Componentizing Application State

se## A little about me

Nick Nisi

– Staff Software Engineer at C2FO
– (A KC Company)
– Work remotely from Omaha, NE
– Cohost on JS Party
– Former Emcee
  – NEJS Conf (2015 - 2019)
  – TypeScript Conf US (2018 - 2021)

I like JavaScript and TypeScript a lot.

React is cool, too.

Your application state is
Too complex

Impossible States

– State that violates expected behavior of the system
– Cannot occur within the defined constraints and rules of the program
– Often results from programming error or incorrect assumptions
– Leads to unexpected behavior

In other words, they're nonsense states

A stop light

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

  const switchLight = () => {
    // ok this is really contrived 🥵
    const randomLight = lights[Math.floor(Math.random() * lights.length)];
    setLight(randomLight);
  };

  return (
    <div className="m-16">
      <div className={light === 'red' ? 'bg-red-600' : 'bg-red-100'} />
      <div className={light === 'yellow' ? 'bg-yellow-300' : 'bg-yellow-100'} />
      <div className={light === 'green' ? 'bg-green-600' : 'bg-green-100'} />
      <button onClick={switchLight}>
        Switch light
      </button>
    </div>
  );
};
```

Green-> Red -> Yellow -> Red -> Green 😱

The problem

```
  const switchLight = () => {
    // ok this is really contrived 🥵
    const randomLight = lights[Math.floor(Math.random() * lights.length)];
    setLight(randomLight);
  };
```

Random setting doesn't make sense for a stop light

The solution

```
const [light, setLight] = useState('red');
const [arrow, setArrow] = useState(undefined);

const lights = ['red', 'green', 'yellow'];
const [lightIndex, setLightIndex] = useState(0);
const switchLight = () => {
  const newIndex = (lightIndex + 1) % lights.length;
  setLightIndex(newIndex);
  setLight(lights[lightIndex]);
};
```

But we can keep getting more complex

Getting more complex

- add turn arrows?
- Have other factors?
  - Time of day
  - Day of week

WE CAN KEEP ADDING ON
BUT THE COMPLEXITY KEEPS GROWING

💸 Mo ~~Money~~ Variables Mo Problems[1]

```
  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)]);
  };
```

---

[1] Mo Money can also lead to mo problems.

#TMTOWTDI

There's more than one way to do it

More ways to handle state

The Context way

```
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>
);
```

It's a good idea, but it lacks any real structure for dealing with the state object.

Redux

```
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 develop our state like a Component? 🤔

XState

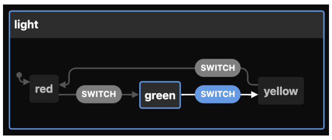
XState

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

export const lightMachine = createMachine({
  id: 'light',
  initial: 'red',
  states: {
    red: {
      on: { SWITCH: 'green' },
    },
    yellow: {
      on: { SWITCH: 'red' },
    },
    green: {
      on: { SWITCH: 'yellow' },
    },
  },
});
```

- – Finite states
- – Infinite states handled as private context
    - – such as number of jeopardy questions
- – Side-effects declarative and explicit
- – Framework agnostic
- – Transitions defined to only work in specific states

State machines render to state charts

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

export const lightMachine = createMachine({
  id: 'light',
  initial: 'red',
  states: {
    red: {
      on: { SWITCH: 'green' },
    },
    yellow: {
      on: { SWITCH: 'red' },
    },
    green: {
      on: { SWITCH: 'yellow' },
    },
  },
});
```



Let's Talk about

React Components

REACT MAKES IT EASY TO CREATE DECLARATIVE UIS

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

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

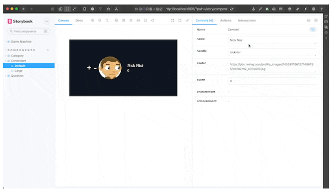
Developing UIs

- Working with declarative UIs is fast and fun
  - Define inputs (props) and outputs (what to render)
- Build a harness page to test the components by themselves without needing to spin up the entire app

Storybook

Storybook helps build components faster

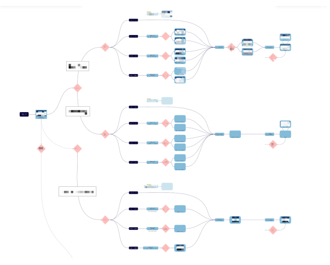
- Build components outside the app, in isolation
- Control inputs
- Streamline UI development and testing



Name	Control
name	Nick Nisi
handle	nicknisi
avatar	https://pbs.twimg.com/profile_images/1455971061271498755/olr282mQ_400x400.jpg
score	0
onIncrement	-
onDecrement	-

🤔 What if we could treat our app state the same way? 🤯

Secret: We kind of want to do this already 🤫



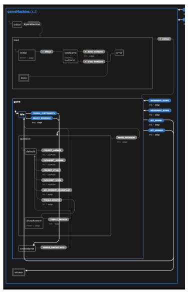
Componentizing Application State

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
- – Walk through the flow with non-technical stakeholders

State machines render to state charts

- – Visual representation of what's happening
- – always up-to-date (sorry, Miro)
- – Interactive!



