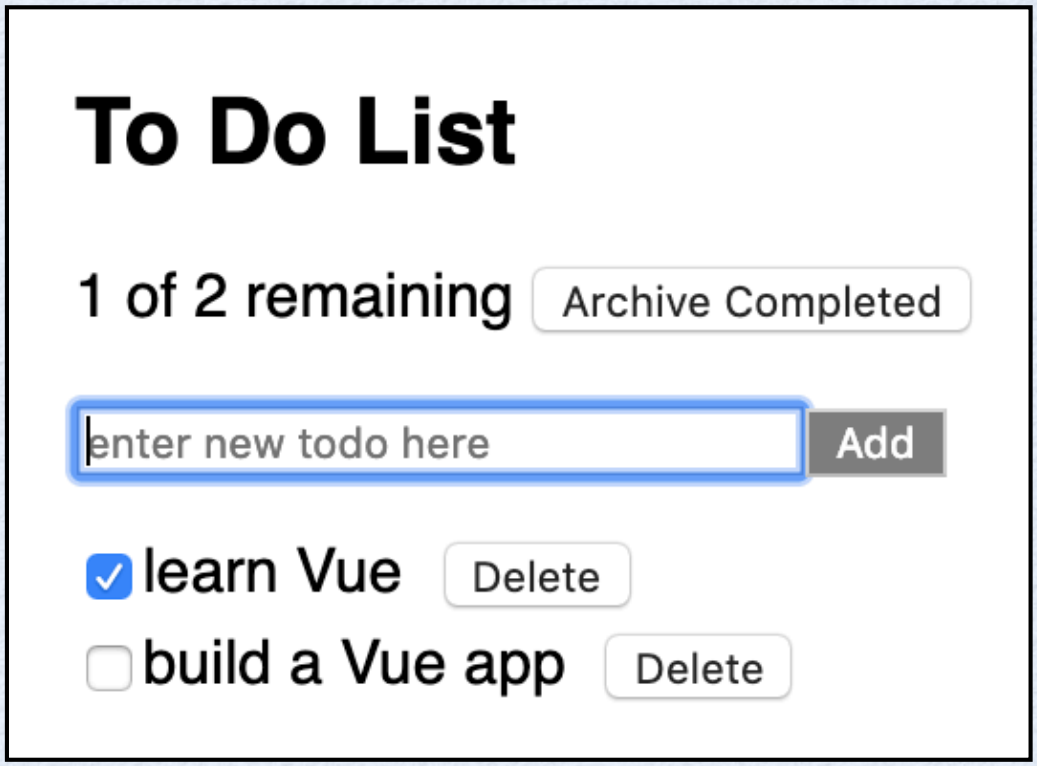
Vuex

Vuex Overview

- Most popular state management library for Vue
- Developed by and maintained by the Vue team
- Somewhat similar to Redux
  - uses a single store to hold application state
- Vue CLI can install and configure Vuex
- To install when not using Vue CLI,
  `npm install vuex`

Demo App

- To demonstrate use of Vuex, we'll build a simple Todo app
- Source files to review
  - `store.js`
  - `App.vue`
  - `TodoList.vue`
  - `Todo.vue`



Store Setup

- Typically done in **`src/store.js`**
- Vue CLI creates this file
- Store has four parts
  - **state** - stores application data
  - **mutations** - modify state
  - **getters** - compute values based on state
  - **actions** - perform asynchronous actions (like REST calls) and invoke mutations

Mutations

- Only way to modify state
- Defined by a set of synchronous methods (no REST calls)
- Invoked in components by calling `this.$store.commit`, not direct calls
- Within a mutation there are two ways to update a state property
  - for example, to change property `foo.bar` to `'baz'` use
    `state.foo.bar = 'baz'`
    or
    `Vue.set(state.foo, 'bar', 'baz')`

    Why would this form ever be preferred?
- Mutations do not need to treat state as immutable
  - except to set a specific array element
  - rather than using syntax like `state.colors[2] = color`
    use `Array splice` method like `state.colors.splice(2, 0, color)`
    or Vue-supplied method `$set` like `state.colors.$set(2, color)`

store.js ...

