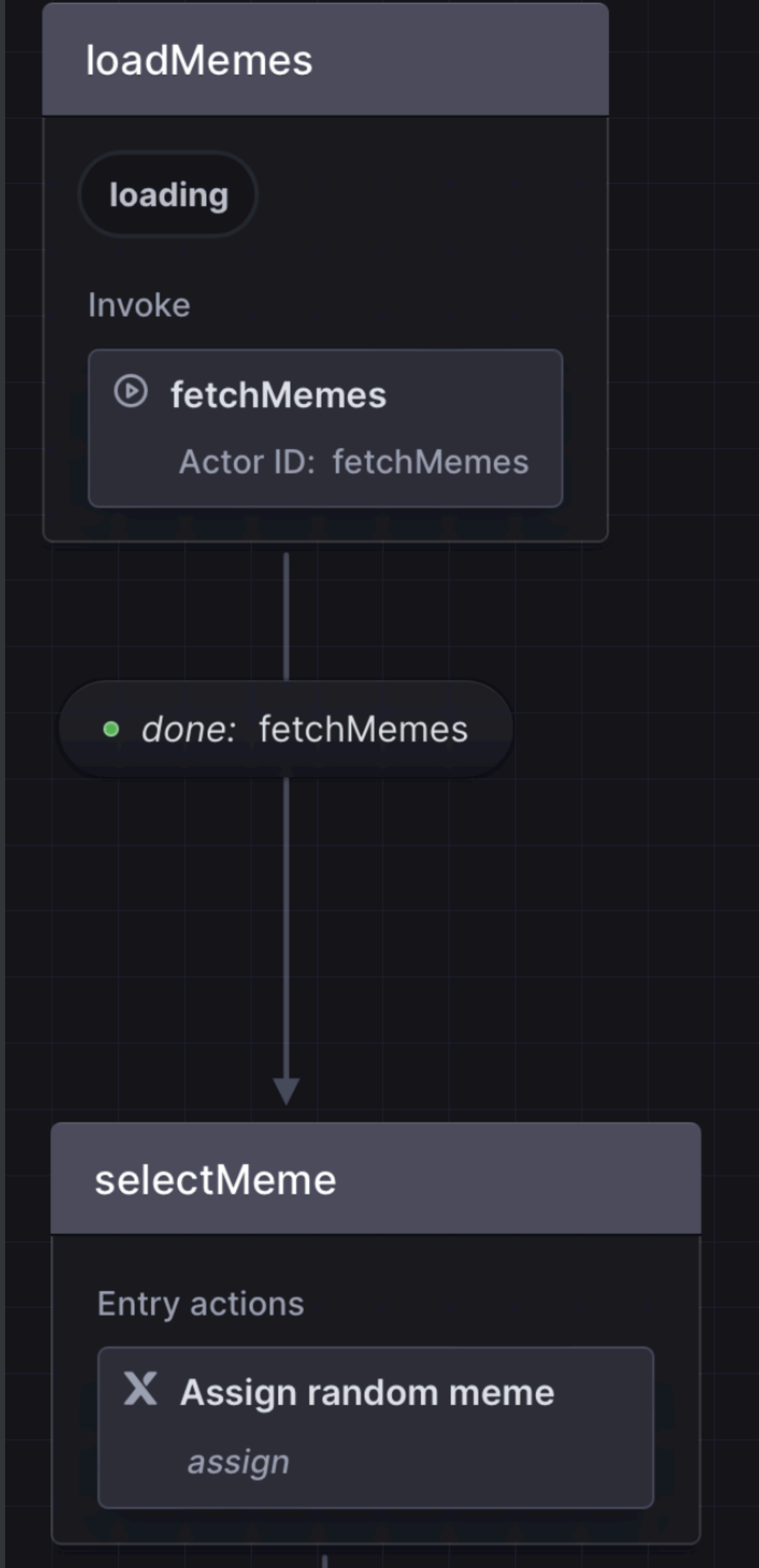


Invoking machines from machines

Promises are finite state machines, too!

```
loadMemes: {
  tags: ['loading'],
  invoke: {
    id: 'fetchMemes',
    src: 'fetchMemes',
    onDone: {
      target: 'selectMeme',
      actions: assign({
        memes: ({ event }) => event.output,
      }),
    },
  },
},
```

