Lecture 6
Vuex

Amal Aboulhassan

Vuex

- Vuex is a state management pattern + library for Vue.js applications.
- It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion.

State management with and without Vuex

- Example: a simple counter application
- It is a self-contained app with the following parts:
  - The state:
    - the source of truth that drives our app; (The state of the counter which its current value in this case.
  - The view:
    - a declarative mapping of the state;
  - The actions:
    - the possible ways the state could change in reaction to user inputs from the view.

```
const Counter = {
  // state
  data () {
    return {
      count: 0
    }
  },
  // view
  template: `
    <div>{{ count }}</div>
  `,
  // actions
  methods: {
    increment () {
      this.count++
    }
  }
}

createApp(Counter).mount('#app')
```

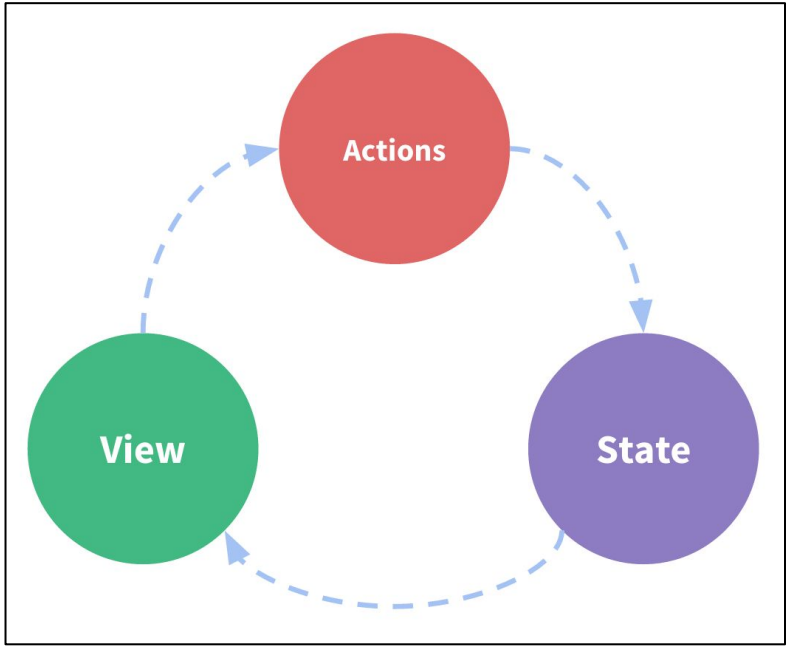
https://vuex.vuejs.org/

Note

- If we are creating a vue application through an IDE, we will need to create an instance of the application and mount it
- https://vuejs.org/guide/essentials/application

