INDIAN INSTITUTE OF TECHNOLOGY MADRAS

सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

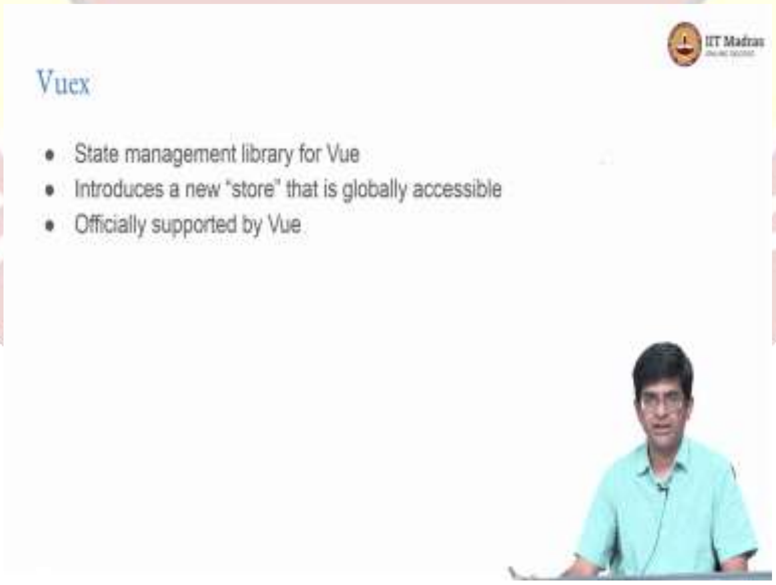
Modern Application Development - II
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Introduction to Vuex

(Refer Slide Time: 00:14)

Vuex

Now, let us look at Vuex.

(Refer Slide Time: 00:17)



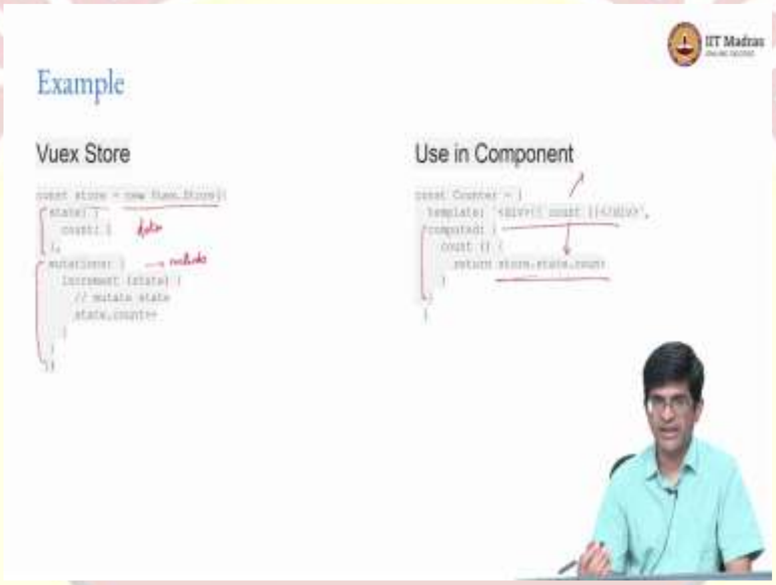
Vuex

- State management library for Vue
- Introduces a new "store" that is globally accessible
- Officially supported by Vue

So, what is Vuex? It is a state management library for the vue dot js framework. And it effectively implements most of the ideas of state management that we have seen before. What it does is introduces a concept called a store that is accessible globally. We have already sort of seen why it makes sense to have the state visible globally to all the components.

And the good thing about Vuex is that it is officially supported by vue, which means that any updates to the vue dot js framework are most likely also going to see updates in the Vuex state management library. And it will usually be kept in sync and it is very popularly, popular and widely used. So, you can expect it to be maintained fairly well as well.