assign sets the meme array in the context

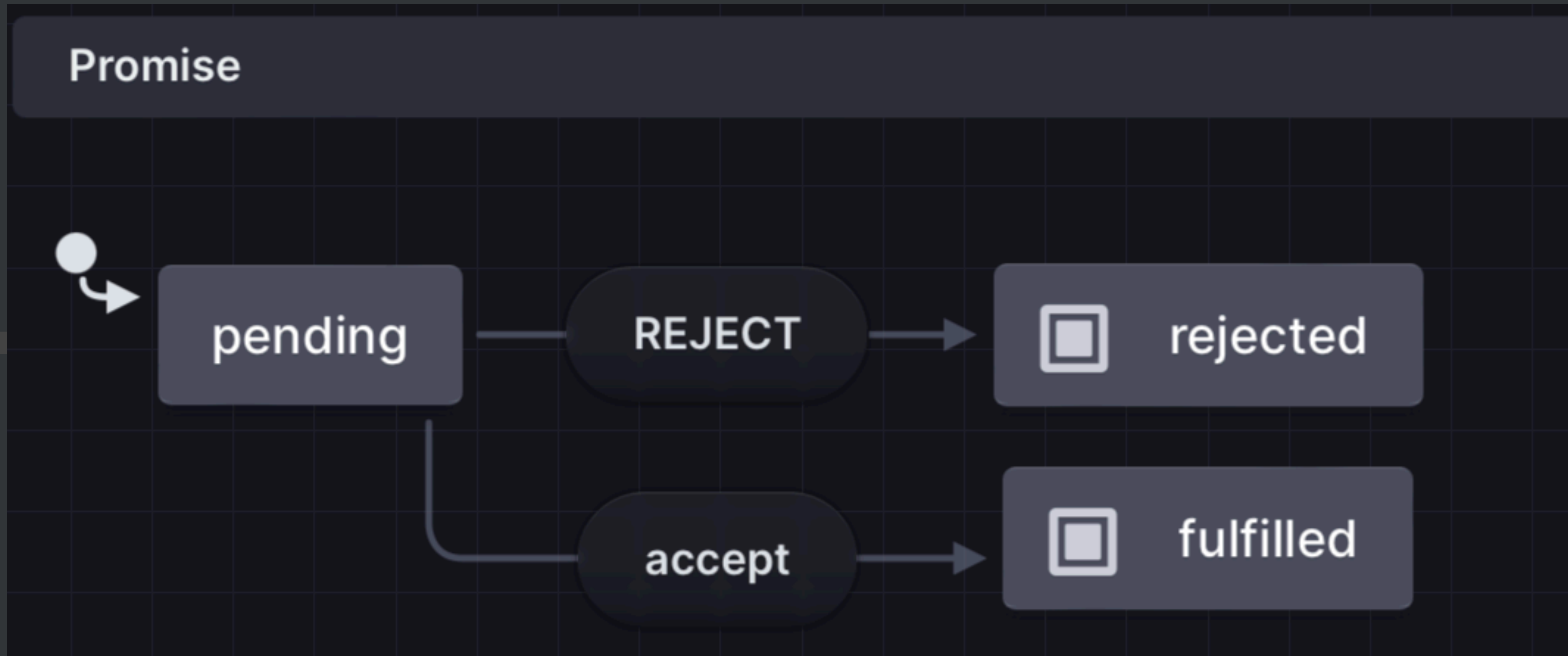


What's actually fetching the memes?

Invoking other actors

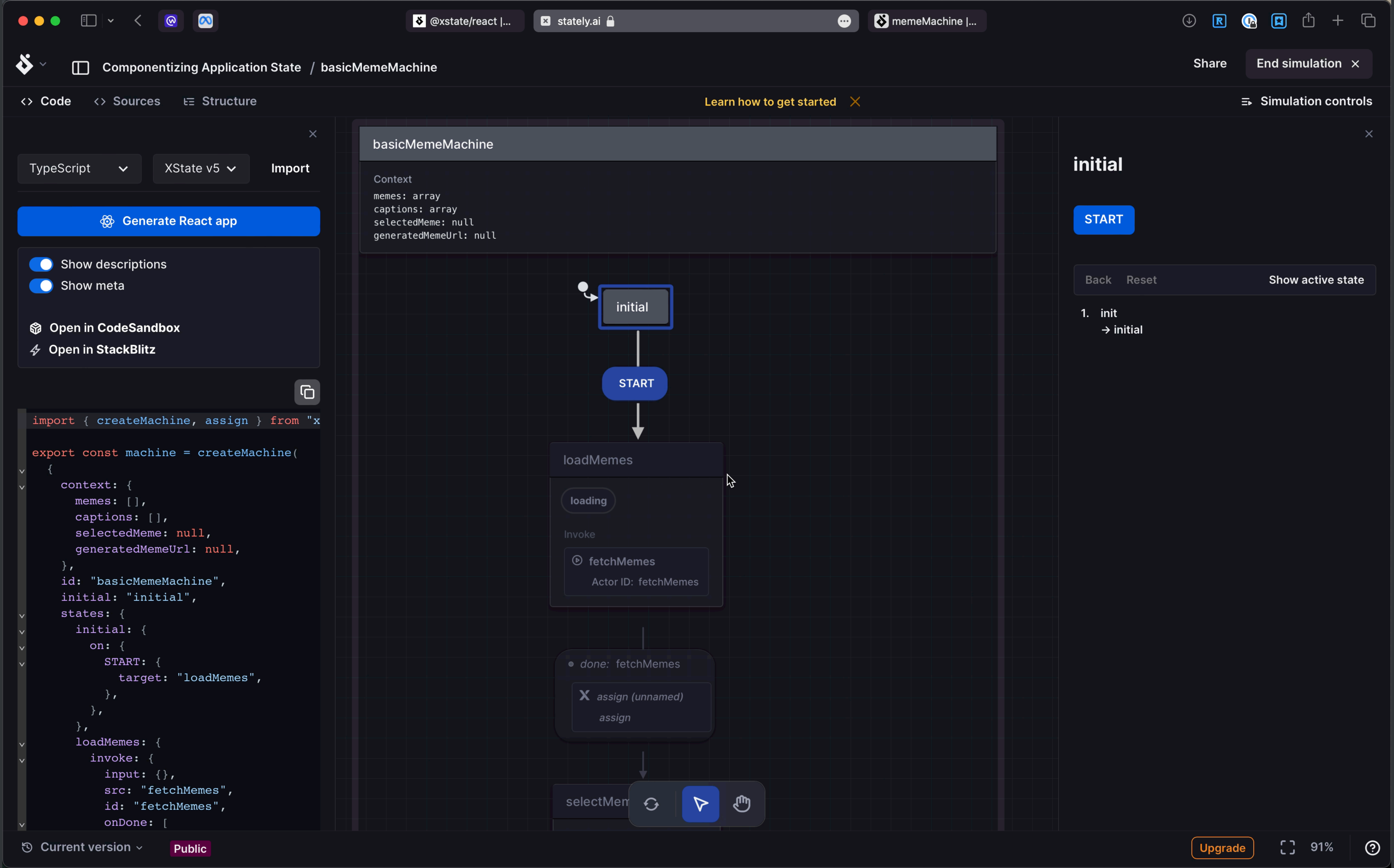
```
actors: {
  fetchMemes: fromPromise(() => fetchMemes()),
}
```

```
/**
 * Fetches memes from the Imgflip API
 */
export async function fetchMemes(): Promise<Meme[]> {
  const response = await fetch(`${API_BASE_URL}/get_memes`);
  const json = await response.json();
  return json.data.memes;
}
```



