UTILIZING STATE MACHINES IN FRONT-END DEVELOPMENT

Joonas Tiala

Software Designer @ Solita

Speaker notes

- Hello there folks. My name is Joonas Tiala, I'm a Software Designer located in Oulu, doing mostly full stack development.
- Most recently I've been working with TypeScript and React on the front and Java on the back. In the recent years I've also done some nodejs backend, some AWS stuff and some DevOps stuff. Mostly React front tho.
- The title of the presentation is Utilizing state machines in front-end development. In my current project we have been doing just that quite a bit in last few years.

WHAT ARE FINITE STATE MACHINES?

Speaker notes

- Ok then, to the meat and potatoes. I will first give you a quick 101 lecture of state machines and statecharts and after that we will take a look at some real world examples on how to use state machines in React web app context using a library called XState. And actually also without using any library at all.
- Let us start with some theory. If you studied computer science at the university or some other school, you might remember this stuff from class. If not, you still might have heard the term Finite State Machine or FSM for short.
- So, what are the FSMs?

"an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition."

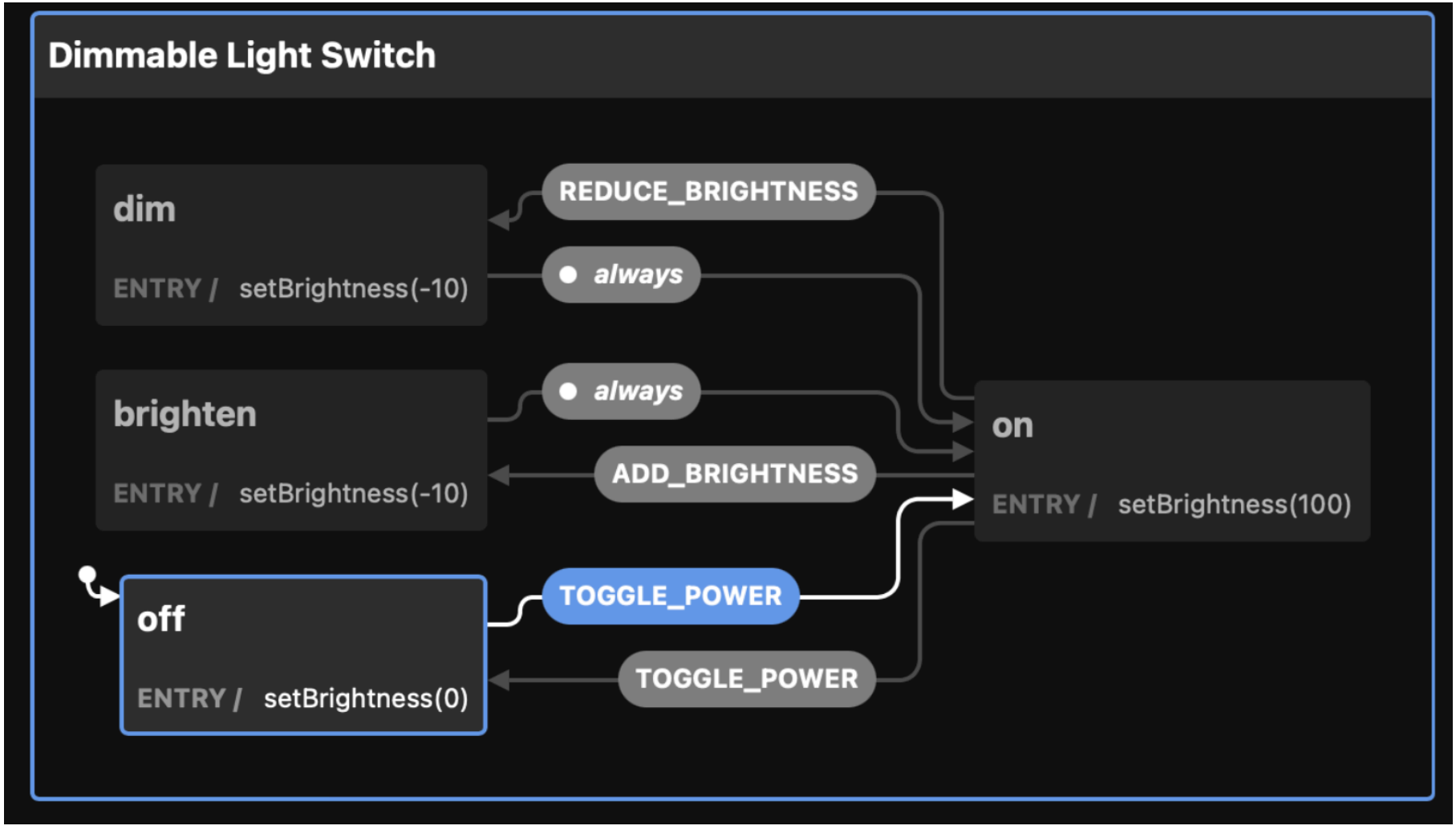
Speaker notes

- According to Wikipedia, FSM is (Read the definition out loud, press space to animate the highlights)
- The nice thing about state machines is that they are really convinient to present visually


Traffic Light
TIMER
green
TIMER
yellow
TIMER
red

Speaker notes

- For example, this is a state machine that represents a traffic light. Or at least a super simplified version of a traffic light.
- The machine can be in three different states, green, yellow or red. The machine can be in only one of these states at the time.
- The little dot with arrow on the left marks the initial state, in our case it's the green state. So, the green state is activated, when the machine is excecuted.
- The arrows between states are the possible transitions. In this very simple machine, every state have only one possible transition.
- And finally the pill-shaped boxes are the external inputs. Lets call them events from now on. We only have one event called timer, that triggers a transition from one state to another. Notice how the same event transitions to a different state based on the current state.
- This visualization is done with XState Visualizer, more on that later but this should be interactive, let's see: (Demo)


Dimmable Light Switch
dim
ENTRY / setBrightness(-10)
brighten
ENTRY / setBrightness(-10)
off
ENTRY / setBrightness(0)
on
ENTRY / setBrightness(100)
REDUCE_BRIGHTNESS
always
always
ADD_BRIGHTNESS
TOGGLE_POWER
TOGGLE_POWER