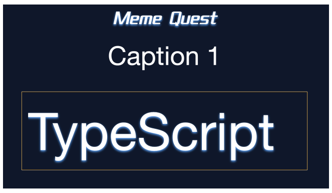
Your state comes a component 🤯

- – Render state charts directly from the actual application flow
- – Walk through the state and verify all possible routes from one state to another
- – Walk through entire application flow before the UI exists
- – Do all of this in Storybook

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

So, let's build a state machine!

Meme Game - Caption a random meme



A literal `meme` machine

```
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

This is the data that you'd like he state machine to store

- General/supplemental data about the states
- The data that cannot be codified into the machine itself

```
interface MemeMachineContext {
  memes: Meme[];
  selectedMeme: Meme;
  captions: string[];
  generatedMemeUrl: string;
}

export const memeMachine = createMachine<
  MemeMachineContext
>({
  context: { /* ... */ },
 // ...
});
```

The States

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

```
- Represents all possible states
```

- That's the `finite` part 😉
- Define the starting state with `initial`

Events

All possible actions that can occur while in a state

- quietly ignored if not defined
- Finite list of actions
- Full control

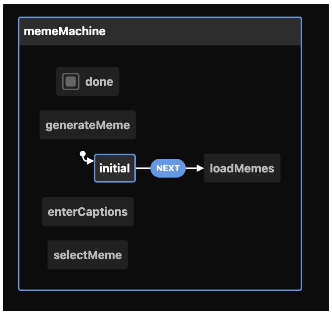
Meme Events

```
export type MemeMachineEvent =
 | { type: 'ADD_CAPTION'; value: string }
 | { type: 'NEXT' };

export const memeMachine = createMachine<MemeMachineContext, MemeMachine-
```

```
Event>({
  // ...
});
```

– `NEXT` - Move to the next state (when defined)
– `ADD_CAPTION`- provide a `value` which will be stored in the machine's context
– Events are fully-typed and can have payloads
– Defined as a **Discriminated Union**

MOVING FROM `INITIAL` TO `LOADMEMES`



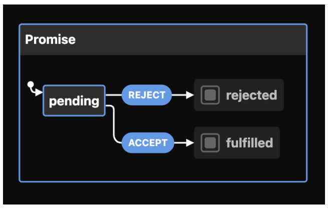
```
export const memeMachine = createMachine({