State management without Vuex

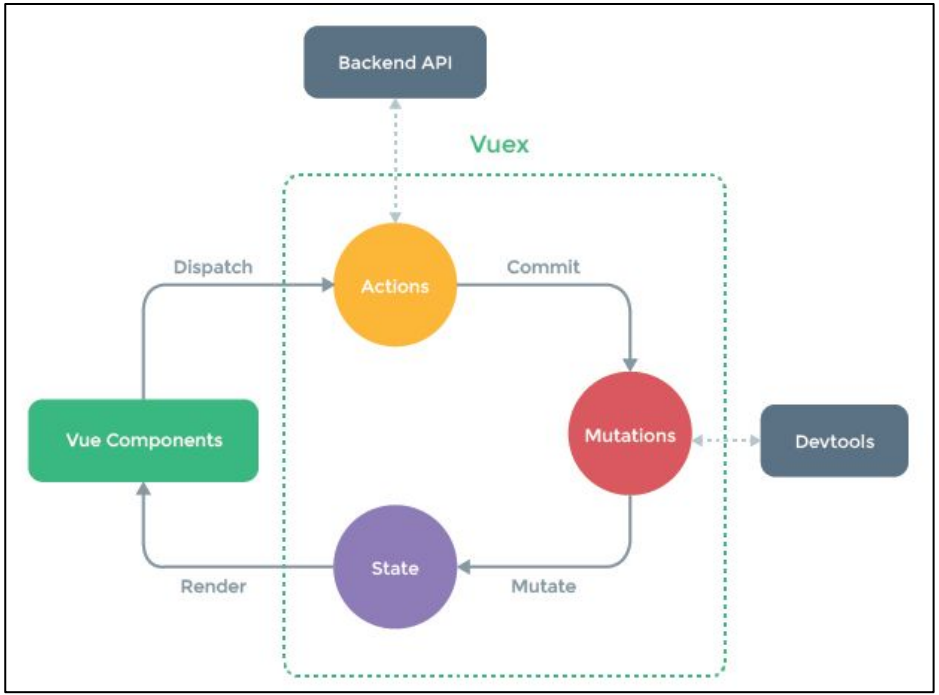
- In Vue, if we don’t use VueX, the states are managed as a one-way-data flow
- Some problems and complications:
  - Multiple components that share the same state
  - Multiple views may depend on the same piece of state
    - passing props can be tedious for deeply nested components.
    - doesn't work for sibling components.
  - Actions from different views may need to mutate the same piece of state is cumbersome with
    - reaching for direct parent/child instance references.
    - trying to mutate and synchronize multiple copies of the state via events.



https://vuex.vuejs.org/

State management **with** Vuex

- The shared state is extracted out of the components, and manage it in a global singleton.
- With this, the component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree.



https://vuex.vuejs.org/

Main Technical differences: Vue and Vuex

	Vue	Vue + VueX
At the center of every application	Vue instance	Vuex.Store
Data object	Vue instance has a reactive data property	Vuex store has a reactive state
Updates	Instance methods	Store actions
Data access	Instance Computed Properties + getters/setters	Store Getters
Mutations	None	Mutations: handles updating the State

https://medium.com/vue-mastery/vuex-explained-visually-f17c8c76d6c4

Example: Implementing the Counter Example

```
import { createApp } from 'vue'
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state () {
    return {
      count: 0
    }
  },
  mutations: {
    increment (state) {
      state.count++
    }
  }
})

const app = createApp({ /* your root component */ })

// Install the store instance as a plugin
app.use(store)
```

```
store.commit('increment')

console.log(store.state.count) // -> 1
```

Vuex Store

- At the center of every Vuex application is the store. A "store" is basically a container that holds your application state.
- There are two things that make a Vuex store different from a plain global object:
  - Vuex stores are reactive. When Vue components retrieve state from it, they will reactively and efficiently update if the store's state changes.
  - You cannot directly mutate the store's state. The only way to change a store's state is by explicitly committing mutations. This ensures every state change leaves a track-able record, and enables tooling that helps us better understand our applications.