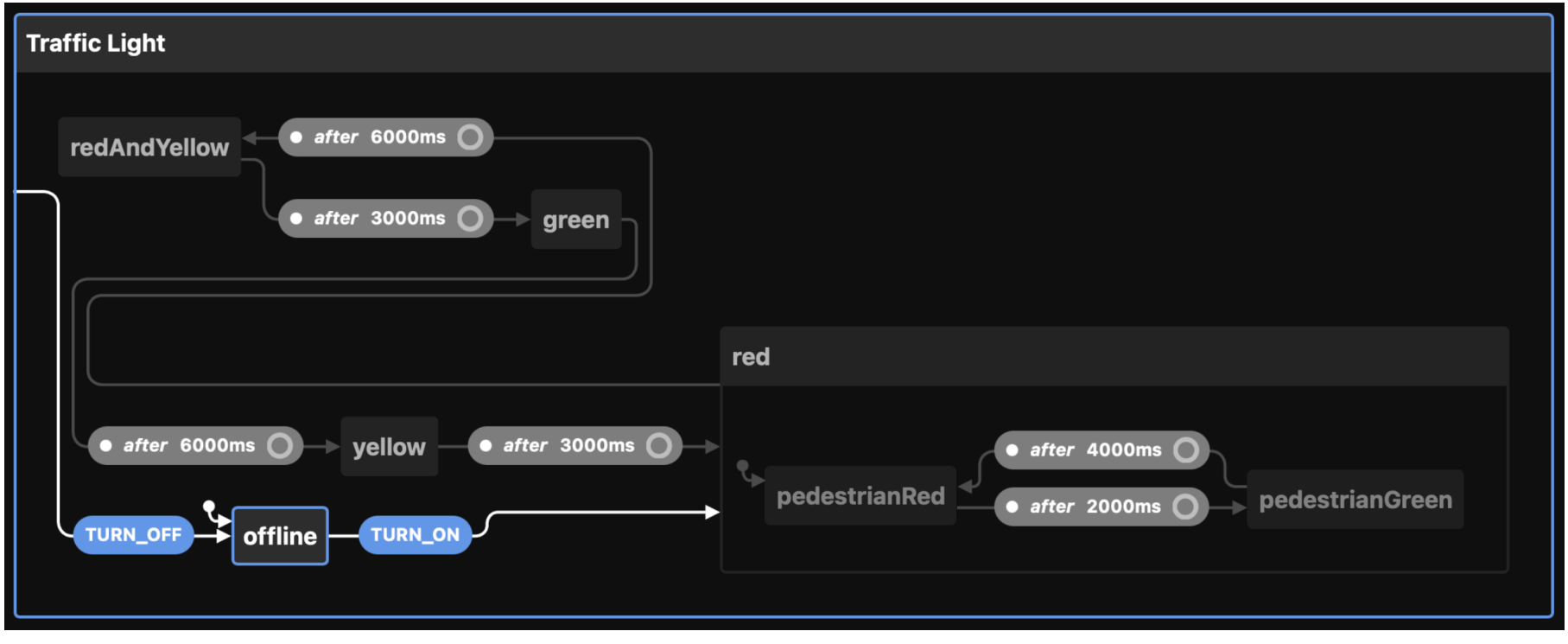
Speaker notes

- Here's an another example. This one is for a dimmable light switch.
- We a couple of new conecpts here. First, when we enter the intial state, off, we immediately trigger an action. Action are fire-and-forget side effects that doesn't affect the machine itself. This one is an entry action, meaning it will get triggerd when we enter the state. And calling the light switch API or something that controls the actual light in real world.
- We could also have actions for exiting a state or actions to be launched while the machine is transitioning to an another state.
- When we act on REDUCE_BRIGHTNESS or ADD_BRIGHTNESS events on the on state, we'll get transitioned to dim or brighten state. Those states have entry actions that control the actual light and after that we have a transient transiotion, which means that we'll transition to another state right away. So back to on state we go.

WHAT ARE STATECHARTS?

Speaker notes

- So, next up statecharts. What are statecharts?
- In many ways, statecharts are the “bigger brother” of state machines. A statechart is essentially a state machine that allows any state to include more machines, in a hierarchical fashion.


Traffic Light
redAndYellow
after 6000ms
after 3000ms
green
red
after 6000ms
yellow
after 3000ms
TURN_OFF
offline
TURN_ON
pedestrianRed
after 4000ms
after 2000ms
pedestrianGreen

Speaker notes

- If we take the traffic light example from before, and make it a bit more complicated, it might look something like this.
- The initial state of the machine is offline. When we send the turn on event to the machine, it transitions to red state. Ignore the stuff inside red for now.
- After 6 seconds timer, a transision to redAndYellow state occurs. At least here in Finland traffic lights light up red and yellow at the same time before going to green. From green we go to yellow and the back to red.
- The turn off event triggers transition to off state from any state of the machine. It's basically a global or root-level transition that XState allows us to do. Its the same as giving every state a transition to the off state, it just makes our job a bit easier and the visual representation a bit clearer.
- Lastly, the red state is it's own little state machine, with its own states, initial state, events and so on. When the parent machine reaches red, the child machine gets executed and starts from its intial state.
- So basically when the main traffic lights are red, the pedestrian traffic lights become green after a bit of a delay for safety. Then they turn red, and the parent machine continues.

WHEN, WHY AND HOW TO USE STATE MACHINES IN FRONT-END CODE?

Speaker notes

- Great. Now you know some of the theory behind state machines. Now what? How can you utilize this new knowledge in your front-end code?
- Lets create a React component for demonstration.

```jsx
const SearchForm = () => {

  return (
    <div />
  );
};
```

Speaker notes

- Let's start with this empty React component called SearchForm

```jsx
const SearchForm = () => {
  const [keyword, setKeyword] = useState("");
  const [results, setResults] = useState([]);

  const search = () => {
    fetchResults(keyword)
      .then((results) => setResults(results));
  };

  return (
    <div>
      <input onChange={(event) => setKeyword(event.target.value)} />
      <button onClick={search}>
        Search
      </button>

      {results.length > 0 && results.map((result) => <p>{result}</p>)}
      {results.length === 0 && <p>Sorry! No results.</p>}
    </div>
  );
};