🧐 Reminder: we haven’t created any UI yet

We’re doing everything in Stately Studio / Storybook



Context

```
memes: array
captions: array
selectedMeme: null
generatedMemeUrl: null
```

1. init
   → initial

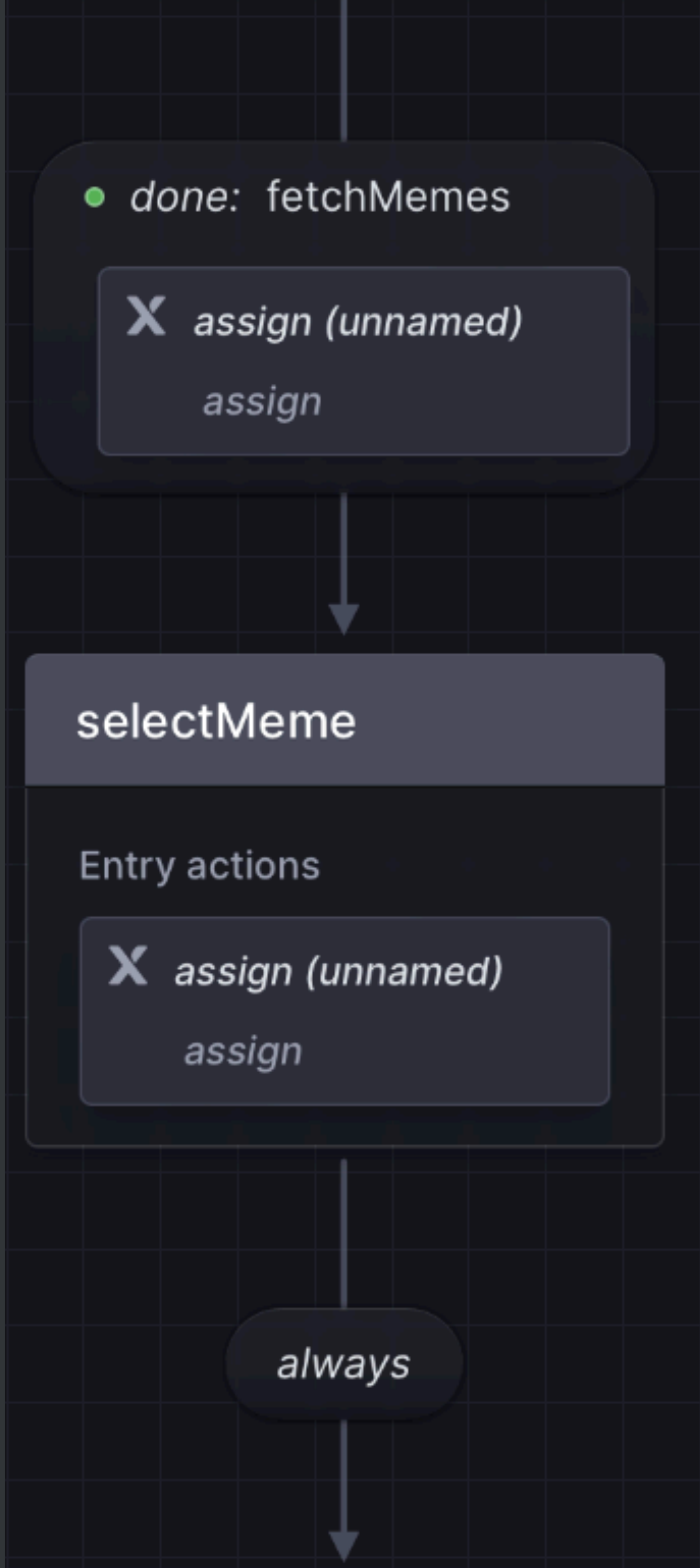
```
import { createMachine, assign } from "x

export const machine = createMachine(
  {
    context: {
      memes: [],
      captions: [],
      selectedMeme: null,
      generatedMemeUrl: null,
    },
    id: "basicMemeMachine",
    initial: "initial",
    states: {
      initial: {
        on: {
          START: {
            target: "loadMemes",
          },
        },
      },
      loadMemes: {
        invoke: {
          input: {},
          src: "fetchMemes",
          id: "fetchMemes",
          onDone: [
```

