Introduction to State Machines Using XState

Transcripts for Kyle Shevlin (https://egghead.io/instructors/kyle-shevlin) course on egghead.io (https://egghead.io/courses/introduction-to-state-machines-using-xstate).

Description

The difficulty of managing state is one of the primary reasons our applications become so complex. We try and manage this complexity with a lot of booleans, somewhat semantic variables like `isLoading`, `wasFetched`, and `hasError`, and over-engineered systems that are *still* full of bugs. Surely, there's a better way.

That better way is *state machines*.

State machines formalize how we define and transition through the states of our application and give us ultimate control of the most complex parts of our apps.

In this course, we will explore the problems state machines purport to solve, like boolean explosion. We'll try to solve it our own way first, get so far, and then demonstrate how state machines get us all the way. After that, we'll dive into the XState library, JavaScript's premiere state machine library, to learn its API and how to use it to solve our problems.

By the time you're done taking this course, you should have a solid education about state machines and be able to start applying them

Check out the Community Notes for this course (https://github.com/eggheadio-projects/introduction-to-state-machines-using-xstate-notes).

Course Intro and Overview

Kyle Shevlin: [0:01] Hey, everyone. Kyle Shevlin here with an introduction to my course on state machines and the XState library. I wanted to provide you with a brief overview of what we'll cover in the course. This course is roughly divided into three parts.

[0:14] In the first part, we'll explore the problems that state machines set out to solve. We'll take a function. We'll iterate on it until we get to a place where we understand where state machines come in and save the day.

[0:26] The second part is an introduction to the XState library and all its fundamental concepts, like machines, interpreters, states, events, actions, context, and guards. By this point, you should have a solid foundation for using the XState library on your own.

[0:42] In the final part, we explore more advanced state machines, such as hierarchical, parallel, and history states, as well as invoking services, like promises, callbacks, and even other machines.

[0:54] This is by no means an exhaustive introduction to the topic of state machines. My hope is that this introduction demonstrates their utility and encourages you to start using them in your work today. Finally, thanks for watching and enjoy the course.

Eliminate Boolean Explosion by Enumerating States

Instructor: [00:00] Here I have a function (`lightBulb`) that simulates the functionality of a light bulb. It tracks the state of the light bulb through the combination of two Booleans, `isLit` and `isBroken`. As this function is currently written, it is trivial to get into an impossible state. I will toggle the bulb from `unlit` to `lit`, call the `break` method, and we have a bulb that is both lit and broken.

index.js

```
function lightBulb() {
  let isLit = false
  let isBroken = false

  return {
    state() {
      return { isLit, isBroken }
    },
    toggle() {
      isLit = !isLit
    },
    break() {
      isBroken = true
    }
  }
}

const bulb = lightBulb()

const log = () => {
  console.log(bulb.state())
}

bulb.toggle()
bulb.break()
```

Terminal

```
node index.js
-> { isLit: true, isBroken: true }
```

[00:21] We can solve this problem imperatively by guarding against certain outcomes. We can add a guard and toggle that checks the `isBroken` state, guarantees `isLit` is set `false`, and returns early. We can also ensure that when the bulb is broken, we also update `isLit` to `false`. Now, running the same methods and logging them out produces a better result.

index.js

```
function lightBulb() {
  let isLit = false
  let isBroken = false

  return {
    ...
    toggle() {
      if(isBroken) {
        isLit = false
        return
      }
      isLit = !isLit
    },
    break() {
      isBroken = true
      isLit = false
    }
  }
}
```

Terminal

```
node index.js
-> { isLit: false, isBroken: true }
```

[00:45] There's a better way to solve this, though. To start, we're going to enumerate only the possible states of our light bulb. I'm going to create an enum using an object of only the possible states of the light bulb, `lit`, `unlit`, and `broken`. You could also use a map, if you prefer.

```
const STATES = {
  lit: 'lit',
  unlit: 'unlit',
  broken: 'broken'
}
```

[01:02] Next, we'll refactor our function by removing the Booleans `isLit` and `isBroken` and replacing that with a single value called state. We will set the default state to unlit and update the state method to return this value instead.

[01:17] Next, we'll refactor our `toggle` method. Instead of toggling a Boolean, we'll instead attempt to toggle between `lit` and `unlit` states. We can do this with a switch statement. If we're in the `lit` state, we'll set the state to `unlit` and vice versa. If we're in neither of those states, we know that we're in the `broken` state, and we don't have to do anything.

[01:36] Lastly, we can update our break method to set the state to `broken`. We now have a light bulb function that has enumerated only the possible states, methods that can only set the state to one of those possible ones, and returns a single possible state with each event performed on the light bulb.

```
function lightBulb() {
  const { lit, unlit, broken } =
LIGHT_BULB_STATES
  let state = unlit

  return {
    state() {
      return state
    },
    toggle() {
      switch (state) {
        case lit:
          state = unlit
          break

        case unlit:
          state = lit
          break
      }
    },
    break() {
      state = broken
    }
  }
}
```

Replace Enumerated States with a State Machine

Instructor: [0:00] We're going to replace this lightbulb functions that uses enumerated states with a state machine. We'll start by deleting the whole function (`lightBulb`). Next, instead of having the states enumerated in

an object, we'll create individual objects for each possible state. We can combine these objects into another object that we'll call states.

[0:16] After that, we'll need to define an initial state, so I'll define a variable called `initial`, and I'm going to set it to the string `"unlit"`. Now I can combine initial and states into an object we're going to call config. This will be the configuration object we'll pass the XState machine momentarily.

[0:32] Our config should also have an ID, and since this is a configuration for a lightbulb, I'll give it the string of `lightbulb`. Now we've enumerated our states, but we've yet to enumerate the events and transitions between states in our machine configuration.

```
const lit = {}
const unlit = {}
const broken = {}

const states = { lit, unlit, broken }

const initial = 'unlit'

const config = {
  id: 'lightBulb',
  initial,
  states
}
```

[0:45] With state machines, we trigger transitions through events, and we define which state nodes respond to which events. We'll do this by adding an on property to the state nodes that should respond to events. The `lit` and `unlit` states should both respond to a `break` event. Let's add those simultaneously.

[1:01] By convention, we capitalize the name of the event, and the event's value is the targeted state that we would like to transition to. Now, our lit and unlit states also respond to a toggle event, but they transition to different targets.

```
const lit = {
  on: {
    BREAK: 'broken',
    TOGGLE: 'unlit'
  }
}
const unlit = {
  on: {
    BREAK: 'broken',
    TOGGLE: 'lit'
  }
}
```

[1:16] Notice that the broken state responds to no events, that is because it's a final of our machine. We can put the final touch on our config by setting the `type` of `'final'` to our broken state. Yes, that pun was intended.

```
const broken = {
  type: 'final'
}
```

[1:29] We can now import the machine factory function from the XState library. Since I'm merely using Node for my example, I will require it with common JS instead of importing as an ES6 module. I'll create my

`lightBulbMachine` by passing our config to the machine factory function.

```
const { Machine } = require('xstate')
```

```
const lightBulbMachine = Machine(config)
```

[1:45] XState's machine function comes with a few useful getters and methods. Let's try some of them out on our `lightBulbMachine`. We can get the initial state node by using the initial state getter, logging this out we will see the entirety of the unlit state returned by the machine. It's quite a lot of information.

```
console.log(lightBulbMachine.initialState)
```

[2:01] Next, the most useful method on a machine is the transition method. Transition is a pure function, it receives a state and an event argument, and returns the next state object. That object was so big it didn't even fit in my full terminal window, so from here on out, I'll only log out the value returned to us by the state object. That's more manageable.

```
console.log(lightBulbMachine.transition('unlit',
 'TOGGLE').value)
```

[2:23] Let's try this out with some of the other states of our machine. If we set `unlit` to `lit`, it should toggle to unlit. If we set it to `broken`, it should stay broken. What happens if we pass it a state that the machine can't

handle? It throws an error, and it tells us the child state foo does not exist on lightbulb.

```
console.log(lightBulbMachine.transition('lit',
'TOGGLE').value)

// -> unlit
```

```
console.log(lightBulbMachine.transition('broken'
, 'TOGGLE').value)

// -> broken
```

[2:45] What about when we pass it an event that the machine doesn't handle? By default, the machine does nothing. It returns the state node that it started in, and doesn't take any transition.

```
console.log(lightBulbMachine.transition('lit',
'FOO').value)

// -> lit
```

[2:56] However, we can set our machine's config to `strict: true`, and then calling an event that our machine can't handle throws an error. We get the error "machine lightbulb does not accept event foo."

```
const config = {
 ...
 strict: true
}
```

Use an Interpreter to Instantiate a Machine

Instructor: [00:00] Using machine.transition to change state is useful, but it's tedious. We need a way to instantiate a machine that maintains its state and allows us to send events and more to it. To do this, we'll use the interpreter xState provides for us, `interpret`.