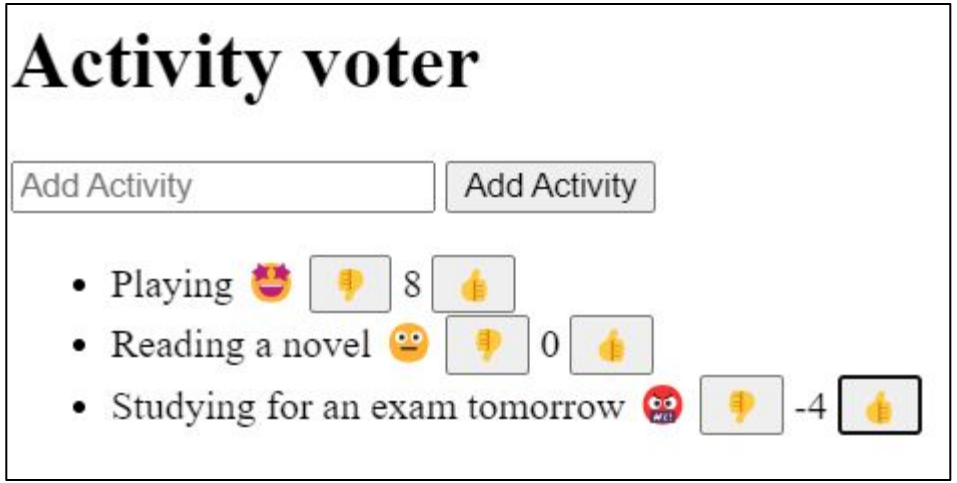
https://vuex.vuejs.org/guide/

Case Study

https://gist.github.com/perborgen/ce11d4f36cfb97f2ddb15ae73bfa10dd

A plan making application

- We will be Building a plan-making application.
- A user can type in activities and then vote how much they like/dislike them.



https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

App View: without data binding or states

```
<div id="app">
  <h1>Activity voter</h1>
  <form>
    <input type="text" placeholder="Add Activity" />
    <button id="button">Add Activity</button>
  </form>
  <ul>
    <li>
      <span>
Go Snowboarding</span>
<span>?</span>
        <button>?</button>
        5
        <button>?</button>
      </span>
    </li>
  </ul>

  </div>
```

We will start by hard-coded code for now

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Adding the store

- We will start by adding an empty store
- Notice that we added Vuex together with Vue in the script block
- Notice the difference between the store and the instance

```
<script>
  Vue.use(Vuex);

  const store = new Vuex.Store({


  });

  new Vue({
    el: "#app",
    store
  });

  </script>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Skeleton of a state

- These properties reflect the type of data the store holds:
  - state for state (global data),
  - mutations (commits that mutate data),
  - actions (dispatches that call mutations),
  - getters (store computed properties).

```
import { createStore } from 'vuex'

export default createStore({
state: {
      },
mutations: {

},
actions: {

},
getters: {

}
  })
```

https://www.digitalocean.com/community/tutorials/how-to-manage-state-in-a-vue-js-application-with-vuex

Adding the store

- We will fill the store with activities and emojis lists

```
<script>
  Vue.use(Vuex);

  const store = new Vuex.Store({
    state: {
      activities: [{ name: "go snowboarding", rating: 5 }],
      emojis: ["?"]
    }
  });

  new Vue({
    el: "#app",
    store
  });

  </script>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Adding the store

- We will fill the store with activities and emojis lists
- To allow our state to change reactively, we can use Vuex mapState to handle computed state properties. Notice that is added in the Vue constructor

```
<script>
  Vue.use(Vuex);

  const store = new Vuex.Store({
state: {

      activities: [{ name: "go snowboarding", rating: 5 }],
       emojis: ["?"]

  });

  new Vue({
    el: "#app",
    store,
    computed: Vuex.mapState(["activities", "emojis"])
});


  </script>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Note

- Reactivity
  - Reactivity is a programming paradigm that allows us to adjust to changes in a declarative manner. Component state consists of reactive JavaScript objects. When you modify them, the view updates.
  - Detailed explanation: https://vuejs.org/guide/extras/reactivity-in-depth.html
- Computed state properties
  - A computed property automatically tracks its reactive dependencies.
  - For example, simplifying the logic of a nested array
  - Detailed explanation: https://vuejs.org/guide/essentials/computed.html

Implementing a component

- Now we have activities inside our state. Let's render a separate component for each of those activities. Each one will need activity and emojis props.

```
<script>
Vue.component("activity-item", {
  props: ["activity", "emojis"],
  template: `
    <li>
      <span>{{ activity.name }}
        <span>{{ emojis[0] }}</span>
        <button>?</button>
        {{activity.rating}}
        <button>?</button>
      </span>
    </li>
    `
});
  Vue.use(Vuex);
  const store = new Vuex.Store({
state: {

      activities: [{ name: "go snowboarding", rating:
5 }],
       emojis: ["?"]