Selecting a random meme

entry and always automate the entire state

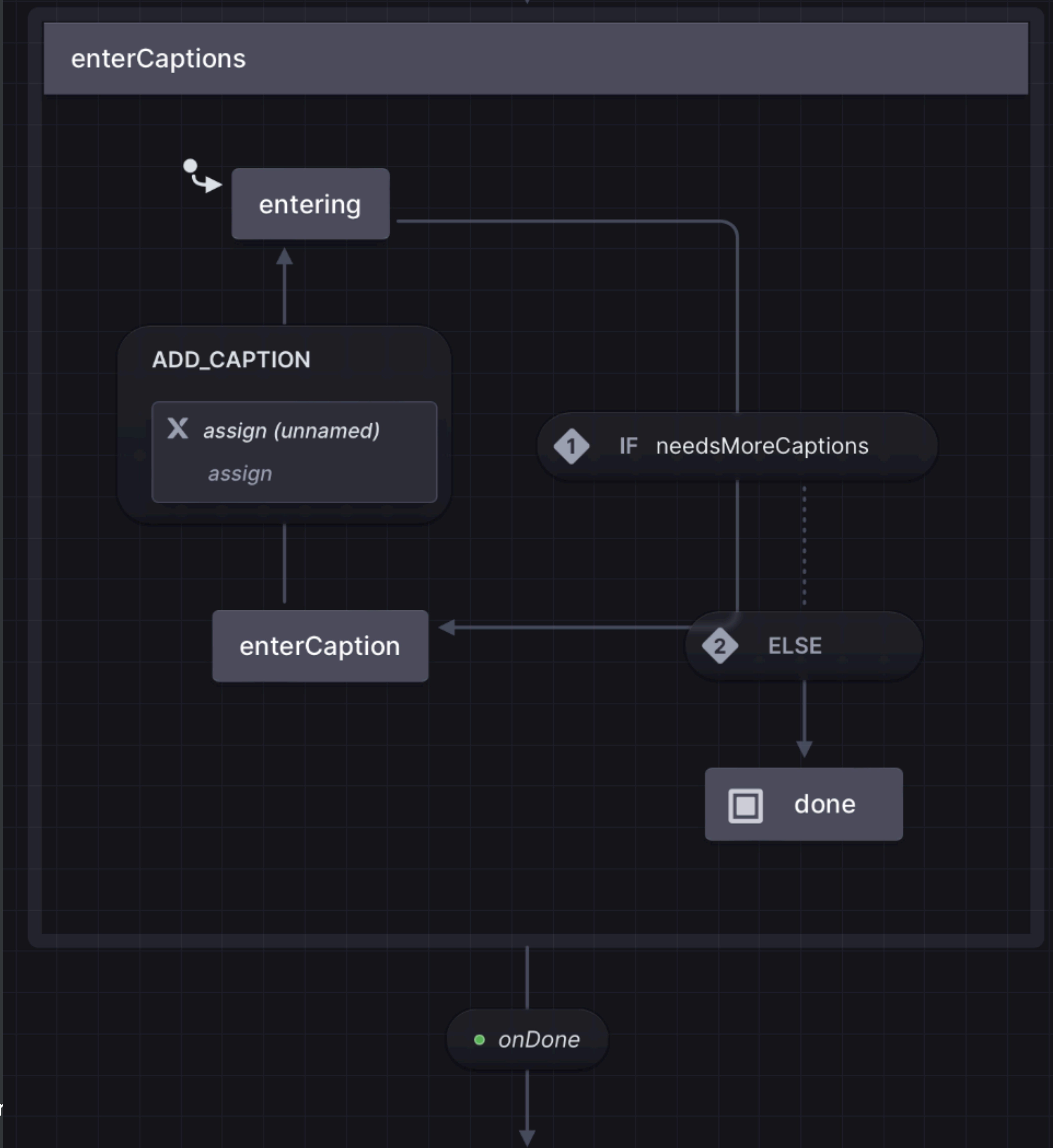
```
selectMeme: {
  entry: assign({
    selectedMeme: ({ context: { memes } }) =>
        memes[Math.floor(Math.random() * memes.length)] ?? null,
  }),
  always: 'enterCaptions',
},
```