(Refer Slide Time: 01:06)



Example

Vuex Store

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      // mutate state
      state.count++
    }
  }
})
```

Use in Component

```
const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return store.state.count
    }
  }
}
```

So, what we are going to do is just look at a couple of examples of code, but not necessarily trying to sort of see how they work. We will not be looking at examples of functioning code over here. What the Vuex Store looks like, for example, would be that you basically declare something as a variable, as a new instance of Vuex dot Store. So, Vuex dot Store is essentially a class. And a new instance of that class is instantiated at the top level app.

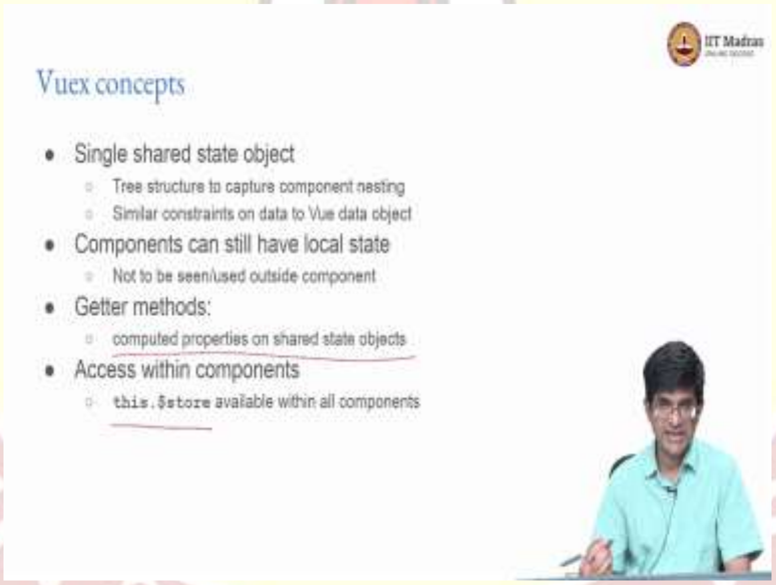
And effectively, an example of what it would have is, it would have its own notion of state, instead of data. So, where we usually had data in the terms of the Vue app, we now call it the state. And you will also see that there is something called mutations, which looks a little bit like the methods that were defined on the Vue app. So, what exactly are mutations? We will get to those later.

So, now, how do you use state in a component? We would be able to directly, for example, do something like this, just like you could declare a component where it could say that it had access to a variable called count. I could say that, I have a computed variable which returns store dot state dot count. The alternative would have been directly over here, I could have this value, store dot state dot count.

To make it a bit easier, in some cases, what you do is you have the store dot state, which has access to the variables. And in order to make your internal access within the component a bit easier, you declare computer components, computed values that derived from those state variables. Mostly optional, that is more a question of convenience.

But ultimately, remember, why are we even using these patterns. It is more about the readability and trying to make the code understandable and maintainable. So, having some of these instances over here can help to do that. Make it a bit cleaner to read and understand what the code is all about.

(Refer Slide Time: 03:13)



Vuex concepts

- Single shared state object
  - Tree structure to capture component nesting
  - Similar constraints on data to Vue data object
- Components can still have local state
  - Not to be seen/used outside component
- Getter methods:
  - computed properties on shared state objects
- Access within components
  - `this.$store` available within all components

So, a few of the concepts associated with Vue. One is of course that there is a single shared state object. So, it is a global variable. It is shared across all the components. It has a tree structure. Remember what I said about the component nesting being a tree kind of structure. So, therefore, the state object also has a tree structure. So, it sometimes called the shared state tree.

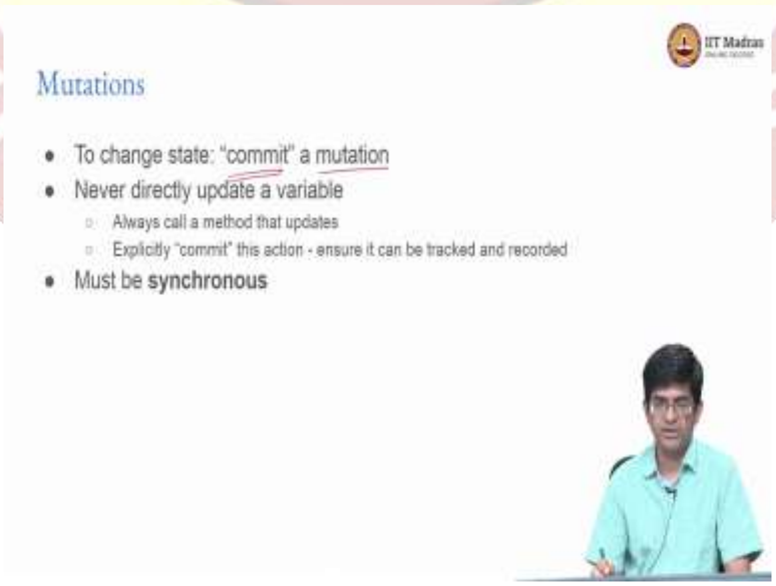
And in terms of what data can be present inside the shared state object, it is pretty much the same constraints as whatever you can have inside a Vue data object. So, what you could put inside Vue data, you can also have as a shared state object over here inside the Vuex framework. Now, a component can still have local state. It can have its own data variables, which are not passed back and forth to siblings or to even the parents. That is still acceptable. Vuex does not prevent you from doing that.

And in some cases, it may make sense. If a component very clearly only has certain variables that it does not need to communicate with the outside world, it probably makes sense to keep them internal to the component, because otherwise the amount of data in the global store becomes large. More importantly, it becomes hard to understand why they are there, because they are not used by anyone except one particular component.

Now, this idea of a getter method is something that basically says, okay, how do I access a particular value. So, there are certain kinds of getter methods that are specified to make the code easier. So, rather than particularly having to go and access the complete dot, dot, dot, way of getting to the particular value, there may be a simpler way by which you can access certain values. Once again, this is mostly for convenience.