```

Speaker notes

- We will probably want to store the keyword and the search results somewhere, lets use useState for that.
- We'll need to execute the search somehow. Lets add a function that fetches the search results from the API and stores them in our components state.
- We need a few form elements, an input for the search field and a button to execute the search.
- Finally, we want to display the results, if any and if not, print out a message for that.

```jsx
const SearchForm = () => {
  const [keyword, setKeyword] = useState("");
  const [results, setResults] = useState([]);
  const [isLoading, setIsLoading] = useState(false);

  const search = () => {
    setIsLoading(true);

    fetchResults(keyword)
      .then((results) => setResults(results))
      .finally(() => setIsLoading(false));
  };

  return (
    <div>
      <input onChange={(event) => setKeyword(event.target.value)} />
      <button onClick={search} disabled={isLoading}>
        Search
      </button>

      {isLoading && <p>Searching...</p>}
      {results.length > 0 && results.map((result) => <p>{result}</p>)}
      {results.length === 0 && <p>Sorry! No results.</p>}
    </div>
  );
};

```

Speaker notes

- How about loading state? We will probably want to show a spinner or something while the search is executing.
- Let's do that by adding a new useState call, setting the loading state in the search function and disabling the button and showing the spinner while the app is loading.

```jsx
const SearchForm = () => {
  const [keyword, setKeyword] = useState("");
  const [results, setResults] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  const [hasError, setHasError] = useState(false);

  const search = () => {
    setIsLoading(true);
    setHasError(false);

    fetchResults(keyword)
      .then((results) => setResults(results))
      .catch(() => setHasError(true))
      .finally(() => setIsLoading(false));
  };

  return (
    <div>
      <input onChange={(event) => setKeyword(event.target.value)} />
      <button onClick={search} disabled={isLoading}>
        Search
      </button>

      {isLoading && <p>Searching...</p>}
      {hasError && <p>Oh no!</p>}
      {results.length > 0 && results.map((result) => <p>{result}</p>)}
      {results.length === 0 && <p>Sorry! No results.</p>}
    </div>
  );
};

```

Speaker notes

- Error handling! We need error handling! Lets do this, this and this. Now we are handling the errors.
- How does the search form look like before the user has searched anything? It just says "Sorry, no results", right? I'm sure we don't want this. We could add a new bit of state, that keeps count if the user has searched or not, but lets not do that quite yet.
- Can the isLoading and hasError boolean flags be both true at the same time? How about the new hasSearched flag? Which combinations of these three are possible and what should we print out for each combination?
- Every boolean flag or a piece of state we are adding multiplies the possible or impossible combinatios. Every piece makes the component more complicated and more difficult to comprehend. Still, it's our job to make sure the app functions well in every situation.
- So, how could we fix this?

STATE MACHINES TO THE RESCUE!

Speaker notes

- State machines! You knew it!

```
const states = {
  INITIAL: "INITIAL",
  LOADING: "LOADING",
  FAILURE: "FAILURE",
  NO_RESULTS: "NO_RESULTS",
  SHOW_RESULTS: "SHOW_RESULTS",
};

const SearchForm = () => {
  const [state, setState] = useState(states.INITIAL);
  const [keyword, setKeyword] = useState("");
  const [results, setResults] = useState([]);

  const search = () => {
    setState(states.LOADING);

    fetchResults(keyword)
      .then((results) => {
        setResults(results);
        setState(
          results.length > 0 ? states.SHOW_RESULTS : states.NO_RESULTS
        );
      })
      .catch((error) => setState(states.FAILURE));
  };

  return (
    <div>
      <input onChange={(event) => setKeyword(event.target.value)} />
      <button onClick={search} disabled={state === states.LOADING}>
        Search
      </button>

      {state === states.INITIAL && <p>Hi! Enter keyword please.</p>}
      {state === states.LOADING && <p>Searching...</p>}
      {state === states.FAILURE && <p>Oh no!</p>}
      {state === states.NO_RESULTS && <p>Sorry! No results.</p>}
      {state === states.SHOW_RESULTS &&
        results.map((result) => <p>{result}</p>)}
    </div>
  );
};

```

Speaker notes

- First, let's define the states the component can be in. We'll use this object for easier autocompletion. If we were using TypeScript, an enum would be even better.
- Now we can use just one useState for the state and one for keyword and results each, like before.
- We'll always just set the state to one of the predefined values.
- Now generating the output is much easier since we know all the possible states the component can be in. The rendering here isn't that pretty to be honest, but that's not the point here. We know now that we are handling all the cases.
- Much better already. But it's not quite a state machine yet.