  id: 'memeMachine',
  initial: 'initial',
  states: {
    initial: {
      on: {
        NEXT: 'loadMemes'
      },
    },
    loadMemes: { /* ... */ },
    // ...
  },
});
```

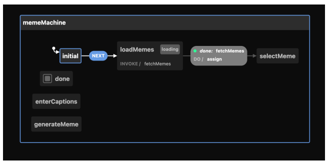
Invoking machines from machines

PROMISES ARE FINITE STATE MACHINES, TOO

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

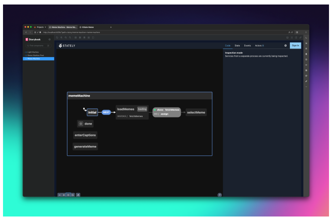
`assign` sets the meme array in the context.





🧐 REMINDER: WE HAVEN'T CREATED ANY UI

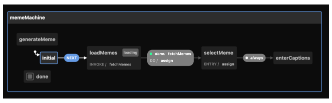
WE'RE DOING EVERYTHING IN STORYBOOK



Inspection mode
Services from a separate process are currently being inspected.

Selecting a random meme

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



`entry` and `always` automate the whole state

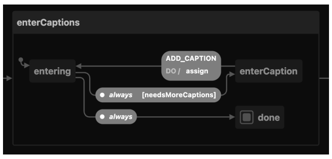
States can have their own states 😱

– Allows for sequential or sub-states

– `onDone` defined to determine target when sub-machine has finished

–

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



`ENTERING` STATE - TYPE GUARDS

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

– Runs the first `target` if the `cond`ition is met

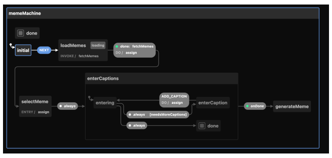
– Falls back to the next `target`, otherwise

IN THIS STATE, WE'RE ENSURING THAT IF WE NEED MORE CAPTIONS, WE ASK FOR THEM BEFORE MOVING ON

Entering Captions ✍️

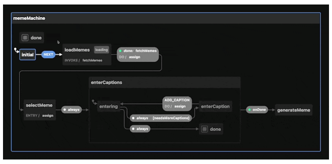
```
enterCaption: {
  on: {
    ADD_CAPTION: {
      actions: assign({
        captions: ({ captions }, event) => ([...captions, event.value]),
      }),
      target: 'entering',
    },
  },
},
```

TARGET THE `ENTERING` STATE TO LOOP BACK AND SEE IF WE NEED MORE CAPTIONS



No React but look at my "component" 😉

DID I MENTION THIS STATE CHART IS INTERACTIVE?



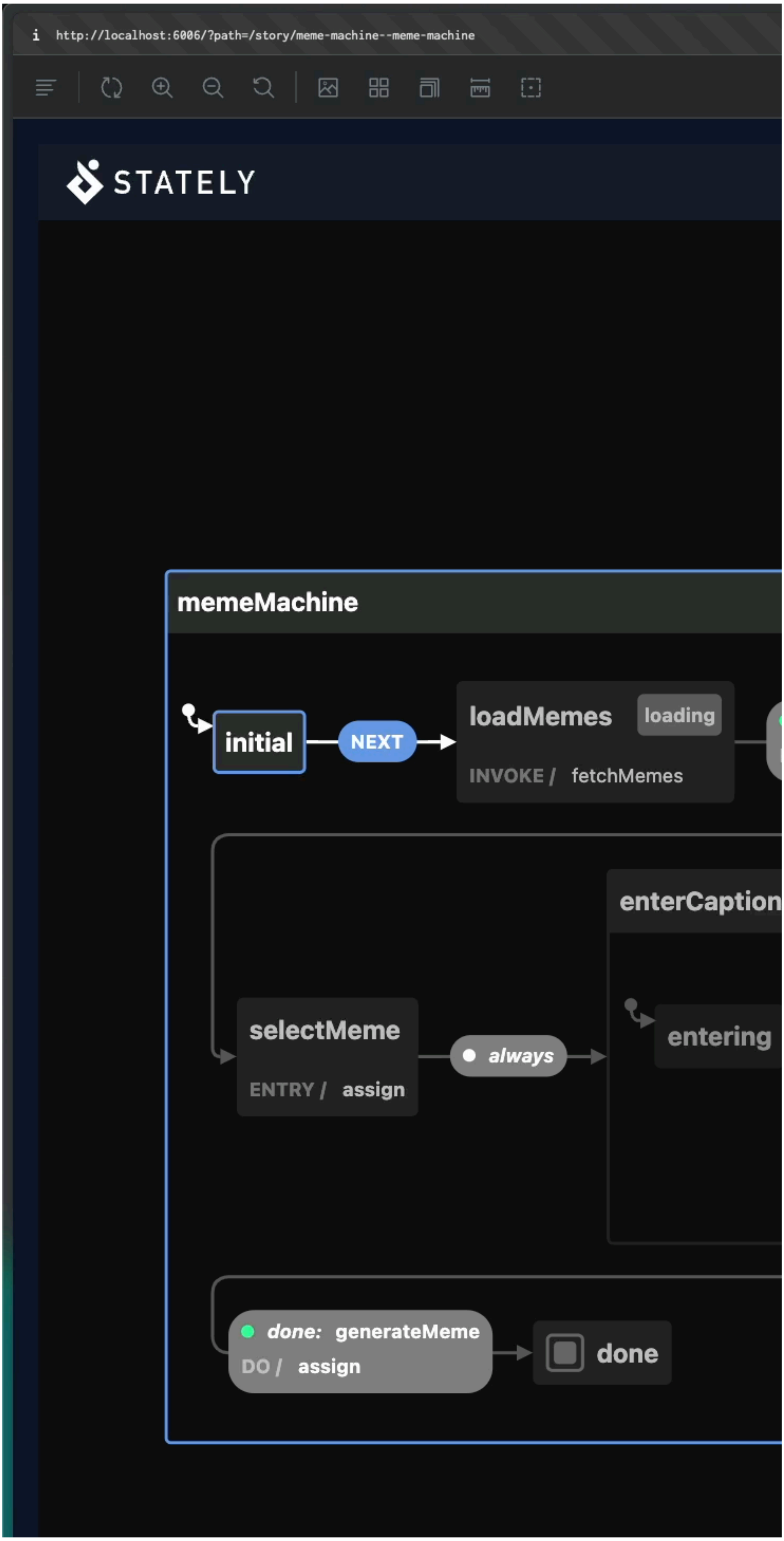
Generating the meme