And the other thing is, when you register Vuex using the normal recommended base of coding, there is a variable called this dot dollar store that becomes available within all the components, which means that any component can basically access the store, the global store through this dot dollar store, and in turn can then you know, access the variables inside the store. So, all of these are the basics of how data gets stored inside the shared state of a Vuex, of the Vuex library.

(Refer Slide Time: 05:43)



Mutations

- To change state: "commit" a mutation
- Never directly update a variable
  - Always call a method that updates
  - Explicitly "commit" this action - ensure it can be tracked and recorded
- Must be **synchronous**

Now, the question becomes how do you make any changes to that state? And as we already mentioned, one of the main points of that state management pattern was to prevent arbitrary changes. So, now what we say is in Vuex, if you want to change state anywhere, you change a variable, you have to do something called a mutation, and in particular, you have to commit a mutation.

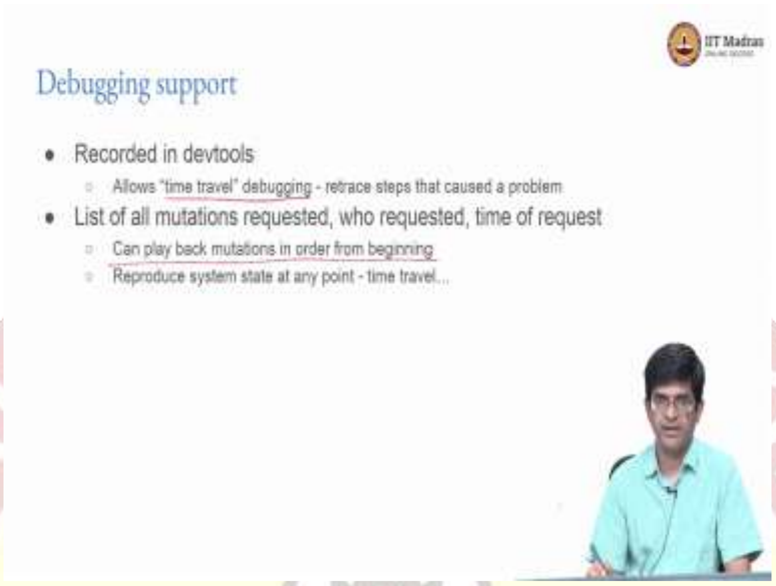
Now, this commit is in a similar sense to what you would see in a database transaction or in something like a git commit. What is happening in each of those cases is that you are sort of explicitly saying, I want this change to happen now, and you are saying, okay, we are not going to sort of, in once I make this change, it is recorded, it cannot be undone easily.

So, committing a mutation, in other words, will pass on some information to the internal state variable or internal state object, which tells it to update a certain value in a certain way. So, you never directly update a variable. You have access to them, but you never go and directly access, updater those variables. You always call a method that updates variable. And you explicitly commit this action.

Now, the nice thing about this commit way of doing things is that I can actually add an extra sort of piece of code inside the commit function that says, anytime I am committing a mutation, I will also track it, meaning that maybe I will either in local storage or maybe just in memory, keep track of all the changes and the time at which it happened so that I can get some useful functionality out of it. What is that, we will see in a moment.

Now, one important thing to keep in mind as far as mutations are concerned is that anything happening inside a mutation must be synchronous that has implications later. So, we will get to that in a moment. But for now just keep in mind that anything that happens inside a mutation must be synchronous code.

(Refer Slide Time: 07:46)



Debugging support

- Recorded in devtools
  - Allows "time travel" debugging - retrace steps that caused a problem
- List of all mutations requested, who requested, time of request
  - Can play back mutations in order from beginning
  - Reproduce system state at any point - time travel...

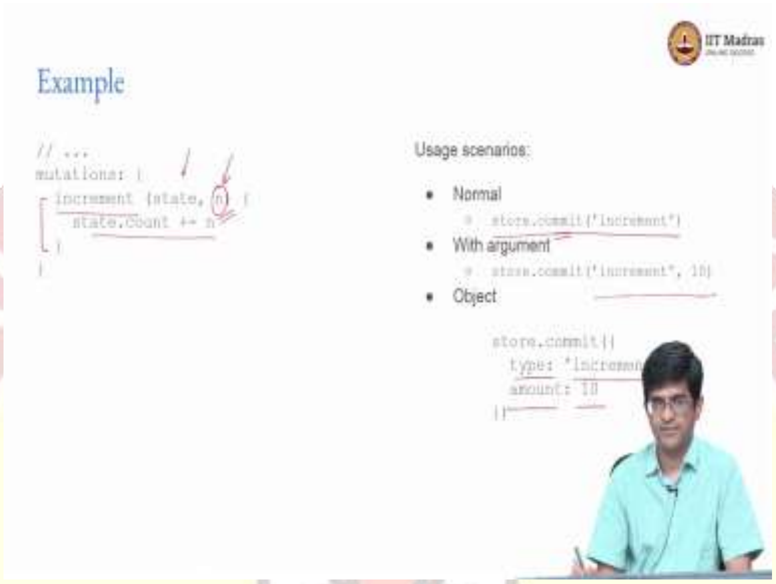
So, the, like I said, you can track what happens inside when mutations are committed. And what is usually done is that the Vue dev tools, the extensions that we have for debugging, allow you to record what happens in each mutation. So, every time a mutation is committed, something gets logged so to say. It could either be just in the memory of the system or it might even be written out somewhere.

In the dev tools, it is basically the sort of logged in memory. The important point is that every single mutation that was requested, who requested it, the time of request, all of that information gets stored inside the devtools memory, which means that you can do something called time travel debugging. What is time travel debugging? You can basically go back and play the mutations in order from the beginning.

At any point, in principle, you could sort of step backwards, but stepping backwards is a little bit more complicated. It means you need to store the previous state and the previous state and the previous state all the way back to the beginning. In some cases, it is simpler to say, if I want to go back one step, I will just reset the system and start from the beginning and played till the n minus oneth step.

If I am, if I have finished n mutations, I will reset and play up to n minus 1 in order to get back to the same state, which means that we can basically reproduce the system state at any point. And this is what is given the nice sounding term time travel. It sort of allows you to go back and forth in the history of the system, which makes debugging a lot easier.

(Refer Slide Time: 09:27)



Example