“an abstract machine that can be in ~~exactly one~~ of a finite number of states at any given time. The FSM can ~~change from one state to another~~ in response to some ~~external inputs~~; the change from one state to another is called a ~~transition~~. An FSM is defined by a list of its ~~states~~, its ~~initial state~~, and the conditions for each transition.”

Speaker notes

- Let's refer back to the definition of a FSM.
- We know that our component now can be only in one state of a finite set.
- We transition to different states based on events.
- Our initial state is set.
- But we haven't defined the conditions for each trasition yet.

```
const states = {
  INITIAL: "INITIAL",
  LOADING: "LOADING",
  FAILURE: "FAILURE",
  NO_RESULTS: "NO_RESULTS",
  SHOW_RESULTS: "SHOW_RESULTS",
};

const transitions = {
  INITIAL: ["LOADING"],
  LOADING: ["FAILURE", "NO_RESULTS", "SHOW_RESULTS"],
  FAILURE: [],
  NO_RESULTS: [],
  SHOW_RESULTS: [],
};

const validTransition = (before, after) => transitions[before].includes(after);

const SearchForm = () => {
  const [state, setState] = useState(states.INITIAL);
  const [keyword, setKeyword] = useState("");
  const [results, setResults] = useState([]);

  const search = () => {
    if (validTransition(state, states.LOADING)) {
      setState(states.LOADING);

      fetchResults(keyword)
        .then((results) => {
          setResults(results);
          setState(
            results.length > 0 ? states.SHOW_RESULTS : states.NO_RESULTS
          );
        })
        .catch((error) => setState(states.FAILURE));
    }
  };

  return (
    <div>
      <input onChange={(event) => setKeyword(event.target.value)} />
      <button onClick={search} disabled={state === states.LOADING}>
        Search
      </button>

      {state === states.INITIAL && <p>Hi! Enter keyword please.</p>}
      {state === states.LOADING && <p>Searching...</p>}
      {state === states.FAILURE && <p>Oh no!</p>}
      {state === states.NO_RESULTS && <p>Sorry! No results.</p>}
      {state === states.SHOW_RESULTS &&
        results.map((result) => <p>{result}</p>)}
    </div>
  );
};