  });

  new Vue({
    el: "#app",
    store,
    computed: Vuex.mapState(["activities", "emojis"])
});
</script>
```

Note

- Components:
  - Vue implements its own component model that allows us to encapsulate custom content and logic in each component. Components allow us to split the UI into independent and reusable pieces, and think about each piece in isolation. It's common for an app to be organized into a tree of nested components
  - Detailed explanation: https://vuejs.org/guide/essentials/component-basics
- Props:
  - Props is a special keyword that stands for properties and is used for passing data from one component to another.
  - Detailed explanation of props in Vue: https://vuejs.org/guide/components/props.html

Back to the app view

- We will start by the temporary hard-coded code that we put in the start
- Now as the activities and emojis list are ready, we can just loop on them as they are added by the form and show them in the web page
- How can we loop on data binded from the script part?

```
<div id="app">
  <h1>Activity voter</h1>
  <form>
    <input type="text" placeholder="Add Activity" />
    <button id="button">Add Activity</button>
  </form>
  <ul>
    <activity-item
      v-for="item in activities"
          v-bind:activity="item"
          v-bind:emojis="emojis"
          v-bind:key="item.name">
    </activity-item>

</ul>



  </div>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Updating the store in Vuex

- We need to change the counter state by incrementing and decrementing the values
- How can we do this?
- Can we do it directly?

```
<script>
  Vue.use(Vuex);

  const store = new Vuex.Store({
      state: {
activities: [{ name: "go snowboarding", rating: 5 }],
      emojis: ["?"]

  });

  new Vue({
    el: "#app",
    store,
    computed: Vuex.mapState(["activities", "emojis"])
});


  </script>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Updating the store in Vuex

- We will use mutations
- We have added two functions for the incrementation and decrementation. The function do not include the logic yet

```
<script>
  Vue.use(Vuex);
  const store = new Vuex.Store({
  state: {
    activities: [
      { name: "go snowboarding", rating: 5 },
    ],
    emojis: ["?"]
  },
  mutations: {
    increment(state, activityName) {
      console.log('increment')
    },
    decrement(state, activityName) {
      console.log('decrement'); },  }});
```

```
new Vue({
  el: "#app",
  store,
  data() {
    return {
      activityName: ""
    };
  },
  computed: Vuex.mapState(["activities", "emojis"]),
  methods: {
    increment(activityName) {
      this.$store.commit("increment",
activityName);
    },
    decrement(activityName) {
      this.$store.commit("decrement",
activityName);
    }}});</script>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Note

- Commit:
  - The only way to actually change state in a Vuex store is by committing a mutation.
  - Detailed explanation: https://vuex.vuejs.org/guide/mutations.html#using-constants-for-mutation-types

Back to the app view

- As they are ready now, we will add them to the activities

```
<div id="app">
  <h1>Activity voter</h1>
  <form>
    <input type="text" placeholder="Add Activity" />
    <button id="button">Add Activity</button>
  </form>
  <ul>
    <activity-item
      v-for="item in activities"
            v-bind:increment="increment"
            v-bind:decrement="decrement"
             v-bind:activity="item"
             v-bind:emojis="emojis"
             v-bind:key="item.name">
    </activity-item>

</ul>



  </div>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Update the component

- Now we have activities inside our state. We can render a separate component for each of those activities. Each one will need activity and emojis props.

```
<script>
Vue.component("activity-item", {
  props: ["activity", "emojis"],
  template: `
    <li>
      <span>{{ activity.name }}
          <span>{{ emojis[0] }}</span>
          <button
@click="decrement(activity.name)">?</button>
          {{activity.rating}}
          <button
@click="increment(activity.name)">?</button>
      </span>
    </li>   `
});
  Vue.use(Vuex);
  const store = new Vuex.Store({
state: {

     activities: [{ name: "go snowboarding", rating: 5
}],
      emojis: ["?"]