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

let lastId = 0;
const createTodo = (text, done = false) => ({id: ++lastId, text, done});

export default new Vuex.Store({
  strict: true, // throws if state is modified outside a mutation
  state: {
    todoText: '',
    todos: [
      createTodo('learn Vue', true),
      createTodo('build a Vue app')
    ]
  },
```

... store.js

```
  mutations: {
    addTodo(state) {
      const todo = createTodo(state.todoText);
      state.todos.push(todo);
      state.todoText = '';
    },
    archiveCompleted(state) {
      state.todos = state.todos.filter(t => !t.done);
    },
    deleteTodo(state, todoId) {
      state.todos = state.todos.filter(t => t.id !== todoId);
    },
    toggleDone(state, todo) {
      const todoToToggle = state.todos.find(t => t.id === todo.id);
      todoToToggle.done = !todoToToggle.done;
    },
    updateTodoText(state, todoText) {
      state.todoText = todoText;
    }
  },
  getters: {
    uncompletedCount: state => state.todos.filter(t => !t.done).length
  }
});
```

just removes completed todos in this version

App.vue

- Registers store which "injects" access to it into all descendant components

```
<template>
  <TodoList/>
</template>

<script>
import TodoList from './components/TodoList';
import store from './store';

export default {
  components: {TodoList},
  store
};
</script>

<style>
body {
  font-family: sans-serif;
  padding-left: 10px;
}
</style>
```

State

- Any component can access with **`this.$store.state`**
- When many items are needed from the state, it is convenient to use **`mapState`** to make them accessible via computed properties
- **`mapState`** returns an object that should be spread into the **`computed`** object
- Example

will appear in **`TodoList.vue`**

```
computed: {
  ...mapState({
    todos: state => state.todos,
    todoText: state => state.todoText
  })
},
```

Committing Mutations

- To commit mutations from a component, call
  `this.$store.commit(mutationName, arg)`
- Example
```
this.$state.commit('toggleDone', todo);
```
- Only a single argument can follow mutation name
- To supply more than one value,
  pass an array or object containing all the values
- Any components that use state
  affected by the mutation will be updated

mapMutations

- Another option is to use **mapMutations** to generate methods that make these calls for you
- Returns an object that should be spread into the **methods** object
- Example

```
methods: {
  ...mapMutations([
    'addTodo',
    'archiveCompleted',
    'deleteTodo',
    'toggleDone',
    'updateTodoText'
  ])
}
```

will appear in **TodoList.vue**

- Allows **this.$state.commit('toggleDone', todo)** to be replaced by **this.toggleDone(todo)**

Getters

- Set of methods defined in the store
  that compute derived state from stored state
- Earlier we saw this one (slide 7)

in `store.js`

```
getters: {
  uncompletedCount: state => state.todos.filter(t => !t.done).length;
}
```

- To retrieve value of this getter in a component, call
  `this.$store.getters.uncompletedCount()`

mapGetters

- Can create computed properties from getters
  so they can be referred to with computed property names
  instead of explicit calls
- **mapGetters** returns an object that
  should be spread into the **computed** property
  - takes an array of getter method names
- Example

```
computed: {
  ...mapGetters(['uncompletedCount'])
},
```

will appear in **TodoList.vue**

Todo.vue

```
<template>
  <li>
    <input type="checkbox" :checked="todo.done" @change="onToggleDone">
    <span :class="doneClass">{{todo.text}}</span>
    <button @click="onDeleteTodo">Delete</button>
  </li>
</template>

<script>
export default {
  props: {
    onDeleteTodo: {
      type: Function,
      required: true
    },
    onToggleDone: {
      type: Function,
      required: true
    },
    todo: {
      type: Object,
      required: true,
      validator(obj) {
        return obj.text &&
          obj.done !== undefined;
      }
    }
  },
  computed: {
    doneClass() {
      return 'done-' + this.todo.done;
    }
  }
};
</script>

<style scoped>
button {
  margin-left: 10px;
}

li {
  margin-top: 5px;
}

.done-true {
  color: gray;
  text-decoration: line-through;
}
</style>
```

TodoList.vue ...