[00:14] I will require that function as well (`interpret`). The return value from an interpreted machine is known as a service, so I'll call this service. A service will maintain the state of our machine as it transitions from state to state, but it won't do anything until we start the service.

[00:29] Once we've started the service, we can send events. Sending an event returns us the next state, so we can actually save it and log out that object. We can see that it changed to the value `'lit'`. However, we don't need to save the next time every time we send.

index.js

```
const service =
interpret(lightBulbMachine).start()

const nextState = service.send('TOGGLE')
```

[00:44] We can also use the state getter on the service and see what the current state is. Again, the value is `'lit'`. It's the same state object, so that makes sense. However, this can't be the most useful way to log out the next state of our service. There's got to be a better way, and there is.

index.js

```
console.log(service.state)
```

[01:00] There are many methods that allow us to add listeners to our service and respond to changes to it. The most useful one to us is the onTransition method. onTransition takes listener function that receives a state argument.

[01:13] This is always sent the next state of the machine, and we can move our logging action to there (`onTransition` callback function). Now, each time that we send an event to our service, we'll see a value logged out for it in the console. We can also utilize some of the information from the state object itself here to limit our logging.

index.js

```
service.onTransition(state => {
  console.log(state.value)
})

service.send('TOGGLE')
service.send('TOGGLE')
service.send('BREAK')
```

Terminal

```
$ node index.js
lit
unlit
broken
```

[01:32] For instance, we can check and ensure that the state has actually changed before logging. I'm going to remove the type of final from our `broken` state to be able to send the break event twice. Notice that broken was only logged out once.

index.js

```
service.onTransition(state => {
  if (state.changed) {
    console.log(state.value)
  }
})
```

```
// Remove type of final
const broken = {}
```

```
service.send('TOGGLE')
service.send('TOGGLE')
service.send('BREAK')
service.send('BREAK')
```

Terminal

```
$ node index.js
lit
unlit
broken
```

[01:47] How about, instead of seeing that a state is changed, we check to see if a state matches a particular value and only log out when that happens? We can see that we only logged out, "Yo, I'm broke," when we were in the broken state.

index.js

```
service.onTransition(state => {
  if (state.matches('broken')) {
    // This will only run in a 'broken' state
  }
})
```

Use XState Viz to Visually Develop and Test Your Machine

Instructor: [00:00] (Tip: to get full use of this lesson, you are going to want to watch the video.) Xstate-viz is an online tool for visualizing our state machines. Here, I have already made the light ball machine. It allows you to write any state machine, the code panel on the right, and see it visualizes on the left.

[00:12] Not only is the visualization helpful, but it's interactive. We're able to test out our machine manually by clicking the various events on the left. I can click toggle from the unlit state to go to the lit state, toggle to go back, and break to go to broken.

[00:27] I can hit reset at any time to reset the machine. As we go further in the course, we'll discover other features of the visualization as well. There're two other tabs on the right panel that are useful to us.

[00:39] The first is the state tab. This tab gives us information regarding the current state of our machine. It provides us an object with a value. If we had context, actions, and more, it would say that information as well.

[00:52] The second tab is the event tab. We're able to call events here as well. Anything available to the current state notice made into a button here on the bottom right. Notice that there's break and toggle for the unlit and the visualization and break and toggle down here.

[01:08] What is useful about the event panel is that we can pass along extra information on our events, then we can with just a string represent a transition target.

[01:16] For example, I might want to send the break event, but I also might want to indicate what location, the light ball was in when it broke. We can send the event by clicking the send button and it'll get added to the history of events called.

[01:29] We can actually toggle this open and we can see that we call the type break and we added the information location leaving room. We're able to replay or edit any event that we want and we could see timestamps of when they were called.

[01:41] Lastly, if we're logged in like I am, we can save our machine definition as a gist. Notice that the URL has been appended with a gist query prem. The string is the ID of our gist. We could see it by going to

gist.githug.com/klyeshevlin in the ID.

Add Actions to Transitions to Fire Side Effects

Instructor: [0:00] We have a light bulb machine here, and I'd like to send an action when the light bulb breaks in the `unlit` state. Down here in `unlit`, I have a `break` event that leads us to `broken`. I'd like to fire off some kind of action, maybe a `console.log` that tells us that the build is broken.

```
const lightBulbMachine = Machine({
    id: 'lightBulb',
    initial: 'unlit',
    states: {
    lit: {
      on: {
        BREAK: 'broken',
        TOGGLE: 'unlit'
      }
    }
    unlit: {
      on: {
        BREAK: 'broken',
        TOGGLE: 'lit'
      }
    },
    broken: {}
  }
})
```

[0:16] To start with, I'm going to replace this string with an object. This string is really a shorthand for this object of target `broken`. We're targeting the next state. Next, we're going to add an actions property. The

value of actions can be a single function, or it can be an array of functions.

[0:35] Each function receives the context of the state machine and the event object that triggered the transition. In fact, from here, we can just log those out. We can update the machine, and we could see that the visualization changed, and it even shows the action that we'll take when we break.

```
const lightBulbMachine = Machine({
    id: 'lightBulb',
    initial: 'unlit',
    states: {
    ...
    unlit: {
      on: {
        BREAK: {
          target: 'broken',
          actions: [(context, event) => {
            console.log(context, event)
          }]
        },
        TOGGLE: 'lit'
      }
    },
    ...
  }
})
```

[0:52] If I were to open the console, and I hit the break, we could see the context was currently `undefined`, and the event that was taken was the type `'break'`. Closing my console and resetting the machine, there's an alternative way to define this action.

[1:09] Machine takes a second argument known as the options object. On this object, we can define our actions as methods on the actions object. We'll create a method -- in this case, log broken -- and we'll simply log out, "Yo, I am broke."

[1:27] Then, where we want that action to be taken, we can simply write a shorthand of a string. We can update our machine and see that the visualization is updated and tells us which action name we're now calling.

```
const lightBulbMachine = Machine({
    id: 'lightBulb',
    initial: 'unlit',
    states: {
    ...
    unlit: {
      on: {
        BREAK: {
          target: 'broken',
          actions: ['logBroken']
        },
        TOGGLE: 'lit'
      }
    },
    ...
  }
}, {
  actions: {
    logBroken: () => { console.log('yo I am
broke') }
  }
})
```

[1:43] If I open the console again, we'll see that it still works. What's nice about having the context and the event sent to an action is that we can

actually pass more information on our event to be used by an action.

[1:56] I'm going to add context and event back to this log `broken`, and on event, I'm going to expect some more information. Perhaps instead of just saying, "I'm broke," I can tell you what room I broke in. We'll update the machine, and then in the events panel, we can select break and add some information to it.

```
const lightBulbMachine = Machine({
   ...
}, {
  actions: {
    logBroken: (context, event) => {
console.log(`yo I am broke in the
${event.location}`) }
  }
})
```

[2:14] We can add a `location` of `office`. When we send this, if we open our console, it says, "Yo, I am broke in the office." Now, if I want to add this to the break event on `lit`, it's as simple as repeating the steps. I update the machine, and we see that the actions are now available on both.

```
const lightBulbMachine = Machine({
    id: 'lightBulb',
    initial: 'unlit',
    states: {
    lit: {
      on: {
        BREAK: {
          target: 'broken',
          actions: ['logBroken']
        },
        TOGGLE: 'unlit'
      }
    }
    ...
  }
}, {...})
```

[2:38] Perhaps I toggle to `lit`, and then I log `broken`. I am in the broken state, which logs out `undefined`, because I forgot to define it on the event object.

Trigger Actions When Entering and Exiting a XState State

Instructor: [00:00] Here, I have a light bulb machine that has actions being called on the break events in the `lit` and `unlit` states. This action tells us that the light bulb has broke, and if a location has been added to the event object, it'll tell us what location it is in.

```
const { Machine } = require('xstate')

const lightBulbMachine = Machine(
```

```
  {
    id: 'lightBulb',
    initial: 'unlit',
    states: {
      lit: {
        on: {
          BREAK: {
            target: 'broken',
            actions: ['logBroken']
          },
          TOGGLE: 'unlit'
        }
      },
      unlit: {
        on: {
          BREAK: {
            target: 'broken',
            actions: ['logBroken']
          },
          TOGGLE: 'lit'
        }
      },
      broken: {}
    }
  },
  {
    actions: {
      logBroken: (context, event) => {
        console.log(`yo I am broke in the
${event.location}`)
      }
    }
  }
)
```

[00:16] However, adding these on both break transitions is a little tedious and unnecessary. What if, instead of calling this action on a transition, we could call it when we entered a state as well? Well, we can by using the `entry` property.