```
// ...
mutations: {
  increment (state, n) {
    state.count += n
  }
}
```

Usage scenarios:

- Normal
  - `store.commit('increment')`
- With argument
  - `store.commit('increment', 10)`
- Object

```
store.commit({
  type: 'increment',
  amount: 10
})
```

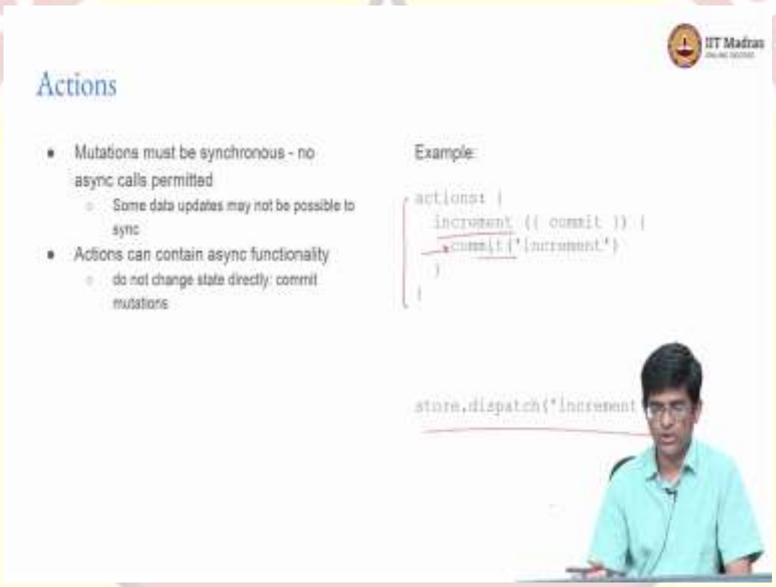
So, an example of what a mutation would look like, it would be defined simply like a function. It is increment. And it could basically take, of course, the state, because after all, this is a mutation that is defined with respect to a Vuex state object. So, it has the state associated with it, the state variable. In this case, what we are saying is, I would also give it some additional parameter in which is how much to increment a particular counter by. And what we are saying is this method or mutation when it is committed will change increment the state count by the value n.

A normal way of calling this would be something like store dot commit increment. Now, in this case, increment is also expecting the second value n. So, unless n has a default of 1, we would probably need to define another function out there, which says that if I, if it is called without this second argument, it would just be called as a commit to increment. You could also give it an argument by saying store commit increment, but now with an argument 10, which will basically be fed in as the second value to the increment function.

And there is yet another notation that basically says you can do store dot commit and pass the entire thing as an object, where the commit type is specified as increment and the amount is given over here as 10. It should probably be the n over here which is given as 10. So, you could specify more details and give them all together in an object, which is then passed into the increment function and used in order to perform the computation.

The important point, I mean, this is just, the right hand side is more about syntax. How do you use it? The important point is that the mutation itself being committed means that internally the devtools, for example, could go and say that, anytime the increment mutation is committed, it gets logged. It does not have to go and change the increment function. The commit function itself will take care of that.

(Refer Slide Time: 11:27)



Actions

- Mutations must be synchronous - no async calls permitted
  - Some data updates may not be possible to sync
- Actions can contain async functionality
  - do not change state directly: commit mutations

Example:

```
actions: {
  increment ({ commit }) {
    commit('increment')
  }
}
```

```
store.dispatch('increment
```

So, now, remember what I said about mutations being synchronous. Let us look at this example. You want to do time travel debugging. You want to be able to sort of say, what is the state of the system after each and every mutation. The problem with async operations is that they basically come back after some time. There is a callback. So, it could either be a timer based callback or a fetch API based callback or something else, which says that I do not immediately have a result.

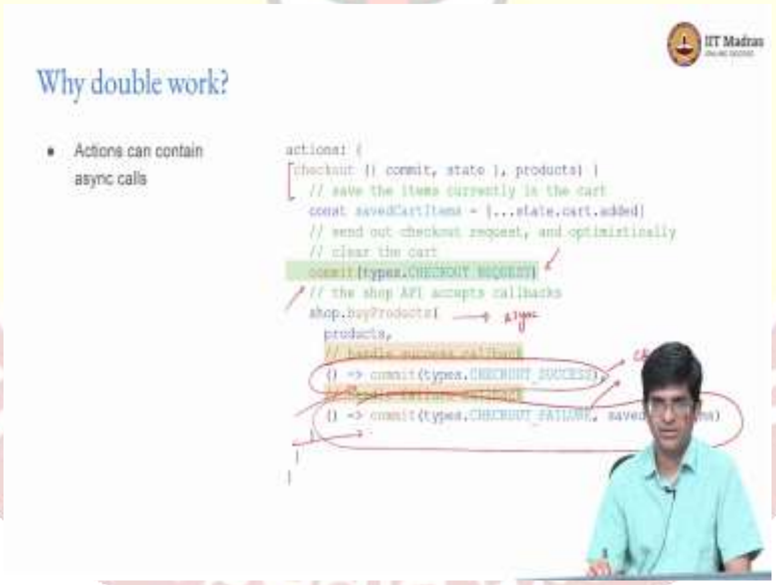
Now, what do you do in such a case? You cannot really record that as a mutation, and then say, I will be able to time travel. I cannot really go back easily. So, mutations, in other words, put a restriction saying, look, if you are committing a mutation, it has to be instantaneous, it has to be deterministic, it has to be synchronous, not deterministic, but synchronous, something which basically is computed immediately and updated.

So, no asynchronous calls are permitted. But there are cases where you would want to have asynchronous updates. So, you bring in a new notion of something called an action. And an action is something that can contain asynchronous functionality. And the way that it is usually done is an action could also be defined with the same name increment. And in this case, what it could say is basically that, what it needs to do is it needs to commit the increment mutation.

And the way that you would call it is rather than store dot commit increment, you will say store dot dispatch increment. The wording has a slightly different meaning. In English dispatch basically means to send a message. So, you are sort of dispatching the increment action. And the idea behind it is you do not expect the result to come back immediately.

If you dispatch it, it then in turn goes, commits the actual mutation, and then comes back. In this case, it might be instantaneous, but even if it is not, it is okay, an action is permitted to come back later. It looks like it is overkill, I mean, what difference does it make whether a call is store dot dispatch increment or store dot commit increment.

(Refer Slide Time: 13:38)



Why double work?

- Actions can contain async calls

```
actions: {
  checkout ({ commit, state }, products) {
    // save the items currently in the cart
    const savedCartItems = [...state.cart.added]
    // send out checkout request, and optimistically
    // clear the cart