```
<template>
  <div>
    <h2>To Do List</h2>
    <div>
      {{uncompletedCount}} of {{todos.length}} remaining
      <button class="archive-btn" @click="archiveCompleted">
        Archive Completed
      </button>
    </div>
    <form @submit.prevent>
      <input
        class="todo-input"
        type="text"
        size="30"
        autofocus
        placeholder="enter new todo here"
        :value="todoText"
        @input="updateTodoText($event.target.value)"
      >
      <button
        class="add-btn"
        :disabled="!todoText"
        @click="addTodo"
      >Add</button>
    </form>
    <ul class="unstyled">
      <Todo v-for="todo in todos"
        :key="todo.id"
        :todo="todo"
        :onDeleteTodo="() => deleteTodo(todo.id)"
        :onToggleDone="() => toggleDone(todo)"
      />
    </ul>
  </div>
</template>
```

`todos` and `todoText` are mapped from state

`updateTodoText`, `addTodo`, `deleteTodo`, and `toggleDone` are all mutations

passing functions that call mutation functions on next slide

... TodoList.vue

```
<script>
import {mapGetters, mapMutations, mapState} from 'vuex';
import Todo from './Todo.vue';

export default {
  components: {Todo},
  computed: {
    ...mapGetters(['uncompletedCount']),
    ...mapState({
      todos: state => state.todos,
      todoText: state => state.todoText
    })
  },
  methods: {
    ...mapMutations([
      'addTodo',
      'archiveCompleted',
      'deleteTodo',
      'toggleDone',
      'updateTodoText'
    ])
  }
};
</script>
```

```
<style scoped>
button:disabled {
  background-color: gray;
  color: white;
}

ul.unstyled {
  list-style: none;
  margin-left: 0;
  padding-left: 0;
}
</style>
```

Actions

- Set of methods that support asynchronous mutations
- Can make any number of REST calls
  and commit any number of mutations
- Invoked by calls to `this.$store.dispatch` in components,
  not direct calls
- Example
  - suppose we have REST services that persist todos in a database
  - can implement actions that make the REST calls
  - on success they can commit mutations

Action Examples

```
actions: {
  async addTodo(context) {
    const todo = createTodo(context.state.todoText);
    const res = await fetch(SERVER_URL, {
      method: 'POST',
      headers: {'Content-Type': 'application/json'},
      body: JSON.stringify(todo)
    });
    if (res.ok) {
      context.commit('addTodo', todo);
    } else {
      alert('Error adding todo');
    }
  },
  async deleteTodo(context, todoId) {
    const res = await fetch(SERVER_URL + todoId, {method: 'DELETE'});
    if (res.ok) {
      context.commit('deleteTodo');
    } else {
      alert('Error deleting todo');
    }
  }
}
```

add in `store.js`

change `addTodo` mutation to use the passed `todo` instead of creating a new one

mapActions

- An alternative to calling `this.$state.dispatch` is to use `mapActions` to generate methods that make these calls for you
- Returns an object that should be spread into the `methods` object
- Example

```
methods: {
  ...mapActions([
    'addTodo',
    'deleteTodo'
  ])
}
```

- Allows `this.$state.dispatch('addTodo', todo)` to be replaced by `this.addTodo(todo)`

Are Actions Needed?

- Any asynchronous processing, such as calling a REST service,
  can be done in an event handling method instead of an action
- When the asynchronous part completes,
  a synchronous Vuex `commit` can be performed
- If common event handling code is needed across multiple components,
  it can be implemented as a plain function
  that is imported into each of the components
  and invoked from their event handling methods
- This is **simpler than using Vuex actions**

Modules

- The Vuex store can be split into multiple “modules”
- Each of these have their own state, getters, mutations, and actions
- Modules can be further divided into sub-modules
- But **using a single store is far easier**

vuex-easy

- Acts as a layer above VueX library
  - so the Vue devtools Vuex tab can still be used
- Makes it unnecessary to implement any mutations
- Based on battle-tested React libraries redux-easy and context-easy
  - https://www.npmjs.com/package/redux-easy
  - https://www.npmjs.com/package/context-easy best option for React
- In npm at https://www.npmjs.com/package/vuex-easy
  - README provides simple setup instructions