  });
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Implement the counter

- First, we need to find an activity by its name, and then update its rating.

```
<script>
  Vue.use(Vuex);

  const store = new Vuex.Store({
  state: {
    activities: [
      { name: "go snowboarding", rating: 5 },
    ],
    emojis: ["?"]
  },
  mutations: {
    increment(state, activityName) {
      state.activities
        .filter(activity => activity.name ===
`${activityName}`)
        .map(activity => activity.rating++);

    },
    decrement(state, activityName) {
      state.activities
        .filter(activity => activity.name ===
`${activityName}`)
        .map(activity => activity.rating--);
},}});
…
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Add activities

- We will implement the form to be able to add activities
- How do you think we can update the activities?

```
<script>
  Vue.use(Vuex);

  const store = new Vuex.Store({
  state: {
    activities: [
      { name: "go snowboarding", rating: 5 },
    ],
    emojis: ["?"]
  },
  mutations: {
    increment(state, activityName) {
      state.activities
        .filter(activity => activity.name ===
`${activityName}`)
        .map(activity => activity.rating++);

    },
    decrement(state, activityName) {
      state.activities
        .filter(activity => activity.name ===
`${activityName}`)
        .map(activity => activity.rating--);
},}});
 …
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Add activities

- We will implement the form to be able to add activities
- How do you think we can update the activities?
  - We will add a mutation

```
<script>
  Vue.use(Vuex);

  const store = new Vuex.Store({
  state: {
    activities: [
      { name: "go snowboarding", rating: 5 },
    ],
    emojis: ["?"]
  },
  mutations: {
    increment(state, activityName) {
      state.activities
        .filter(activity => activity.name ===
`${activityName}`)
        .map(activity => activity.rating++);

    },
    decrement(state, activityName) {
      state.activities
        .filter(activity => activity.name ===
`${activityName}`)
        .map(activity => activity.rating--);
addActivity(state, name) {
      state.activities.push({ name, rating: 0 });
},}});
…
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Add activities

- We will implement the form to be able to add activities
- How do you think we can update the activities?
  - We will add a mutation
- Don't forget to commit too

```
…
    new Vue({
  el: "#app",
  store,
  data() {
    return {
      activityName: ""
    };
  },
  computed: Vuex.mapState(["activities", "emojis"]),
  methods: {
    increment(activityName) {
      this.$store.commit("increment", activityName);
    },
    decrement(activityName) {
      this.$store.commit("decrement", activityName);
addActivity(activityName)
{this.$store.commit("addActivity",
activityName);}}});</script>
```

https://www.freecodecamp.org/news/learn-vuex-in-5-minutes/

Implementing the form submission

- The HTML of the form

```
<form @submit="onSubmit">
  <input type="text" placeholder="Add Activity"
v-model="activityName" />
  <button id="button">Add Activity</button>