```

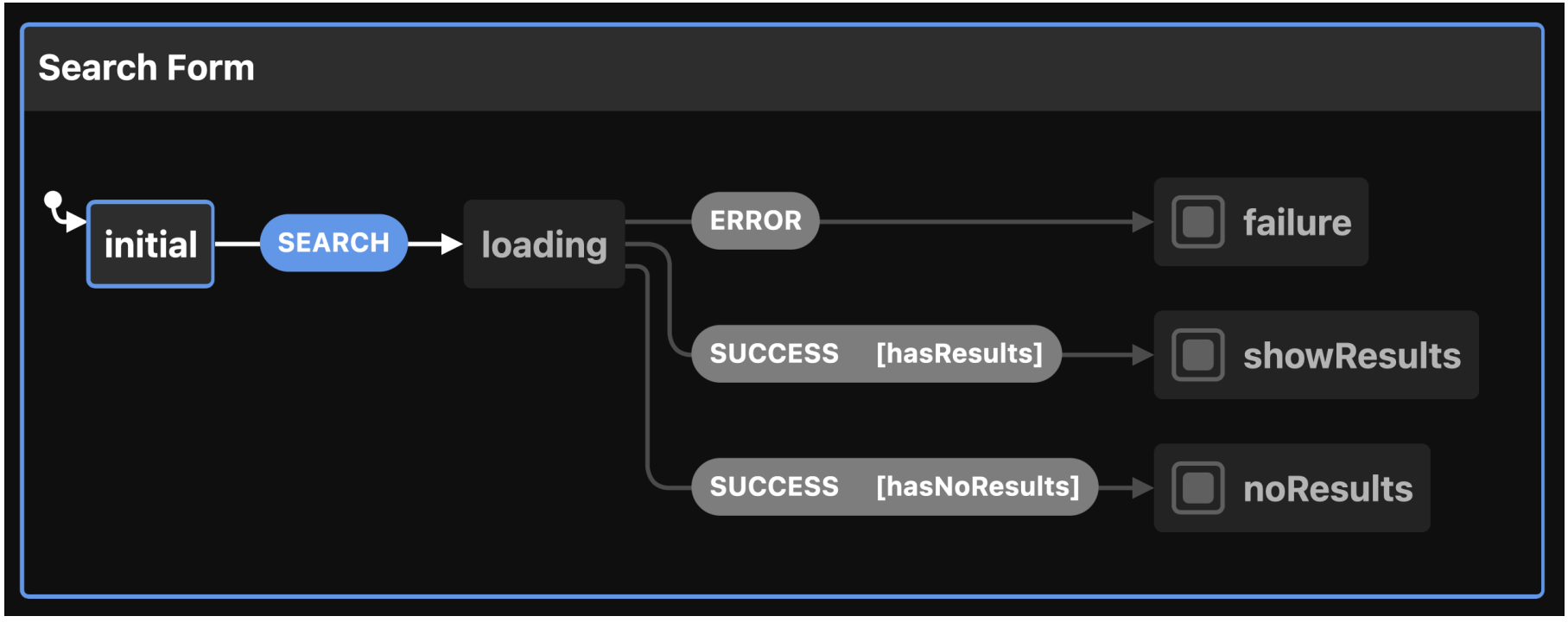
Speaker notes

- We could fix that by doing something like this.
- Lets define to which state we can transition from every state.
- To make our job easier, lets add a utility function to check if a transition from state a to b is valid.
- And then check if the transition to loading is available before transitioning. We should also do the check for the other transitions but lets leave that for another day.
- Nice! Now we pretty much have a finite state machine in place and the code is much easier comprehend, extend and refactor.

ENTER XSTATE

Speaker notes

- Not too shabby. I'm sure our component and the whole imaginary app could be improved a lot pretty easily by just following the specification of finite state machines, doing the work by hand. FSM's have a lot more features we didn't implement yet, so it might be worth your while to take a look at open source implementations of state machines rather than do everything by yourself. Especially when you want to create statecharts they can be a lot of work.
- I've been using a library called XState quite a bit in recent years and I mostly like it. Its pretty popular and widely accepted as the way to do state machines and statecharts in JavaScript ecosystem.
- It allows you to create state machines using a configuration object, which I will show you soon, and also execute the machines.
- Lets refactor our search form to use Xstate instead of our own FSM implementation.


Search Form
initial
SEARCH
loading
ERROR
failure
SUCCESS [hasResults]
showResults
SUCCESS [hasNoResults]
noResults

Speaker notes

- The visual representatios of the state machine could look something like this.
- From the intial state we transition to loading via search event, send by clicking the search button.
- Depending on the result of the fetching, which is not part of the machine, we transiotion to one of the three final states, indicated by the little squares in front of the state name.

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

const searchFormMachine = createMachine(
  {
    id: "Search Form",
    initial: "initial",
    states: {
      initial: {
        on: {
          SEARCH: "loading",
        },
      },
      loading: {
        on: {
          SUCCESS: [
            {
              cond: "hasResults",
              target: "showResults",
            },
            {
              cond: "hasNoResults",
              target: "noResults",
            },
          ],
          ERROR: "failure",
        },
      },
      failure: {
        type: "final",
      },
      noResults: {
        type: "final",
      },
      showResults: {
        type: "final",
      },
    },
  },
  {
    guards: {
      hasResults: (context, event) => event.results.length > 0,
      hasNoResults: (context, event) => event.results.length === 0,
    },
  }
);

```

Speaker notes

- The XState configuration would look something like this.
- First we import the createMachine function from the library, then pass our configuration object to it.
- The config has id of the machine, the intial state, and definitions of the states.
- Each state definition contains all the possible transitions from the state to some another state. From the initial state we transition to the loading state when SEARCH event occurs.
- The loading state transitions to failure on ERROR event and to show results or no results depending on the fetch result.
- The conditions or the guards as they are called in XState are defined in the bottom of the config.
- Finally, the final states are marked as final.
- So, the whole state machine is just 47 rows of configuration object code. Not too bad.