States can have their own states 😱

- Allows for parallel or sequential states
- **onDone** defined to determine target when sub-machine has finished (reached its **final** state)

```
enterCaptions: {
  initial: 'entering',
  onDone: {
    target: 'generateMeme',
  },
  states: {
    entering: { /* ... */ },
    enterCaption: { /* ... */ },
    done: { type: 'final' },  },
},
```



`entering` state - Type Guards

```
guards: {
  needsMoreCaptions: ({ context: { selectedMeme, captions } }) =>
    selectedMeme!.box_count > captions.length,
},
```

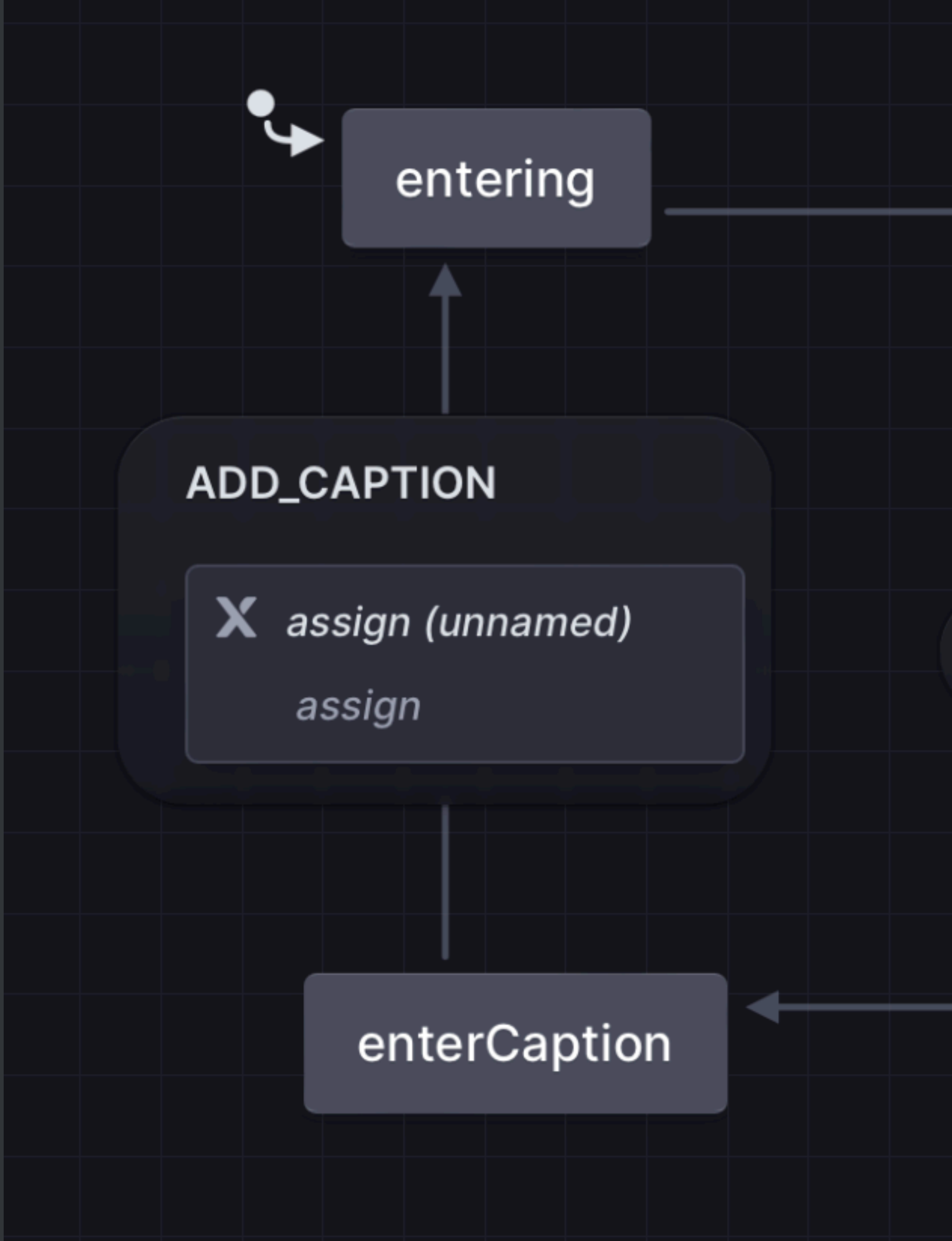
- Runs the first **`target`** if the **`guard`** (optional condition) is met
- Checks the next **`target`**, otherwise
- Guards are provided in the **`guards`** option

