Intro to Vuex

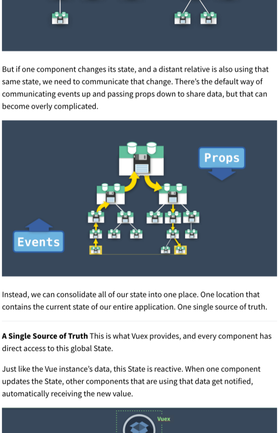
In this course, we'll be exploring the fundamentals of Vuex: Vue's state management library. If you've been following along with our beginner path, this course will pick up where Real World Vue 3 left off. By the end of this course, you'll have a solid understanding of when and why to use Vuex, and you'll be empowered to implement it within your own Vue apps. Lesson by lesson, we'll be adding Vuex to the example app that we created in the Real World Vue 3 course.

But before we get started writing any code, we need to understand the rationale behind Vuex, and look at an example use case that illustrates the different parts of Vuex and how it all works together.

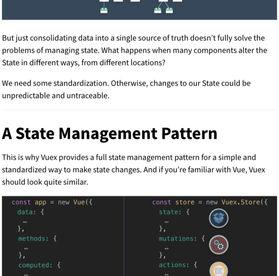
The Case for State Management

Managing state in an application full of components can be difficult. Facebook discovered this the hard way and created the Flux pattern, which is what Vuex is based upon. Vuex is Vue's own state management pattern and library. In this lesson, we'll look at why an application might need Vuex, and how it can enhance your app.

When we talk about state, we mean the data that your components depend on and render. Things like blog posts, to-do items, and so on. Without Vuex, as your app grows, each Vue component might have its own version of state.



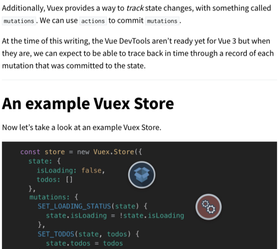
But if one component changes its state, and a distant relative is also using that same state, we need to communicate that change. There's the default way of communicating events up and passing props down to share data, but that can become overly complicated.



Instead, we can consolidate all of our state into one place. One location that contains the current state of our entire application. One single source of truth.

A Single Source of Truth This is what Vuex provides, and every component has direct access to this global State.

Just like the Vue instance's data, this State is reactive. When one component updates the State, other components that are using that data get notified, automatically receiving the new value.

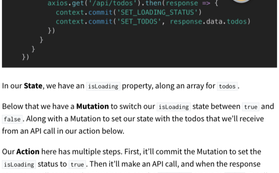


But just consolidating data into a single source of truth doesn't fully solve the problems of managing state. What happens when many components alter the State in different ways, from different locations?

We need some standardization. Otherwise, changes to our State could be unpredictable and untraceable.

A State Management Pattern

This is why Vuex provides a full state management pattern for a simple and standardized way to make state changes. And if you're familiar with Vue, Vuex should look quite similar.



Just as you can create a new Vue instance (or Vue app) with `createApp()`, you can create a Vuex store with `createStore()`.

While the Vue instance has a `data` property, the Vuex store has `state`. Both are *reactive*.

And while the instance has `methods`, which among other things can update `data`, the store has `actions`, which can update the state.

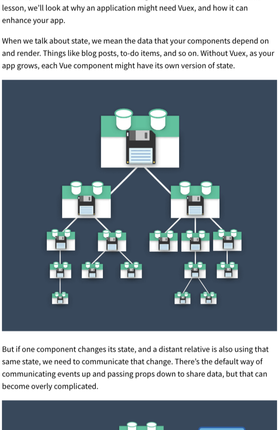
And while the instance has computed properties, the store has `getters`, which allow us to access a filtered, derived, or computed version of our `state`.

Additionally, Vuex provides a way to *track* state changes, with something called `mutations`. We can use `actions` to commit `mutations`.

At the time of this writing, the Vue DevTools aren't ready yet for Vue 3 but when they are, we can expect to be able to trace back in time through a record of each mutation that was committed to the state.

An example Vuex Store

Now let's take a look at an example Vuex Store.



In our **State**, we have an `isLoading` property, along an array for `todos`.

Below that we have a **Mutation** to switch our `isLoading` state between `true` and `false`. Along with a Mutation to set our state with the todos that we'll receive from an API call in our action below.

Our **Action** here has multiple steps. First, it'll commit the Mutation to set the `isLoading` status to `true`. Then it'll make an API call, and when the response returns, it will commit the Mutation to set the `isLoading` status to `false`. Finally it'll commit the Mutation to set the state of our `todos` with the response we got back from our API.

If we need the ability to only retrieve the todos that are labeled done, we can use a Getter for that, which will retrieve only the specific state that we want.

Now let's take a look at this in motion.

Vuex in Motion

Next up...

Hopefully you now understand why you might need Vuex and how it can help enhance your application by providing a single source of truth for your State, along with a common library of Actions, Mutations and Getters.

In the next lesson, we'll start implementing Vuex into our example application we built in Real World Vue 3.

Global State

In this lesson, we're going to implement some global state within our example application.

Updating State

How do we update Vuex state? Learn how to use Mutations to add new State or update State within the Vuex Store.

Fetching State

Learn how Vuex actions can wrap your mutations to perform more complex state management behavior.

Error Handling

Learn how to handle errors that might happen when dispatching Vuex actions.

Next Steps

Let’s explore some of the additional Vuex features we can add to our app as it s

Unlock this lesson by subscribing to a plan.

Unlock Content