```jsx
import { useMachine } from "@xstate/react";
import { searchFormMachine } from "...";

const SearchForm = () => {
  const [state, send] = useMachine(searchFormMachine);
  const [keyword, setKeyword] = useState("");
  const [results, setResults] = useState([]);

  const search = () => {
    send("SEARCH");

    fetchResults(keyword)
      .then((results) => {
        setResults(results);
        send({ type: "SUCCESS", data: results });
      })
      .catch((error) => send("ERROR"));
  };

  return (
    <div>
      <input onChange={(event) => setKeyword(event.target.value)} />
      <button onClick={search} disabled={state.value === "loading"}>
        Search
      </button>

      {state.value === "initial" && <p>Hi! Enter keyword please.</p>}
      {state.value === "loading" && <p>Searching...</p>}
      {state.value === "failure" && <p>Oh no!</p>}
      {state.value === "noResults" && <p>Sorry! No results.</p>}
      {state.value === "showResults" &&
        results.map((result) => <p>{result}</p>)}
    </div>
  );
};

```

Speaker notes

- In the React component, we'll need to import useMachine from xstate's React utilities and our machine code.
- Then we just get the current state and a send event function from the useMachine hook. And voila, we have the state available in our component.
- We'll still use keyword and results like we did before, for now. Now we just send the SEARCH event to the machine, when we start the fetching. And send SUCCESS or ERROR events when we get the results. Notice, that we can send data to the machine as part of the event.
- In the return, we can get the current state name from state.value and render correct stuff.

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

const searchFormMachine = createMachine(
  {
    id: "Search Form",
    initial: "initial",
    context: {
      keyword: "",
      results: [],
    },
    states: {
      initial: {
        on: {
          SEARCH: "loading",
          SET_KEYWORD: {
            actions: assign({
              keyword: (context, event) => event.keyword,
            }),
          },
        },
      },
      loading: {
        on: {
          SUCCESS: [
            {
              cond: "hasResults",
              target: "showResults",
              actions: assign({
                results: (context, event) => event.results,
              }),
            },
            {
              cond: "hasNoResults",
              target: "noResults",
              actions: assign({
                results: (context, event) => [],
              }),
            },
          ],
          ERROR: "failure",
        },
      },
      failure: {
        type: "final",
      },
      noResults: {
        type: "final",
      },
      showResults: {
        type: "final",
      },
    },
  },
  {
    guards: {
      hasResults: (context, event) => event.results.length > 0,
      hasNoResults: (context, event) => event.results.length === 0,
    },
  }
);