[00:31] Entry is exactly like actions. It can take a single function that receives the context and an event object. It can also be an array of functions, and it can be an array of strings. In our case, we can move this log broken action that's been defined on the second argument to machine, the options object. We can add that as an entry action on the broken state.

[00:54] I can then undo this work on both breaks using the shorthand of broken and update my machine. We now see that, whenever the broken state is entered, the log broken action will be taken. We can do this by going to the events panel.

```
const { Machine } = require('xstate')

const lightBulbMachine = Machine(
  {
    id: 'lightBulb',
    initial: 'unlit',
    states: {
      lit: {
        on: {
          BREAK: 'broken',
          TOGGLE: 'unlit'
        }
      },
      unlit: {
        on: {
          BREAK: 'broken',
          TOGGLE: 'lit'
        }
      },
      broken: {
        entry: ['logBroken']
      }
    }
  },
  {
    actions: {
      logBroken: () => {
        console.log('yo i am broke')
      }
    }
  }
)
```

[01:11] We can select `break`, and we can add the location in. We send that to our machine. We see we've entered the broken state, and if I open my console, we'll see, `"Yo, I am broke in the office."` Now, it makes sense that, if we can call actions when we enter a state, we should also be able to call actions when we leave a state.

[01:31] Those are called exit actions and are added on the exit property. Perhaps when I exit the `lit` state, I'll say something about it growing dark and cold, a morbid little light bulb, wouldn't you say? We've updated our machine, and we now see that we have an exit action on the lit state.

```
const { Machine } = require('xstate')

const lightBulbMachine = Machine(
  {
    id: 'lightBulb',
    initial: 'unlit',
    states: {
      lit: {
        exit: () => {
          console.log('it is so dark and cold')
        }
        on: {
          BREAK: 'broken',
          TOGGLE: 'unlit',
        },
      },
      ...
    },
  },
  { ... })
```

[01:49] If I toggle to `lit`, and then I open up my console, we can see that when we exit the `lit` state, `"It is so dark and so cold."` It doesn't matter whether we exit it to go to `unlit` or that we go to `break`. What happens when we call exit transition actions and entry actions all in sequence? What order do they go in?

[02:11] Let's add an action on the break event from lit and see what order they all fire in. Now, we'll toggle into our lit state, and we can see that we have exit actions, we have transition actions, and we have entry actions on broken.

```
const { Machine } = require('xstate')

const lightBulbMachine = Machine(
  {
    id: 'lightBulb',
    initial: 'unlit',
    states: {
      lit: {
        on: {
          BREAK: {
            target: 'broken',
            actions: () => {
              console.log('transitioning to
broken')
            }
          },
          TOGGLE: 'unlit',
        },
      },
      ...
    },
  },
  { ... }
)
```

[02:26] When we call the break event, we see that the exit action of our current state, `lit`, was called before the transition actions of `break`, which was called before the entry actions of `broken`. This is always the order of actions fired.

Replace Inline Functions with String Shorthands

Instructor: [00:00] Rather than defining our actions as inline functions, or even extracting them out, we can handle this by using a string shorthand and the options object on the machine factory function.

[00:13] We'll start by adding an object as the second argument to machine. It's hard to see on this screen, but if we go all the way up, this is the first object that ends here, and this is the second object. This is the options object. On this object, we can define things like `actions`, `guards`, `services`, `activities`, and `delays`. We'll focus on actions in this lesson.

```
const lightBulbMachine = Machine({
  ...
  // This is the first object
}, {
  // This is the actions object
})
```

[00:35] We add an `actions` property. That `actions` property becomes an object. We can define our functions here on this object. Rather than having `logLocation` and `buyANewBulb` up extracted as functions, we can do `logLocation`. Then we can take the function that we had up here. I want to copy that out, get rid of that, and paste that in here.

[01:01] I'll do the same for `buyANewBulb`. I will copy this, cut it out, remove that, and paste it in place here. Now, I can come here (`entry` property in `broken`) and I can turn these into strings. XState knows that I'm looking for actions with this name on this actions object here.

```
const lightBulbMachine = Machine({
  ...
   broken: {
      entry: ['logBroken', 'buyANewBulb']
    },
}, {
  actions: {
    logLocation: (context, event) => {
      console.log(event.location)
    },
    buyANewBulb: () => {
      console.log('buy a new bulb')
    },
  }
})
```

[01:23] I can save this. If I send a break event to the machine and pass along the information of the location, I can see that logged out in the terminal.

Use Internal Transitions in XState to Avoid State Exit and Re-Entry

Instructor: [00:00] Here I have a contrived example of a machine, an idle machine (`idleMachine`), that only has one state, `idle`. On this machine, we only have one event, `DO_NOTHING`. It transitions back to the idle state. Let's give it a couple tries.

```
const idleMachine = Machine(
  {
    id: 'idle',
    initial: 'idle',
    states: {
      idle: {
        entry: ['logEntry'],
        exit: ['logExit']
      }
    },
    on: {
      DO_NOTHING: 'idle'
    }
  },
  {
    actions: {
      logEntry: () => {
        console.log('entered')
      },
      logExit: () => {
        console.log('exited')
      }
    }
  }
)
```

[00:15] We can see that each time we call `DO_NOTHING`, we actually exit the idle state. We leave it, calling the `logExit` action, which is down here in logs exited, and then we reenter the idle state, calling the entry action `logEntry` that logs entered. Hence why, exited-entered, exited-entered, exited-entered.

[00:38] Now, on occasion, you might want to transition back to the same state without ever leaving the state node, and thus, never reentering the state node. How do we do that in XState?

[00:50] Well, we can do it quite simply by adding one character. We add this period, this dot notation, and what we've told the machine is that we would like to make an internal transition.

```
const idleMachine = Machine(
  {
    ...
    on: {
      DO_NOTHING: '.idle',
    },
  },
  {
    ...
  }
)
```

[01:01] We want to make a transition that stays within the state it's in. We don't want to exit it and we don't want to enter it, so we're going to update our machine.

[01:09] Notice that the arrow slightly changed. If we call `DO_NOTHING` now, we'll see that nothing gets logged out to the terminal. I hope you can hear me clicking.

[01:19] This simple change creates a transition where we never leave this node.

Send Events to the Machine with the XState Send Action Creator

Instructor: [0:00] Here I have an echo machine (`echoMachine`) with one state, `listening`, and two events, `SPEAK` and `ECHO`. What I'd like this machine to do is any time I speak, I want the `ECHO` event to be triggered. How can I do this? I can use a special action called the `send` action creator to send events to my machine.

[0:20] I'm going to add the `actions` property here (inside of the `SPEAK` event object). Then I'm going to use the `send` function. The `send` function receives an event. In this case, it'll be `ECHO`. We update our machine. We now see that `SPEAK` is an available event to send. When we send the `SPEAK` event, it'll send an echo event on the next tick of the machine.

```
const echoMachine = Machine({
  id: 'echo',
  initial: 'listening',
  states: {
    listening: {
      on: {
        SPEAK: {
          actions: send('ECHO')
        },
        ECHO: {
          actions: () => {
            console.log('echo, echo')
          }
        }
      }
    }
  }
})
```

[0:40] I'm going to open up the console. Reset the machine. We'll call `speak`. Every time we do, it echoes out. We can also update this from being just a string to an object with a type of event that we want to send. This too will also send echoes.

```
const echoMachine = Machine({
  ...
  states: {
    listening: {
      on: {
        SPEAK: {
          actions: send({
            type: 'ECHO'
          })
        },
        ...
      },
    },
  },
})
```

Track Infinite States with with XState Context

Instructor: [0:00] Not all states can be represented as a set of finite states. Some things are infinite states. In XState, this is considered context or called extended state. I have a lightbulb here that I'm calling a `multicoloredBulb` because I'd like it to act like a modern lightbulb that can change colors.