    commit(types.CHECKOUT_REQUEST)
    // the shop API accepts callbacks
    shop.buyProducts(
      products,
      // handle success callback
      () => commit(types.CHECKOUT_SUCCESS),
      // handle failure callback
      () => commit(types.CHECKOUT_FAILURE, savedCartItems)
    )
  }
}
```

Well, there are cases, for example, like this, where an action can contain asynchronous calls. Think about some hypothetical case from the, this is from the Vuex examples, where maybe you are trying to create checkout shopping cart. So, things like the shopping cart request itself can be committed. It is a simple thing to do. You immediately are just changing a state variable.

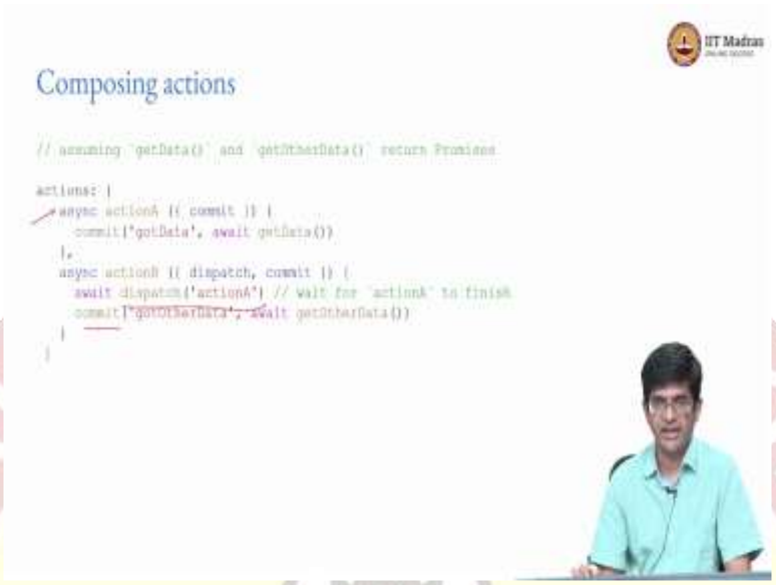
But then after that, the actual byproducts, this function is probably a much more involved thing. It takes you to a page where you need to enter the purchase information, then it comes back from there with either success or failure, which is why this is an asynchronous operation. We cannot expect it to come back immediately. And what it does is, in turn, it could have its own callbacks. So this, these two things are essentially callback functions.

And in the case of a success, it basically commits a new mutation which basically says, it succeeded. So, this entire thing, it is not just the commit types dot success, it is not this, this entire function over here is the success callback. And similarly, this entire function out here is the failure callback. In other words, if the async operation corresponding to byproduct succeeds, it would call the success callback. If it failed, it would call the failure callback. And either way something would get updated.

So, the point is this checkout could no longer be a mutation, because it contains asynchronous operations within it. But by changing it to actions, we allow it to at least we record it in that way. What does it mean? It means that full time travel debugging through an action directly is not possible. But the action in turn is calling a number of commit operations over here, and each of those in turn would get recorded properly.

So, it makes sure that even though you have some kind of asynchronicity, the actual mutations finally do get committed in the order in which they happened at some point. So, that is why you have these additional actions. And even though it looks a little bit like double work for simple cases, it makes sense for slightly more complicated cases over here.

(Refer Slide Time: 16:04)



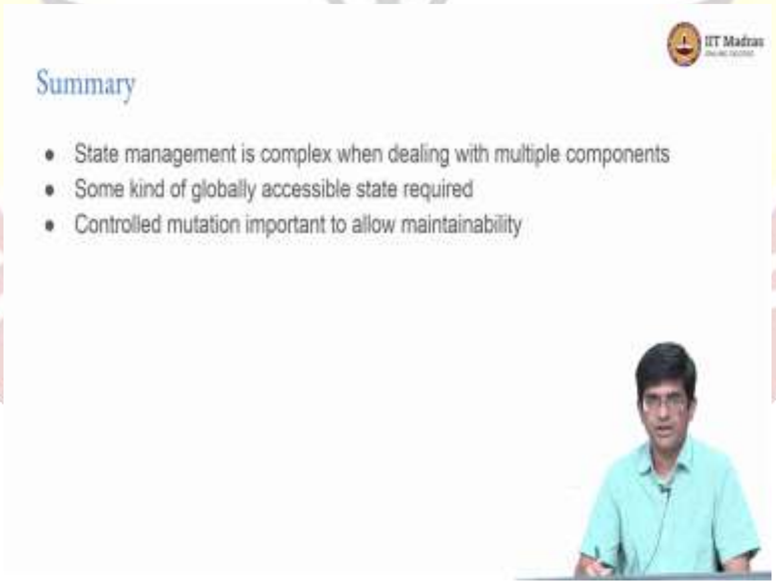
Composing actions