```

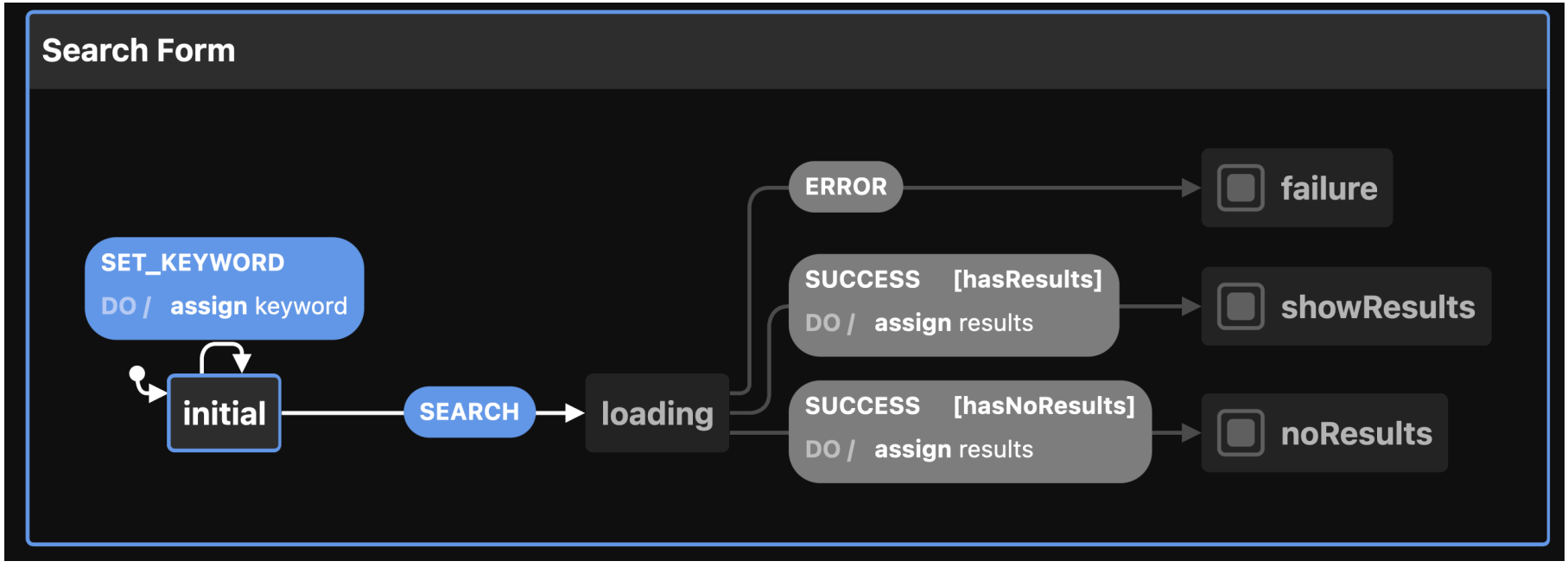
Speaker notes

- We can make it better.
- We define a context for the state machine, with empty initial values. The context is like a data store for the machine.
- Then we listen to SET_KEYWORD event on initial state. When it occurs, we assing its data to our machines context.
- And on SUCCESS event, we save the results to the context

```jsx
import { useMachine } from "@xstate/react";
import { searchFormMachine } from "...";

const SearchForm = () => {
  const [state, send] = useMachine(searchFormMachine);

  const search = () => {
    send("SEARCH");

    fetchResults(state.context.keyword)
      .then((results) => {
        send({ type: "SUCCESS", data: results });
      })
      .catch((error) => send("ERROR"));
  };

  return (
    <div>
      <input
        onChange={(event) =>
          send({ type: "SET_KEYWORD", data: event.target.value })
        }
      />
      <button onClick={search} disabled={state.value === "loading"}>
        Search
      </button>

      {state.value === "initial" && <p>Hi! Enter keyword please.</p>}
      {state.value === "loading" && <p>Searching...</p>}
      {state.value === "failure" && <p>Oh no!</p>}
      {state.value === "noResults" && <p>Sorry! No results.</p>}
      {state.value === "showResults" &&
        state.context.results.map((result) => <p>{result}</p>)}
    </div>
  );
};

```

Speaker notes

- On the component we may now remove the react's useState hooks for keyword and results and access the machines context using state.context
- We were already sending the results as part of the SUCCESS event, so no changes required here.
- On the inputs onChange handler we can straight up call the send function and send the SET_KEYWORD event with the inputs value as data.