[0:18] We can do this with context. We add context as an object on our machine configuration. In this case, I'll give it a property of `color` which I'll set initially to white (`#fff`). We're going to update our machine, and nothing has changed about our visualization.

```
const multiColoredBulbMachine = Machine({
  id: 'multiColoredBulb',
  initial: 'unlit',
  context: {
    color: '#fff'
  },
  ...
})
```

[0:35] When I select state tab, we currently have a context of `color`. We can make updates to our color by creating actions called assign. I'm going to add an event on the lit state called `CHANGE_COLOR`. `CHANGE_COLOR` will not change the state of the lightbulb, so we don't need to give this a target.

[0:53] We'll add the `actions` property. We can use our `assign` function here. `Assign` has two signatures. It can receive an object of key value pairs. In this case, we could set `color` straight to red (`#f00`) for example. We can update the machine. We see that we now have a new event.

```
const multiColoredBulbMachine = Machine({
  id: 'multiColoredBulb',
  initial: 'unlit',
  context: {
    color: '#fff'
  },
  states: {
    lit: {
      on: {
        BREAK: 'broken',
        TOGGLE: 'unlit',
        CHANGE_COLOR: {
          actions: assign({
            color: '#f00'
          })
        }
      },
    },
    ...
  },
})
```

[1:13] We even see what color that this assign will set it to. It says that it sets to red (`#f00`). We'll toggle the lit. We'll call `CHANGE_COLOR`. We'll go to our state tab and we'll see that it's currently red (`#f00`). We'll go back to definition and reset the machine.

[1:28] We can also use this key value pair and provide a function to the value that receives the current context, the event object, in this case `CHANGE_COLOR`. We can assign it based on values from there. In this case, I might want to send a color on the event itself and assign that instead.

[1:50] From the advanced tab, I can call `TOGGLE` to get to the lit state. We have `CHANGE_COLOR` available. We'll select `CHANGE_COLOR`. We can add a color of, let's make it green (`#0f0`). We've called the `CHANGE_COLOR` event. We could see that the color was updated to green (`#0f0`). In fact, if we hover over this, we could see the exact function we assigned it to, context event returns event color.

[2:14] Assign can also take a function as its argument. In this case, the function receives the current context, the event, and returns an object to be merged in with the next context. We can update our machine, toggle to lit, and we can call `CHANGE_COLOR` event again with a color set. This case, let's make it blue (`#00f`). Once again, our state has updated correctly.

```
const multiColoredBulbMachine = Machine({
  ...
  states: {
    lit: {
      on: {
        BREAK: 'broken',
        TOGGLE: 'unlit',
        CHANGE_COLOR: {
          actions: assign((context, event) => ({
            color: event.color
          }))
        }
      },
    },
    ...
  },
})
```

[2:38] Notice that when I hover over assign with this format, I can't see the function that I was called. It is generally preferred to call assign with the object signature. There's one last way that we can add this action. Rather than defining it inline here, we can define it on the second argument to machine, the options object.

[2:58] We'll define an `actions` key here (inside of the actions object) and create a method that will be this action. In this case, we'll call it `changeColor`. We could take this function from up here (inside of CHANGE_COLOR actions object), cut it out, call the `changeColor` method from here (inside of CHANGE_COLOR actions object), and paste this back in here (`changeColor` action). We can update our machine and see that `changeColor` has now replaced our assign, as it now has a name.

```
const multiColoredBulbMachine = Machine({
 ...
  states: {
    lit: {
      on: {
        BREAK: 'broken',
        TOGGLE: 'unlit',
        CHANGE_COLOR: {
          actions: ['changeColor']
        }
      },
    },
    ...
  },
}, {
  actions: {
    changeColor: assign((context, event) => ({
            color: event.color
          }))
  }
})
```

[3:22] We'll toggle again to `lit`, go to the events tab, select `CHANGE_COLOR`. This time, we'll make it black. I don't know how a black (`#000`) light would work this way, but maybe it would. We send the event. We could see that the state updated to black (`#000`).

How Action Order Affects Assigns to Context in a XState Machine

Instructor: [00:00] Here I have a contrived example `doubleCounterMachine` to demonstrate how action order affects assigns in context. This machine only has one state idle, and a response

to one event, `INC_COUNT_TWICE` (increment count). This event fires off four actions. One `console.log` before to show the count, two calls of `incCount`, and one `console.log` after to show the count.

[00:25] We scroll down here we can see what the `incCount` function is, it's an assigned function to `context`, that takes our `context` and gets the current count and adds 1. Let's call this and see what happens. Opening up the console, we see that we actually got `before 2` and `after 2`. That's odd, why did that happen?

```
const doubleCounterMachine = Machine(
  {
    id: 'doubleCounter',
    initial: 'idle',
    context: {
      count: 0
    },
    states: {
      idle: {
        on: {
          INC_COUNT_TWICE: {
            actions: [
              context => {
                console.log(`Before:
${context.count}`)
              },
              'incCount',
              'incCount',
              context => {
                console.log(`After:
${context.count}`)
              }
            ]
          }
        }
```

```
      }
    }
  },
  {
    actions: {
      incCount: assign({ count: context =>
context.count + 1 })
    }
  }
)

// Before: 2
// After: 2
```

[00:44] To understand this we actually have to think about the machine transition method. I'm going to write some pseudocode up here to help it make sense. Machine.transition is a pure function. It's a function of the state and the event. In order for this to be pure function, we have to get back the same object as our next state every time we pass in this particular state and this particular event.

[01:10] The way it does this, is it returns the next context completely, but taking all the actions, the state's exit actions, the transition actions, and the next state's entry actions, and filtering out any assigns that might happen in them, and merging them into the next context object. It looks like this. Since all the assigns are batched together to give us that next context object, all we're left with are any actions that aren't assigns, so it's almost as if these actions are ordered in this way.

```
Machine.transition(state, event) {
  context: nextContext,
  actions: [
    ...state.exit,
    ...actions,
    ...nextState.entry
  ].filter(action => {
    if(assignAction) {
      mergeIntoNextContext()
      return false
    }

    return true
  })
}
```

[01:39] All the assigns first, and then any of the other actions in the order that they were declared. Knowing this, we can remove the pseudocode and we can make a change to how our `doubleCounterMachine` works to actually work as we expect. What we realize is we actually have two kinds of contexts, we not only have a `count`, we have a `previousCount`.

[01:59] Since we have a `previousCount`, it makes sense that we make an action that assigns this value during this. We have `incCount`, we can also have `setPreviousCount`. We can take this, and add this to our actions. Now that I've added that, I can update this to `previousCount`, save the machine, and call `INC_COUNT_TWICE`. I should be able to open the console, and we now see a before of `0` and an after of `2`.

```
const doubleCounterMachine = Machine(
  {
```

```
    id: 'doubleCounter',
    initial: 'idle',
    context: {
      count: 0,
      previousCount: undefined
    },
    states: {
      idle: {
        on: {
          INC_COUNT_TWICE: {
            actions: [
              context => {
                console.log(`Before:
${context.previousCount}`)
              },
              'setPreviousCount',
              'incCount',
              'incCount',
              context => {
                console.log(`After:
${context.count}`)
              }
            ]
          }
        }
      }
    }
  },
  {
    actions: {
      incCount: assign({ count: context =>
context.count + 1 }),
      setPreviousCount: assign({
        previousCount: context => context.count
      })
    }
```

```
  }
)

// Before: 0
// After: 2
```

Use Activities in XState to Run Ongoing Side Effects

Instructor: [00:00] Here, I have a rudimentary alarm clock machine (`alarmClockMachine`). It has two states, `idle` and `alarming`. When we alarm, we go to the `alarming` state, and when we stop, we go back to `idle`. Now, what good is an alarm clock that doesn't beep to wake us up?

```
const alarmClockMachine = Machine({
  id: 'alarmClock',
  initial: 'idle',
  states: {
    idle: {
      on: { ALARM: 'alarming' }
    },
    alarming: {
      on: { STOP: 'idle' }
    }
  }
})
```

[00:14] Right now, `alarming` doesn't do anything. There's no actions. An action doesn't really fit what we want. What we really want is an action that's ongoing for the entire time we're in the `alarming` state. That's where activities come into play.

[00:29] We create activities by adding an activities property, and this can be a single function, or it can be an array of functions, each function receiving the current context and the event that caused the transition. We can also write them with string shorthand.