```
generateMeme: {
  tags: ['loading'],
  invoke: {
    id: 'generateMeme',
    src: 'generateMeme',
    onDone: {
      target: 'done',
      actions: assign({
        generatedMemeUrl: (_, event) => event.data,
      }),
    },
  },
},
```

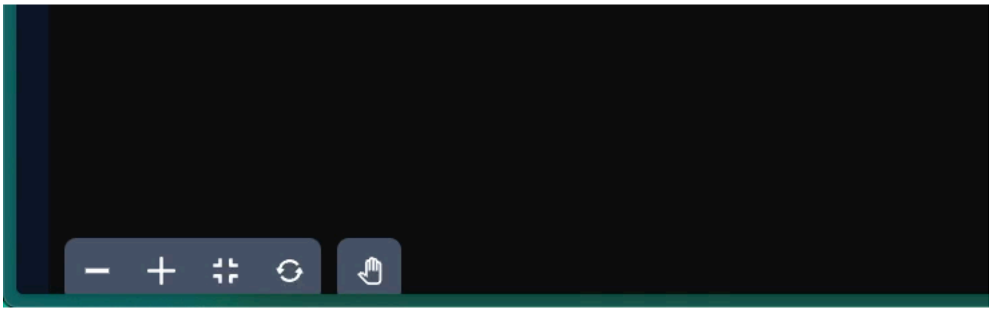
Defining services and guards

```
export const memeMachine = createMachine(
  {/ * ... */ },
  {
    guards: {
      needsMoreCaptions: ({ selectedMeme, captions }) => selectedMeme!.box_-
count > captions.length,
    },
    services: {
      fetchMemes: () => () => fetchMemes(),
      generateMeme:
        ({ selectedMeme, captions }) =>
        () =>
          captionMeme(selectedMeme!.id, captions),
    },
  },
);
```

Let's put it all together



http://localhost:6006/?path=/story/meme-machine--meme-machine



But how do we use this in React

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

// Create an Actor context
const MemeMachineContext = createActorContext(memeMachine);

// export a Provider component
export const MachineProvider = MemeMachineContext.Provider;

// export useActor and useContext hooks to access
// the machine's state and send it messages
export const useActor = MemeMachineContext.useActor;
export const useSelector = MemeMachineContext.useSelector;
```

`useActor`

```
const [state, send] = useActor();
send({
  type: 'ADD_CAPTION',
  value: 'KCDC Rocks 😎'
});
```

- `state` is the current state of the machine
- `send` is how your React code can communicate with the machine

`useSelector`

```
// The number of captions we currently have in state
const captionCount = useSelector(state => state.context.captions.length);
// Whether the current state has a `loading` tag
const loading = useSelector(state => state.tags.has('loading'));
```

- Returns the selected value from a snapshot of an actor
- Will only cause a pre-render if the selected value changes

Let's add a new state

```
generateClue: {
  tags: ['loading'],
  invoke: {
    id: 'generateClue',
    src: 'getClue',
    onDone: {
      target: 'showClue',
      actions: assign({
        clue: (_, event) => event.data,
      }),
    },
  },
},
{state === 'showClue' && clue && (
  <Centered>
    <div className="text-center">
      <p className="text-2xl p-3">Your Clue:</p>
      <p className="text-5xl p-3 whitespace-pre">{clue}</p>
      <button className="p-3 text-lg border-white border rounded-lg"
onClick={() => send('NEXT')}>
        ADD CAPTION(S)
      </button>
    </div>
  </Centered>
)}
```

But Nick, surely there must be downsides?

YES

Lessons from a year of XState

– 😰 Working on a large 'JSON object' can be tedious
– 😮‍💨 Terminology
– 💀 Overkill in some cases
–
⚛️ Can be difficult to interact with React
XState 5 is now in beta and addresses a lot!

Thanks!

– https://vim.dad
– github.com/nicknisi/xstate-meme
– https://xstate.js.org
– https://storybook.js.org