```
// assuming `getData()` and `getOtherData()` return Promises

actions: {
  async actionA ({ commit }) {
    commit('gotData', await getData())
  },
  async actionB ({ dispatch, commit }) {
    await dispatch('actionA') // wait for `actionA` to finish
    commit('gotOtherData', await getOtherData())
  }
}
```

You can also compose actions. You could define one action. And you could have another action, which basically dispatches the first action. Waits for a result from there. And then, finally does a commit. So, there are ways by which you could have multiple actions being composed one on top of another and get more interesting functionality overall.

(Refer Slide Time: 16:28)



Summary

- State management is complex when dealing with multiple components
- Some kind of globally accessible state required
- Controlled mutation important to allow maintainability

So, to summarize all of this, the Vuex state management pattern is something which it is complex. I mean, the point is, state management is quite complex when you are dealing with multiple components and some kind of globally accessible state is required. The most important thing is you need to control how and when the state variables can get changed or mutated, so that you have maintainability of the code. Now, the fundamental question over here, of course, becomes should you just be using Vuex in every app that you build? And the short answer is no.

And in fact, if you look at the Flux documentation, the Redux documentation, there are a number of instances where people basically say, okay, a lot of people ask this question, should we be using Flux or Redux or some kind of state management pattern or library? And the simple answer is, if you have got to the point where your state management is getting too complicated, and you are really thinking of, is there a better way to do this, that is when you should be looking at libraries like Flux, or in this case, Vuex.

If you are dealing with fairly simple hierarchies of components that do not have much of nested data, you do not have like global, you do not have multiple vues that are trying to update the same state and so on, you are probably better off just keeping with the basic vue and not really worrying too much about Vuex. Now, clearly, the more complicated your app gets, the more likely you are to need Vuex. So, it is good to be familiar with it at least at some point.