[00:46] Rather than declaring them here in-line, we can write a string that'll be the name of a method we'll create that is our activity. In this case, I want my alarm clock to beep at me, so I'll create a `beeping` activity.

```
const alarmClockMachine = Machine({
  id: 'alarmClock',
  initial: 'idle',
  states: {
    idle: {
      on: { ALARM: 'alarming' },
    },
    alarming: {
      /*
      Single function:
      activities:  (context, event) => {}

      Array of functions
      activities:  [(context, event) => {}]

      */
      activities:  ['beeping']
      on: { STOP: 'idle' },
    },
  },
})
```

[00:59] Now, down in the second argument of machine, the `options` object, we'll add `activities`. We'll create a `beeping` method to correspond with the beeping string we placed in `activities`. As I said before, this receives the `context` and the `event`. Though to be honest, I don't really need either of those for what I'm going to do.

[01:20] An activity is an ongoing side effect that takes a nonzero amount of time. In our case, we want to beep in the console while we're alarming. To do this, I'll create a `beep` function inside of this held enclosure. This `beep` function will simply log out "beep."

[01:39] Next, I want to call this `beep` the very moment that we start the activity. In order to do that, I can just call the function here (inside of the `beeping` activity). I also want it to repeat every second, so I'm going to `setInterval`. Those of you who are astute might notice, though, that I have no way right now cleaning up this interval.

```
const alarmClockMachine = Machine({
  ...
}, {
  activities: {
    beeping: (context, event) => {
      const beep = () => {
        console.log('beep')
      }

      beep()
      setInterval(beep, 1000)
    }
  }
})
```

[01:59] If I leave it like this and update the machine, I open up the console, and I start the alarm, it's going to beep every second like I expected. When I hit stop, it's going to keep on beeping. We've created a memory leak.

[02:14] We've done this, because we have failed to clean up after ourselves with the interval. I'm going to copy all this and reset my machine. The way we handle this memory leak is that activities can return a function that'll perform any cleanup that we need to do on anything we set up inside of the activity.

[02:35] In this case, `setInterval` will return to us an interval ID (`intervalID`) that we can save, and we can return a function that'll clear that interval. Now, if I update the machine, I can open up the console, and we can trigger our alarm again.

```
const alarmClockMachine = Machine({
  ...
}, {
  activities: {
    beeping: (context, event) => {
      const beep = () => {
        console.log('beep')
      }

      beep()
      const intervalID = setInterval(beep, 1000)

      return () => clearInterval(intervalID)
    }
  }
})
```

[02:49] We'll see that it beeps, and it continues to beep every second. Now, when I stop, the beeping also stops.

Conditionally Transition to States with Guards in XState

Instructor: [00:00] Here I have a state machine for a vending machine (`vendingMachine`). It has two states, `idle` and `vending`. On `idle`, I can select an item which currently targets vending as the transition.

```
const vendingMachineMachine = Machine(
  {
    id: 'vendingMachine',
    initial: 'idle',
    context: {
      deposited: 0
    },
    states: {
      idle: {
        on: {
          SELECT_ITEM: 'vending',
          DEPOSIT_QUARTER: {
            actions: ['addQuarter']
          }
        }
      },
      vending: {}
    }
  },
  {
    actions: {
      addQuarter: assign({
        deposited: context => context.deposited
+ 25
      })
    }
  }
)
```

[00:10] However, that's not how we want our vending machine to work. We don't want them to just be able to select an item without having paid for it. We can do this by adding a guard.

[00:20] To start, we're going to change this (`SELECT_ITEM`) from being a string to an object of `target: 'vending'`, since it's still the state that we want to target. Next, we're going to add the `cond` property. The `cond` property is a predicate function. That's a function that returns a Boolean.

[00:36] In this case, we want it to return `true` when we'd like to take the transition and return `false` when we don't want it to. Guard functions receive the context and event as arguments. However, in this case, we only need the context.

[00:49] We want it to return `true` when `context.deposited` is greater than or equal to 100. We're going to update our machine. We'll see that a `cond` has been set on select item. In fact, it's disabled. We can't select it right now.

```
const vendingMachineMachine = Machine(
  {
    ...
    states: {
      idle: {
        on: {
          SELECT_ITEM: {
            target: 'vending',
            cond: context => context.deposited
>= 100
          },
          DEPOSIT_QUARTER: {
            actions: ['addQuarter'],
          },
        },
      },
      vending: {},
    },
  },
  {
    ...
  }
)
```

[01:03] Now, another way to set the condition is rather than setting the function here in place, we can use the second argument to machine, the options object and add guards to that.

[01:13] I am going to add a guard down here (inside the options object). We'll create a new conditional function. We'll call it `depositedEnough`. We'll take the function that we wrote up here (`SELECT_ITEM.cond`).

We're going to cut that out and replace it with a string of the same method name, so `'depositedEnough'`, and take that function and place it here (in `guards` object).

```
const vendingMachineMachine = Machine(
  {
    ...
    states: {
      idle: {
        on: {
          SELECT_ITEM: {
            target: 'vending',
            cond: 'depositedEnough'
          },
          ...
        }
      },
      vending: {}
    }
  },
  {
    actions: {
      addQuarter: assign({
        deposited: context => context.deposited
+ 25
      })
    },
    guards: {
      depositedEnough: context =>
context.deposited >= 100
    }
  }
)
```

[01:36] We're now going to update our machine. We'll see that the visualization has updated to show us that the `depositedEnough` condition is here. It's also red because it hasn't been met yet.

[01:46] Let's open up the state tab. We'll be able to watch the context increase as I deposit quarters. One, two, three, four. `depositedEnough` is now green. That condition is `true` and I can take this transition by firing this event.

[02:01] I select my item and it gets vended.

Simplify State Explosion in XState through Hierarchical States

Instructor: [00:00] Here I have a state machine of a door (`door`). It's in the initial state of `locked`. I've written out the states `locked`, `unlocked`, `closed`, and `open`, but I haven't given them any events or transitions yet. That's because when we think about a door, it's actually a bit of challenging problem.

```
const door = Machine({
  id: 'door',
  initial: 'locked',
  states: {
    locked: {},
    unlocked: {},
    closed: {},
    opened: {}
  }
})
```

[00:16] When it's `locked`, it's also closed. When it's `unlocked`, it could still be closed, but it could also be `opened`. When it's `opened`, it should never be able to be `locked`. We end up with a pretty confusing graph of states if we try really hard to make this work with all the states on the same level.

[00:33] Fortunately for us, we don't have to keep them at the same level and we can use hierarchical states. We can see that `closed` and `open` really are states that fall under when the door is `unlocked`, and it might make more sense for us to put closed and open as states under `unlocked`. This is our first step in making hierarchical states.

[00:53] The next thing is we actually need to define what's the initial state of this subset of states (`closed`). We can update our machine, and it now reflects that we have a `locked` state and an `unlocked` state. Inside the `unlocked` state, we have child states `closed` and `open`.

```
const door = Machine({
  id: 'door',
  initial: 'locked',
  states: {
    locked: {},
    unlocked: {
      initial: 'closed'
      states: {
        closed: {},
        opened: {}
      }
    }
  }
})
```

[01:09] From here, we can start to fill out our events and they'll make sense. When we're closed, we can open it. When we're opened, we can close it. We'll update that. When we're locked, we can get to unlocked. How do we go from our unlocked state back to our locked state?

```
const door = Machine({
  id: 'door',
  initial: 'locked',
  states: {
    locked: {
      on: {
        UNLOCK: 'unlocked'
      }
    },
    unlocked: {
      initial: 'closed'
      states: {
        closed: {
          on: {
            OPEN: 'opened'
          }
        },
        opened: {
          on: {
            CLOSE: 'closed'
          }
        }
      }
    }
  }
})
```

[01:27] We don't want to be able to lock the door when it's open, so we don't want to put that event here. We can put an event when it's closed to

lock it. However, this is going to throw an error when we update. It's going to say invalid transition for door unlocked closed.

[01:42] What it's saying is that `locked` doesn't exist as a state in `unlocked`. It's one level up. So how do we do this? We could start by using the ID on the machine itself, then doing dot notation to work our way down.

[01:57] In this case, we can set the transition target to `#door.locked`, and it'll go from door down to the `locked` state. We can see we have a lock action. Our door actually works. We've unlocked it, we can open it. We can't lock it from the open state. Closed, locked, and we're back to locked.

```
const door = Machine({
  id: 'door',
  initial: 'locked',
  states: {
    ...
    unlocked: {
      initial: 'closed'
      states: {
        closed: {
          on: {
            LOCK: '#door.locked'
            OPEN: 'opened'
          }
        },
        ...
      }
    }
  }
})
```

[02:15] The other way we can handle this, rather than using the identifier of the door, we can give the `locked` state an ID (`'locked'`), and then we can change this just to `'#locked'`. We update it, and our visualization stays the same. We can unlock the door, open, close, lock it.

```
const door = Machine({
  id: 'door',
  initial: 'locked',
  states: {
    locked: {
      id: 'locked',
      on: {
        UNLOCK: 'unlocked'
      }
    },
    unlocked: {
      initial: 'closed'
      states: {
        closed: {
          on: {
            LOCK: '#locked'
            OPEN: 'opened'
          }
        },
        ...
      }
    }
  }
})
```

Multiple Simultaneous States with Parallel States

Instructor: [00:00] It's approaching winter here in Portland while I'm recording this. I'm inspired by my space heater, so I made a space heater machine (`spaceHeater`).