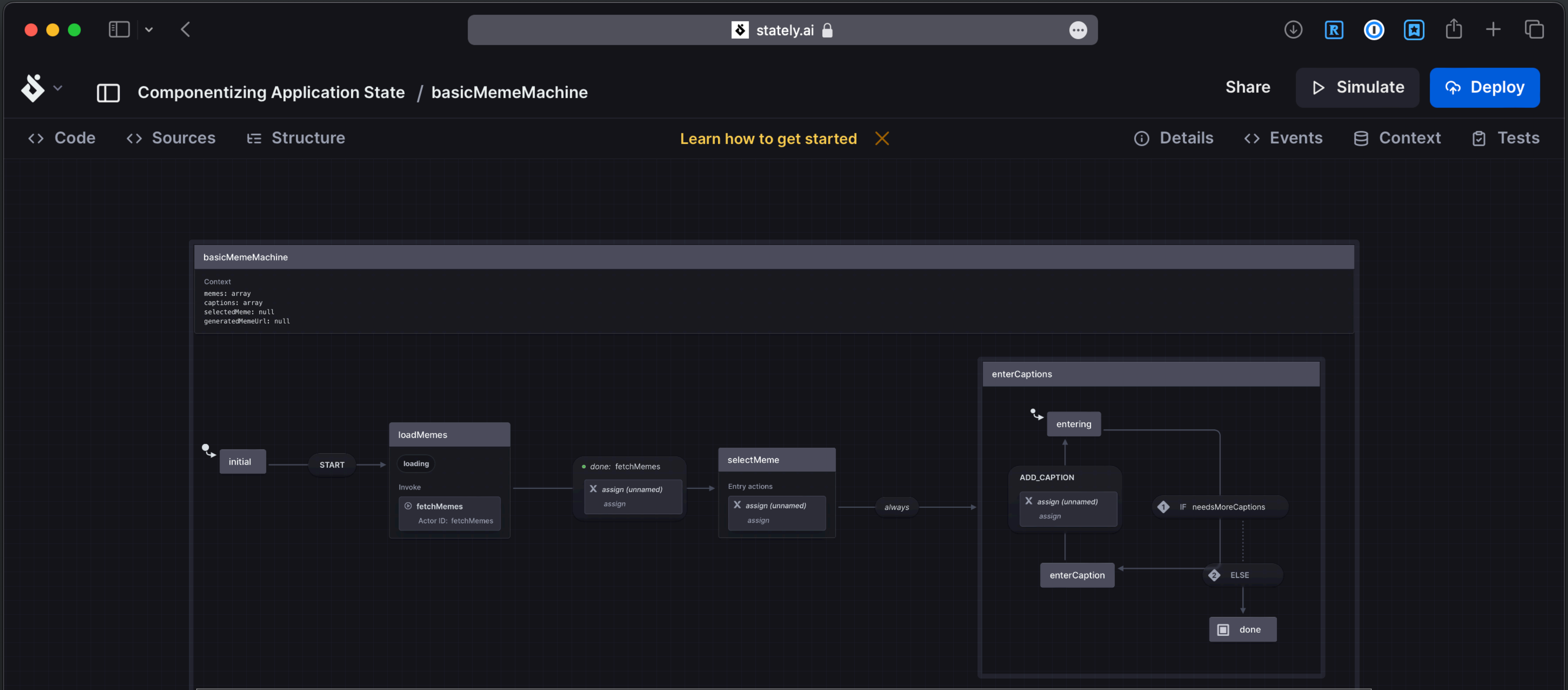
```
entering: {
  always: [
    {
      target: 'enterCaption',
      guard: 'needsMoreCaptions',
    },
    {
      target: 'done',
    },
  ],
},
```

Entering captions ✍️

Targets the entering state to loop back and check if more captions are needed

```
enterCaption: {
  on: {
    ADD_CAPTION: {
      actions: assign({
        captions: ({ context, event }) =>
                  context.captions.concat(
                    event.value ?? ‘DEFAULT'
                  ),
      }),
      target: 'entering',
    },
  },
},
```