- Cool, now the component's whole state is handled by XState. We could even move the fetching to be a part of the machine (maybe a reusable sub-machine?).


Search Form
SET_KEYWORD
DO / assign keyword
initial
SEARCH
loading
ERROR
failure
SUCCESS [hasResults]
DO / assign results
showResults
SUCCESS [hasNoResults]
DO / assign results
noResults

Speaker notes

- And by the way, our machine looks like this now. The SET_KEYWORD event assignes the data but doesn't trigger a transition while success assigns and transitions.

PROS AND CONS

- Add formalism to your code
- Makes your code more robust
- Separates (some of) your app's logic from the view layer
- Great way to make reusable logic
- Easy to represent visually
- XState can replace some other libraries or be used with others
- Requires you to think in a new mental model
- XState can be overwhelming

Speaker notes

- You are forced to think about and describe the behaviour of your app before implementing
- Makes sure there are no impossible states in your code
- Might be a good thing
- You can pretty much replace redux with xstate
- From boolean flags to states
- A state machine with hundreds of lines of config code can seem like you are just moving your spaghetti from a plate to another. You might be, I'm sure you can make bad state machines as well. I'm sure I've made quite a few.

ADD STATE MACHINES TO YOUR DEVELOPER TOOLBOX!

Next time you are adding a third boolean flag to your component, think to yourself, *"could this be a state machine?"*

Speaker notes

- If you want to take just one thing from this presentation, take this.
- You don't need to use any library at all, you don't event have to do a full fledged FSM. Just replace multiple booleans with one state variable with multiple, predefined values.

THANK YOU FOR YOUR TIME.

Any questions?

LINKS

- xstate.js.org
- xstate-catalogue.com
- stately.ai/registry

SOURCES

- statecharts.dev
- Wikipedia