[00:05] It has two top-level states -- `poweredOff` and `poweredOn` -- that are transitioned to with the `TOGGLE_POWER` event. Inside of `poweredOn`, we have a hierarchical state where we also have `lowHeat` and `highHeat` that is toggled with the `TOGGLE_HEAT`.

```
const spaceHeaterMachine = Machine({
  id: 'spaceHeater',
  initial: 'poweredOff',
  states: {
    poweredOff: {
      on: { TOGGLE_POWER: 'poweredOn' }
    },
    poweredOn: {
      on: { TOGGLE_POWER: 'poweredOff' },
      initial: 'lowHeat',
      states: {
        lowHeat: {
          on: { TOGGLE_HEAT: 'lowHeat' }
        },
        highHeat: {
          on: { TOGGLE_HEAT: 'highHeat' }
        }
      }
    }
  }
})
```

[00:19] Now, my space heater has another useful feature that we should add and that's oscillation, the ability for it to go back and forth. Oscillation definitely falls under powered on, but it's not affected by `lowHeat` and `highHeat`, so adding it here doesn't seem to make sense.

[00:36] No, oscillating is really the state of the space heater that happens in parallel to the heating of the space heater. In XState, we can create parallel states. I want to comment this out for right now, so that we're not distracted by it.

[00:51] To create parallel states, we first do not provide an initial, since it's in each state all at the same time. Instead, we define a `type` of `parallel`.

[01:02] From here, we then enumerate the states that we're in all at the same time. In this case, we'll be in a `heated` state and we'll be in an `oscillation` state.

```
const spaceHeaterMachine = Machine({
  id: 'spaceHeater',
  initial: 'poweredOff',
  states: {
    poweredOff: {
      on: { TOGGLE_POWER: 'poweredOn' },
    },
    poweredOn: {
      on: { TOGGLE_POWER: 'poweredOff' },
      type: 'parallel',
      states: {
        heated: {},
        oscillation: {}
      }
      ...

    },
  },
})
```

[01:12] Let's update the machine as is. We can now see that when we're powered on, we have two top-level states. If we toggle in, we're inside of `heated` and `oscillation` both at the same time.

[01:25] We can now take the values we had here for our `heated`. We can copy them and move them inside of the `heated` state. We'll update the machine. We'll see that `heated` now has its own state that we can toggle between, and it doesn't change anything about `oscillation`, partly because we haven't written it yet, so let's add that.

```
const spaceHeaterMachine = Machine({
  id: 'spaceHeater',
  initial: 'poweredOff',
  states: {
    poweredOff: {
      on: { TOGGLE_POWER: 'poweredOn' }
    },
    poweredOn: {
      on: { TOGGLE_POWER: 'poweredOff' },
      type: 'parallel',
      states: {
        heated: {
          initial: 'lowHeat',
          states: {
            lowHeat: {
              on: { TOGGLE_HEAT: 'lowHeat' }
            },
            highHeat: {
              on: { TOGGLE_HEAT: 'highHeat' }
            }
          }
        },
        oscillation: {}
      }
    }
  }
})
```

[01:46] I'm going to fix my indentation really quick and give myself some room here to see my editor. Inside of `oscillation`, we'll have an initial of `disabled` and states of `enabled` and `disabled`.

```
const spaceHeaterMachine = Machine({
  id: 'spaceHeater',
  initial: 'poweredOff',
  states: {
    poweredOff: {
      on: { TOGGLE_POWER: 'poweredOn' }
    },
    poweredOn: {
      on: { TOGGLE_POWER: 'poweredOff' },
      type: 'parallel',
      states: {
        ...
        oscillation: {
          initial: 'disabled',
          states: {
            enabled: {
              on: { TOGGLE_OSC: 'disabled'}
            },
            disaabled: {
              on: { TOGGLE_OSC: 'enabled'}
            }
          }
        }
      }
    }
  }
})
```

[02:03] We'll update our machine. We can now see that when we're powered on, we're both in the initial state of `lowHeat` for heated and `disabled` for oscillation.

[02:13] I can toggle these and they have no effect on the other state. When I click `TOGGLE_POWER`, I leave all those states and go back to powered off.

Recall Previous States with XState History States Nodes

Instructor: [00:00] Inspired by the cooler weather, I have a space heater machine. It has two top-level states, `poweredOff` and `poweredOn`. When we're in the `poweredOn` state, we have two levels of heat -- we have `low` and we have `high`.

[00:11] It sure would be nice if my space heater would remember what setting I was on the next time that I power it back on. We can do this by using history state nodes. We add a history state node by defining the type as history.

```
const spaceHeaterMachine = Machine({
  id: 'spaceHeater',
  initial: 'poweredOff',
  states: {
    poweredOff: {
      on: { TOGGLE_POWER: 'poweredOn' }
    },
    poweredOn: {
      on: { TOGGLE_POWER: 'poweredOff' },
      initial: 'low',
      states: {
        low: {
          on: { TOGGLE_HEAT: 'high' }
        },
        high: {
          on: { TOGGLE_HEAT: 'low' }
        },
        hist: {
          type: 'history'
        }
      }
    }
  }
})
```

This creates a state node that when transitioned to will return to the previous state of this area of the machine.

[00:36] In this case, that means the previous state underneath the `poweredOn` state. In order to trigger this transition, we'll need to actually set `TOGGLE_POWER` here not to `poweredOn`, but to the specific history state node by using dot notation.

```
poweredOff: {
  on: { TOGGLE_POWER: 'poweredOn.hist' }
},
```

[00:52] We can update our machine, and we now see that `TOGGLE_POWER` actually goes directly to this history state node. By default, if history doesn't have a previous state to go to, it'll go to the initial state of that area.

[01:05] When we click `TOGGLE_POWER`, it goes to the low state. If I switch it to high, and then I toggle it off, when I click `TOGGLE_POWER` again, we'll go to that history node, and it remembers that I was in the high state.

[01:20] What this history node has done is remembered a shallow history of our `poweredOn` state. Let's change our example up a bit and create several states underneath `poweredOn` using parallel states.

[01:33] I'll move low and high to a heated state node, and I'll create enabled and disabled state nodes in our `oscillating` state node. Then I'll move my history state node as one of the parallel states.

```
const spaceHeaterMachine = Machine({
  id: 'spaceHeater',
  initial: 'poweredOff',
  states: {
    poweredOff: {
      on: { TOGGLE_POWER: 'poweredOn.hist' }
    },
    poweredOn: {
      on: { TOGGLE_POWER: 'poweredOff' },
      type: 'parallel',
      states: {
        heated: {
```

```
            initial: 'low',
            states: {
              low: {
                on: { TOGGLE_HEAT: 'high' }
              },
              high: {
                on: { TOGGLE_HEAT: 'low' }
              },
            }
          },
          oscillating: {
            initial: 'disabled',
            states: {
              disabled: {
                on: { TOGGLE_OSC: 'enabled' }
              },
              enabled: {
                on: { TOGGLE_OSC: 'disabled}
              }
            }
          },
          hist: {
            type: 'history'
          }
        }
      }
    }
  })
```

I'll update my machine, and we'll now see that I have parallel states of heated and `oscillating`. When I toggle the power on, I'm in low heat and I've disabled oscillation.

[01:59] Now by default, if I change these settings, and I toggle this off, it won't remember that I was in high heated and enabled oscillating. The reason for this is by default, this history is `shallow`. We can enable the ability for it to remember all child states and set them by setting `history` to `deep`.

```
hist: {
  type: 'history',
  history: 'deep'
}
```

[02:18] If we update our machine now, we can `TOGGLE_POWER` on. If we switch these to various settings, in this case, high and enabled, and we `TOGGLE_POWER` off and we toggle it back on, because of that deep setting, it remembers where we were in the child states of `poweredOn`.