Context

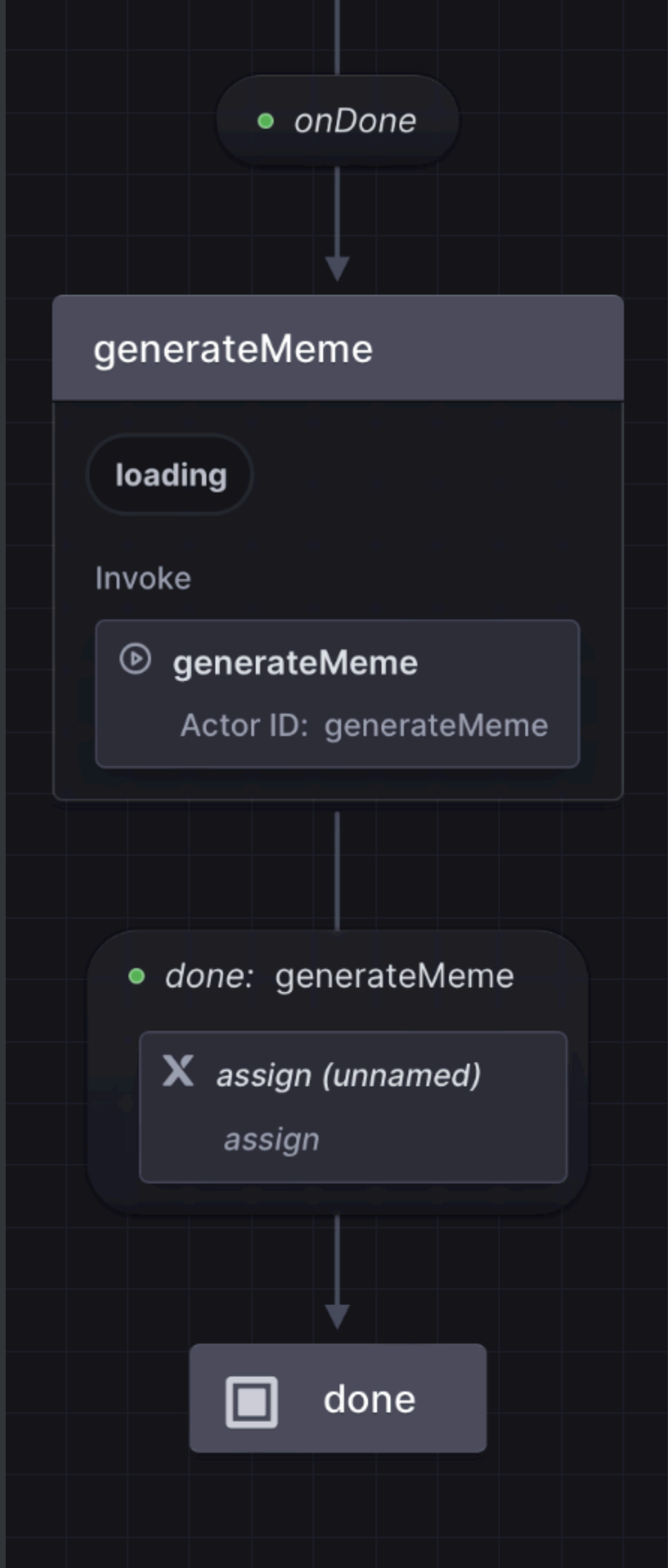
```
memes: array
captions: array
selectedMeme: null
generatedMemeUrl: null
```

No React yet, but look at my ***Component*** 😉

Generating the meme

Invoke the actor and then move to the done state

```
generateMeme: {
  tags: ['loading'],
  invoke: {
    id: 'generateMeme',
    src: 'generateMeme',
    input: ({ context: { selectedMeme, captions } }) =>
      ({ selectedMeme, captions }),
    onDone: {
      target: 'done',
      actions: assign({
        generatedMemeUrl: ({ event }) => event.output,
      }),
    },
  },
},
```



THAT Conference
localhost
Meme Machine
Initial Machine
Clue Machine
Final Machine
Actors
Sequence diagram
Events

XSTATE-MEME

Welcome to Meme Quest! Press the button below to get started.

START

basicMemeMachine

Context

```
memes: array
selectedMeme: null
captions: array
generatedMemeUrl: null
```

initial
START
loadMemes
loading
Invoke
fetchMemes
Actor ID: fetchMemes

Using XState from React

```js
import { createActorContext } from '@xstate/react';
import memeMachine from '../machines/basicMemeMachine.js';

// create an actor context
const MachineContext = createActorContext(memeMachine);

// export a provider component
export const MachineProvider = MachineContext.Provider;

// export useActorRef and useSelector hooks to directly
// access the machine's state and send it messages
export const useActorRef = MachineContext.useActorRef;
export const useSelector = MachineContext.useSelector;
```

#TMTOWTDI

useActorRef

- **ref** is the current actor reference
- **send** is how your React code can communicate / send events to the machine

```
const { ref, send } = useActorRef();

send({
 type: 'ADD_CAPTION',
 value: 'THAT Conference is awesome!',
});
```

useSelector

- Returns the current value from a snapshot of an actor, via callback
- Will only cause a re-render if the selected value changes

```
// access Context values
const selectedMeme = useSelector(snapshot => snapshot.context.selectedMeme);
const captions = useSelector(snapshot => snapshot.context.captions);

// check if a specific event can occur in the current state
const canAddCaption = useSelector(snapshot => snapshot.can('ADD_CAPTION'));
```

Let’s add a new state

Warning: LLMs are unpredictable

```
const defaultSystemMessages = [
  'Be as funny as possible. Lean into puns and wordplay.',
  'Make sure to never use curse words or offensive language.',
  'Do not repeat back anything I said to you.',
  'Seriously, be funny. This is a game. Make it fun.',
  'Even more seriously, don’t be offensive. Make it fun for everyone.
];
```

Let’s generate a meme clue

```
getClue: {
  tags: ['loading'],
  invoke: {
    id: 'getClue',
    src: 'getClue',
    input: ({ context: { selectedMeme } }) => ({ selectedMeme }),
    onDone: {
      target: 'showClue',
      actions: assign({
        clue: ({ event }) => event.output,
      }),
    },
  },
},
```

Showing the new UI

```
<>
  {state === 'showClue' && (
    <>
      <div className="text-center">
        <p className="p-3 text-2xl">Your Clue:</p>
        <p className="whitespace-pre p-3 text-5xl">{clue}</p>
        <div className="flex justify-center gap-2">
          <button
            type="button"
            className="rounded-lg border border-white p-3 text-lg"
            onClick={() => send({ type: 'ENTER_CAPTIONS' })}
          >
            ADD CAPTIONS
          </button>
        </div>
      </div>
    </>
  )}
</>
```

Still too hard? Let’s add another state!