Use XState Null Events and Transient Transitions to Immediately Transition States

Instructor: [00:00] There's a saying, "If at first you don't succeed, try, try again." I want to represent that as a machine. I have states of `idle`, `trying`, and `success`. Every time we enter the trying state, we're going to increment the number of tries in context.

```
states: {
  idle: {
    on: { TRY: 'trying' }
  },
  trying: {
    entry: ['incTries'],
  },
  success: {}
}
```

[00:16] I don't ever want to stay in the `trying` state. I should go back to the `idle` state automatically. If I do happen to try enough times and succeed, I should also automatically go to `success`. This seems like the kind of thing that should be chosen without an explicit event.

[00:32] In XState, we have something called the null event. On the `trying` state node, I'm going to add to the on property an empty string. This represents the null event. A null event is immediately taken when we enter a state. That's called a transient transition. Every time I enter trying, I'm going to immediately take the transitions I have here.

```
trying: {
  entry: ['incTries'],
  on: {
    ''
  }
}
```

[00:53] Now, with `trying`, I want to either transition to `success` if I've tried enough, or I want to transition back to `idle`. To do this, we'll set two targets by using an array. The first `target` will be `success` with a

condition. We'll write our tried enough condition in a moment.

[01:11] Our second `target` will be `idle`, in case our first condition isn't met.

```
trying: {
  entry: ['incTries'],
  on: {
    '': [
      { target: 'success', cond: 'triedEnough'
},
      { target: 'idle' }
    ]
  }
}
```

Down here in the second argument to machine the options object, we'll add our `guards`. We'll update our machine.

```
{
  actions: {
    incTries: assign({
      tries: ctx => ctx.tries + 1
    })
  },
  guards: {
    triedEnough: ctx => ctx.tries > 2
  }
}
```

[01:22] We can see now that we have an event that doesn't have a name. That's our null event. The null event is fired immediately upon entering the state and we attempt a transition to the next one. I find this very useful for setting up conditional branching in states like this.

[01:37] Let's give it a try on our machine. You notice that we never went into the trying state. We are immediately back into the idle state. If we look at the state panel, we'll see that our tries incremented by one.

[01:49] Let's try again. We see tries are two.

```
{
  "value": "success",
  "context": {
    "tries": 3
  }
}
```

Now that we've succeeded, we've tried enough times, we go immediately to success.

Delay XState Events and Transitions

Instructor: [0:00] Here I have your basic stop light machine with three states -- `red`, `green` and `yellow`. We transition between those states with a simple `TIMER` event. Now triggering these events manually, and I don't just mean with a mouse, it's a little tedious when we know that a light can just happen after a certain amount of time is passed.

[0:18] Wouldn't it be nice if we could delay transitions and events in xstate? We can. Very simply, we're going to remove the `on` Property and replace it with `after`. We can replace the event name with the number of milliseconds we'd like to transpire before we take the transition.

```
const stoplightMachine = Machine({
  id: 'stoplight',
  initial: 'red',
  states: {
    green: {
      after: {
        TIMER: 'yellow'
      }
    },
    yellow: {
      after: {
        TIMER: 'red'
      }
    },
    red: {
      after: {
        TIMER: 'green'
      }
    }
  }
})
```

[0:36] For the sake of this video, I'll make some of the shortest durations of stop lights you've probably ever seen. A `green` light will only last three seconds. A `yellow` light, one second. The `red` light, four seconds.

```
const stoplightMachine = Machine({
  id: 'stoplight',
  initial: 'red',
  states: {
    green: {
      after: {
        3000: 'yellow'
      }
    },
    yellow: {
      after: {
        100: 'red'
      }
    },
    red: {
      after: {
        400: 'green'
      }
    }
  }
})
```

[0:49] We'll update our machine and we could see that the timers have automatically started. Each time the timer is hit, we take the transition to the next state. It might be nice if we would give these times their own identifying names so that read a little bit nicer in the state chart.

[1:04] We can do this with the second argument to machine, the options object and define our delays. I am going to set the `GREEN_TIMER` to `3000`, the `YELLOW_TIMER` to `1000`, and the `RED_TIMER` to `4000`. Then in my machine, I'm going to replace this with `GREEN_TIMER`, `YELLOW_TIMER`, and `RED_TIMER`.

```
const stoplightMachine = Machine({
  id: 'stoplight',
  initial: 'red',
  states: {
    green: {
      after: {
        GREEN_TIMER: 'yellow'
      }
    },
    yellow: {
      after: {
        YELLOW_TIMER: 'red'
      }
    },
    red: {
      after: {
        RED_TIMER: 'green'
      }
    }
  }, {
    delays: {
      GREEN_TIMER: 3000,
      YELLOW_TIMER: 1000,
      RED_TIMER: 4000
    }
  }
})
```

[1:26] We update our machine. We now see that our events have updated though their times are the same. What's nice about doing this way is we can read this very clearly. The green timer happens, and then we go to yellow, and we can modify these. We can make these times dynamic if we'd like.

[1:44] We can actually set each of these as the result of a function that receives context and event. For instance, what if during rush hour we added a co-efficient that we could multiply these by? Maybe something that look like this.

[1:58] I am just going to copy-paste this into the other one.

```
delays: {
  GREEN_TIMER: ctx => ctx.rushHourMultiplier *
3000,
  YELLOW_TIMER: ctx => ctx.rushHourMultiplier *
1000,
  RED_TIMER: ctx => ctx.rushHourMultiplier *
4000
}
```

Now, let's add this to `context`. By default, we'll set it to one.

```
initial: 'red',
context: {
  rushHourMultiplier: 1
},
```

We'll update our machine. We'll see that the time doesn't change because we're multiplying by one.

[2:12] If we add a way to update this context, we'll see that the timers change. Let's add an `on` Event to the top level of our machine since we want to respond to it in any of the states that we're in. Now we need to add the `INC_RUSH_HOUR` action to our machine's configuration. I'll add that down here in `actions`. This will be in a sign that will set the `rushHourMultiplier` to its current context value plus one.

```
const stoplightMachine = Machine({
  id: 'stoplight',
  initial: 'red',
  context: {
    rushHourMultiplier: 1
  },
  on: {
    INC_RUSH_HOUR: {
      actions: ['incRushHour']
    }
  },
  states: {
    green: {
      after: {
        GREEN_TIMER: 'yellow'
      }
    },
    yellow: {
      after: {
        YELLOW_TIMER: 'red'
      }
    },
    red: {
      after: {
        RED_TIMER: 'green'
      }
    }
  }, {
    actions: {
      incRushHour: assign({
        rushHourMultiplier: ctx =>
ctx.rushHourMultiplier + 1
      })
    }
```

```
    delays: {
      GREEN_TIMER: ctx => ctx.rushHourMultiplier
* 3000,
      YELLOW_TIMER: ctx =>
ctx.rushHourMultiplier * 1000,
      RED_TIMER: ctx => ctx.rushHourMultiplier *
4000
    }
  }
})
```

[2:42] We'll update our machine and we'll see that we have added an `INC_RUSH_HOUR` event to our whole machine. I'm going to go the state tab really quick. I'm going to hit `INC_RUSH_HOUR`. We'll see that it's by two. You can actually see that the timers take twice as long.

```
{
  "value": "green",
  "context": {
    "rushHourMultiplier": 2
  }
}
```

[3:00] If I increment it again, maybe traffics are really heavy and we want to let them go by for long time, I can increment it to three and you can see that the timers are going even slower. We can actually see it down here the delay has been set to 9,000 for green, 3,000 for yellow, and it should 12,000 for red.

Invoking a Promise for Asynchronous State Transitions in XState

Instructor: [00:00] Here I have a state machine called `cuteAnimalsMachine` because who doesn't like some cute animals. I want to be able to fetch some animals. When they load, I want to know when it's successful to go to the `success` state over here or if it fails go to the `failure` state and give me the option of retrying.

```
const cuteAnimalMachine = Machine({
  id: 'cuteAnimals',
  initial: 'idle',
  context: {
    cuteAnimals: null,
    error: null,
  },
  states: {
    idle: {
      on: { FETCH: 'loading' },
    },
    loading: {},
    success: {
      type: 'final',
    },
    failure: {
      on: {
        RETRY: 'loading',
      },
    },
  },
})
```

[00:17] How can I do this with XState? You might notice that a promise looks a lot like a state machine itself. In fact, every promise can be represented as a state machine, with `idle`, `loading`, `success`, and `failure` states.

[00:30] Because of this fact, we can invoke promises when we enter a state. I'm going to quickly write a promise to get some cute animals from Reddit. Man, that's a lot of data properties for Reddit JSON.

```
const fetchCuteAnimals = () => {
  return
fetch('https://www.reddit.com/r/aww.json')
    .then(res => res.json())
    .then(data => data.data.children.map(child
=> child.data))
}
```

[00:43] Now we went to fetch that when we call this `fetch` event. When we call `fetch`, we'll transition to loading. Inside of `loading`, we're going to invoke our promise. We do this by using the `invoke` property. We can give this invocation an `id`. In this case, I'll give it the `id` of `fetchCuteAnimals`.

[01:03] We'll give it a `source`. In this case, it's the promise function we have up above. This is `fetchCuteAnimals`. Promises will respond to two specific events, `onDone` when they resolve and `onError` when they reject.

```
loading: {
  invoke: {
    id: 'fetchCuteAnimals',
    src; fetchCuteAnimals,
    onDone: {},
    onError: {}
  }
}
```

[01:19] We'll write the `onError` object first. In that case, we want to `target: 'failure'` as our transition, and we want to take some `actions`. Namely, we want to update the `error` in `context`. I'll just write that inline here. With promises, the `error` is returned on the `data` property of the event object.

```
onError: {
  target: 'failure',
  actions: assign({
    error: (context, event) => event.data
  })
}
```

[01:39] Now we'll write our `onDone` object. In this case, we'll `target` the `success` state node. We'll also take the `actions` of assigning the `cuteAnimals` in `context`. The `data` returned from resolve is also put on the data property of the `event` object.

```
onDone: {
  target: 'success',
  actions: assign({
    cuteAnimals: (context, event) => event.data
  })
}
```

[01:56] From here, we can update our machine. We could see it got a whole lot more complicated but a whole lot more useful. When we're in the idle state, we'll trigger the fetch. You see that the promise immediately resolved to success because it worked.

[02:10] I'm going to reset the machine really quickly. Let's just say, for good measure, that our fetchCuteAnimals doesn't work. We'll comment this out really quick. We'll `return Promise.reject()`, just to show that will go to the failure state.

```
const fetchCuteAnimals = () => {
  // return
fetch('https://www.reddit.com/r/aww.json')
  //   .then(res => res.json())
  //   .then(data =>
data.data.children.map(child => child.data))
  return Promise.reject()
}
```

[02:25] Notice it failed immediately. We're in the failure state. We hit retry. We know that's going to continue to fail. Removing this `return` and uncommenting this, we'll update our machine one more time. We could see that when we fetch, our state is updated, and all our cute animals are in context.

Invoke Callbacks to Send and Receive Events from a Parent XState Machine

Instructor: [00:00] Here I have an `echoMachine` with only one state, `listening`.

```
const echoMachine = Machine({
  id: 'echo',
  initial: 'listening',
  states: {
    listening: {
      on: {
        SPEAK: {},
        ECHO: {
          actions: () => {
            console.log('echo, echo')
          },
        },
      },
    },
  },
})
```

The idea of this machine is when the event `SPEAK` is called, I want to set up a callback that'll send `ECHO` events back to my machine if, and only if, the right type of event is sent. We can do this by invoking a callback as a service.

[00:20] To start, I'm going to add the `invoke` property on the `listening` state node. Invoke takes an `id` and takes a `src`. This source should be a Callback Handler. I'm going to call it `echoCallbackHandler`.

```
listening: {
  invoke: {
    id: 'echoCallback',
    src: echoCallbackHandler
  }
}
```

We'll write that function now.

[00:38] `echoCallbackHandler` is a function that receives the context and event that invoked the service. In this case, it'd be the initial `context` of our machine, which is undefined. The `event` would be the initialization of the machine.

[00:51] This function returns another function. This is where we manage our callback service. This function receives two arguments, a `callback` function that we can use to send events back to the parent machine, and an `onEvent` function that we can use to respond to specific events in the machine.

[01:09] Just for the sake of the concept, I want to respond to any event sent to my Callback Handler, so we'll add `onEvent`. This receives a function that receives that event. Inside here, I'm going to call this `callback` with `ECHO`. This callback will send an echo event to the `echoMachine` anytime an event is sent to our callback service.

```
const echoCallbackHandler = (context, event) =>
(callback, onEvent) => {
  onEvent(e => {
    callback('ECHO')
  })
}
```

[01:35] How do we send events to our callback service? On `SPEAK`, I'm going to add the `actions` property. I'm going to use the `send` action creator. This is a function that receives an event. In this case, it can be anything. What's important is that we add the options object, the second argument, and say where to send it to.

[01:57] I'm going to send it to the ID of my callback service, `EchoCallback`.

```
SPEAK: {
  actions: send('FOO', {
    to: 'echoCallback'
  })
}
```

This will send the Event `Foo` to the service `EchoCallback`. That'll be received by this onEvent function. `onEvent` will trigger the callback of `ECHO` back, which will lead to the `ECHO` event being handled in the listening state, which should log out `'echo, echo'`.

[02:22] I'm going to update my machine. I'm going to open the console. We're going to send the event. You could see, when we send it, the Echo event is sent in the callback and the actions are triggered.

[02:38] To take this a step further, I'd like to only respond this Echo callback when the event is of a certain type. I'm going to say, if `e.type === 'HEAR'`, as in, I want to hear this thing, then callback Echo.

```
const echoCallbackHandler = (context, event) =>
(callback, onEvent) => {
  onEvent(e => {
    if (e.type === 'HEAR') {
      callback('ECHO')
    }
  })
}
```

Going to update my machine. We're still sending Foo to `EchoCallback`.

[03:02] If I open up the console and I send the speak event, which sends the FooEvent to my Callback Handler, we'll see that it doesn't echo. If we update `SPEAK` to the type `HEAR`, and update the machine, we now see that it works again.

Invoke Child XState Machines from a Parent Machine

Instructor: [00:00] Here, I have a machine that I am calling parent with states of `idle`, `active`, and `done`. It starts in the `idle` state. We can see an `ACTIVATE` event sends it to the `active` state. I haven't written a way for it to go from active to done. That is because in the active state, I would like to invoke another machine and let that machine determine when I move to the done state.

```
const parentMachine = Machine({
  id: 'parent',
  initial: 'idle',
  states: {
    idle: {
      on: { ACTIVATE: 'active' },
    },
    active: {
      // invoke another machine
    },
    done: {},
  },
})
```

[00:22] Up here, I've written a second machine. This is `childMachine`. It has several steps of states. `step1`, `step2`, and `step3`, and we can step through them to get to the `final` state.

```
const childMachine = Machine({
  id: 'child',
  initial: 'step1',
  states: {
    step1: {
      on: { STEP: 'step2' },
    },
    step2: {
      on: { STEP: 'step3' },
    },
    step3: {
      type: 'final',
    },
  },
})
```

[00:33] I can invoke this machine from a state within the parent machine by using the `invoke` property. We'll give an `id`, set to child, and a `src` of `childMachine`. When we enter the active state, it will invoke this child machine and it will start it in its initial state.

```
active: {
  invoke: {
    id: 'child',
    src: childMachine
  }
}
```

[00:52] Let's update the machine. We could see that invoke child has been updated. We go to state, we're currently in idle. Let's activate it. When we get into the value active, that is the state of active, we can see that we now have a child state as well. We're in the value of step one.

[01:10] Currently, xstate-viz doesn't support showing the child machine as well. We'll rely on the state panel to indicate where we are in the child machine. Right now, I have no way of sending events to this child machine. How can I do that?

[01:26] In the `active` state node of the parent machine, I am going to add the `on` Property to send events. I'll create an event `STEP`. On this event, I'd like to have an `actions` occurred, which is I want to send a `STEP` event to my child machine.

```
on: {
  STEP: {
    actions: send('STEP', {
      to: 'child'
    })
  }
}
```

We'll update the machine.

[01:47] We can see now that when we're in active, we have the step event available to us. Let's open up the state panel really quick and let's send that step event. We can see that we send a step event to the child and the child is now in step two.

[02:04] If I hit the step event again, I get to step three. Step three is a final state for the child machine. How do we want to respond to that in our parent machine?

[02:15] We can do that under the invoke property with the `onDone` property. This will fire whenever the child machine reaches a final state. When our child machine is finished, in a final state, we'd like to go the done state in our parent machine.

```
active: {
  invoke: {
    id: 'child',
    src: childMachine,
    onDone: 'done'
  }
}
```

[02:32] We update our machine. I am going to back to the state tab again. We'll activate it. The child machine has been invoked. We can see down here. It's at value of step one. We send a step event to our parent machine, which sends it to our child machine, and we're now in step two.

[02:51] When we send a step event again, our child goes in the step three, which is the final state. That final state triggers the onDone and moves us into the done state.