Generate a meme from a prompt

```
enterPrompt: {
  initial: 'initial',
  onDone: { target: 'generateMeme' },
  states: {
    initial: { /* ... */ },
    generateCaptions: {
      tags: ['loading'],
      invoke: {
        id: 'generateCaptions',
        src: 'generateCaptions',
        input: ({ context: { meme, prompt } }) => ({ meme, prompt: prompt ?? '' }),
        onDone: {
          target: 'done',
          actions: assign({
            captions: ({ event }) => event.output,
          }),
        },
      },
    },
    done: { type: 'final' },
  },
},
```

Wrapping up

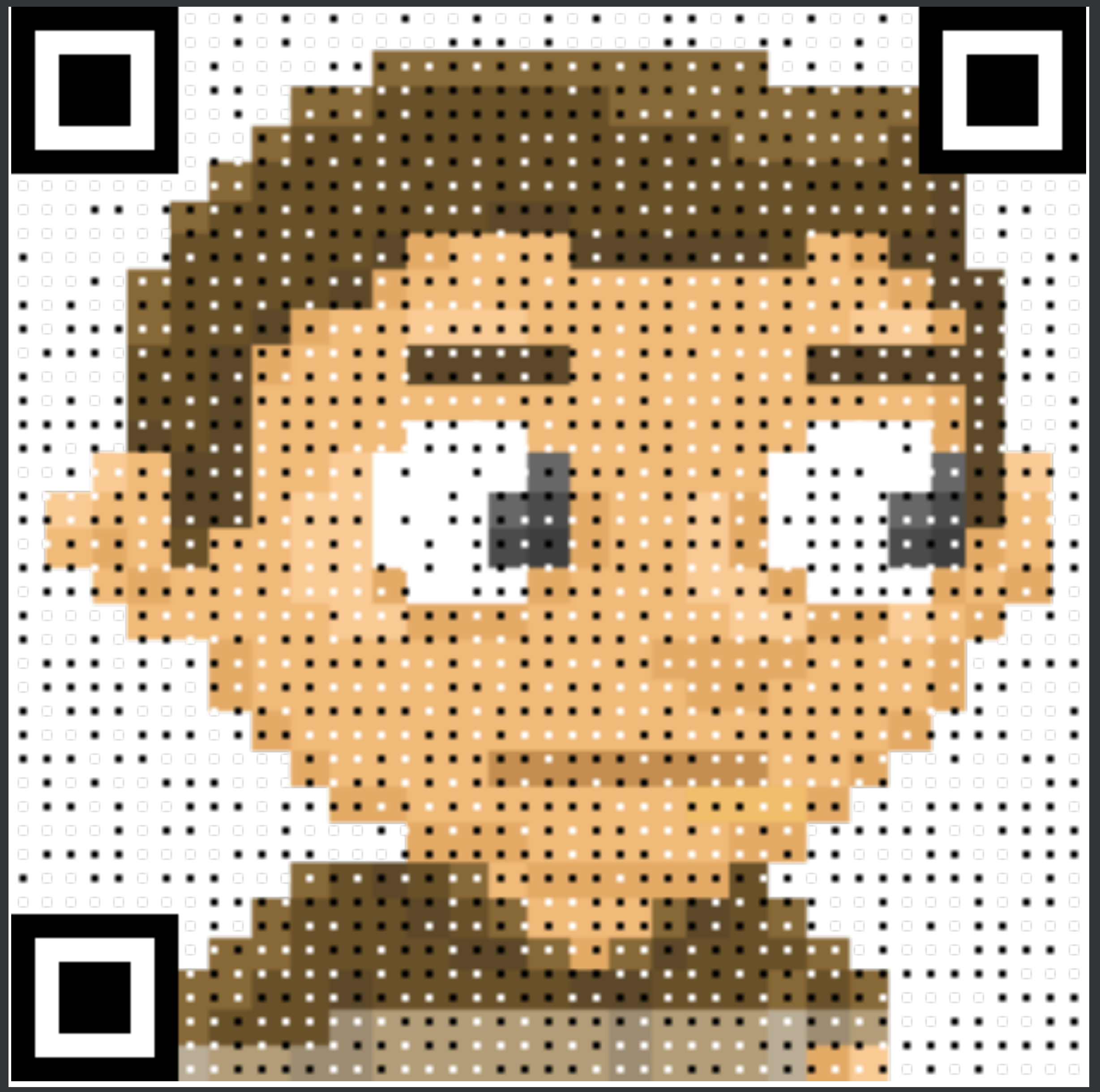
Componentizing state

- Treating your state like a component is a great way to keep your business logic and application state in properly synced
- Visualize business logic with your team
  - Storybook (with `storybook-xstate-addon`)
  - Stately Studio

nicknisi.com/talks/componentizing-application-state

PRESENTING AT ANY OTHER CONFERENCE
PRESENTING AT THAT CONFERENCE IN TEXAS
imgflip.com

THANKS!



nicknisi.com/talks/componentizing-application-state