  </form>
```

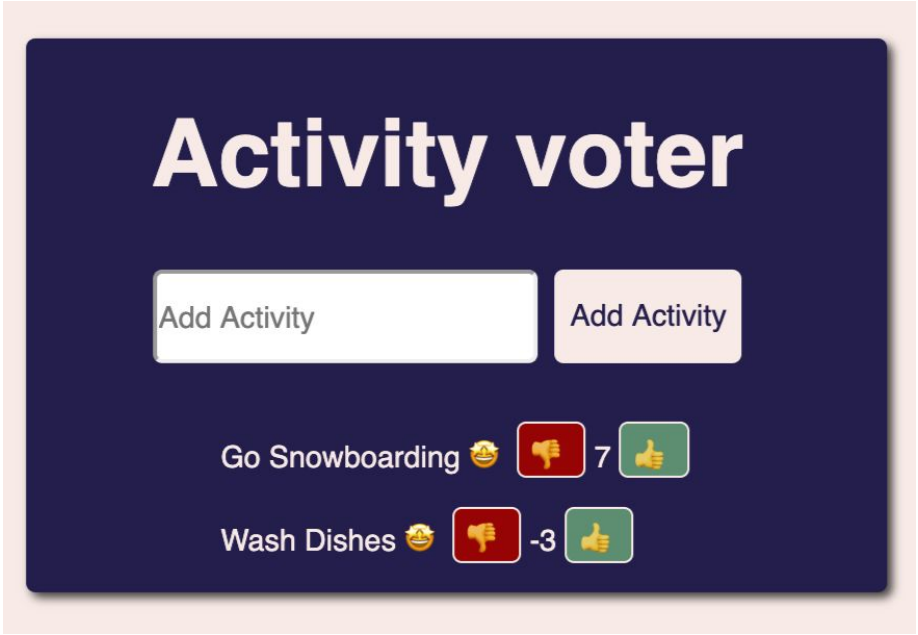
Implementing the form submission

- As everything is now ready, we will call the the relevant functions in the methods block
- We call e.preventDefault() to avoid the form from reloading on each addition of a new activity.
- What is “onSubmit”?

```
methods: {
    ...
    onSubmit(e) {
      e.preventDefault();
      this.addActivity(this.activityName);
      this.activityName = "";
    }

    }
```

Implementing the form submission

- We don’t want to have one emotion for every score, it does not give a good user experience
- So we will now implement the emojis to reflect different feelings



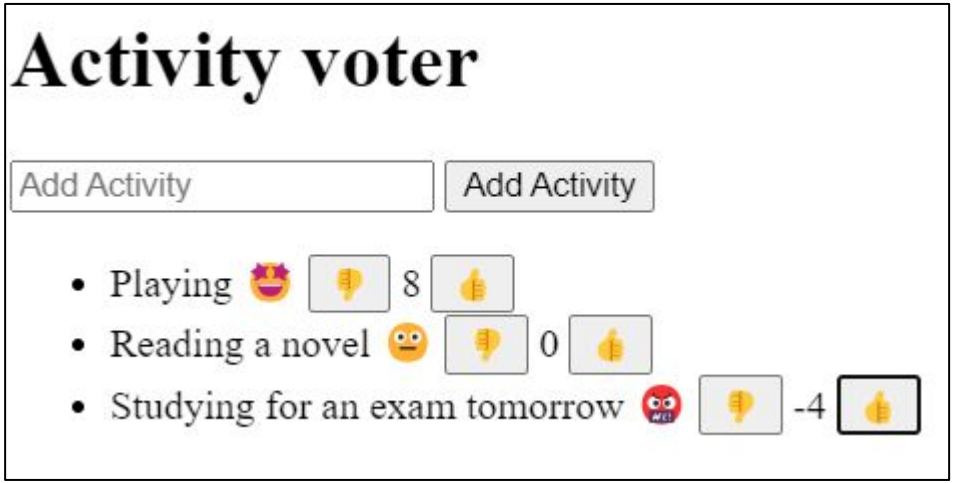
Implementing the form submission

```
state: {
    activities: [],
    emojis: { yay: "?", nice: "?",
meh: "?", argh: "?", hateIt: "?"}
},

  ...
```

```
Vue.component("activity-item", {
  props: ["activity", "emojis", "increment", "decrement"],
  template: `
    <li>
      <span>{{ activity.name }}
        <span v-if="activity.rating <= -5">{{ emojis.hateIt }}</span>
        <span v-else-if="activity.rating <= -3">{{ emojis.argh
}}</span>
        <span v-else-if="activity.rating < 3">{{ emojis.meh }}</span>
        <span v-else-if="activity.rating < 5">{{ emojis.nice }}</span>
        <span v-else>{{ emojis.yay }}</span>
        <button @click="decrement(activity.name)">?</button>
          {{activity.rating}}
        <button @click="increment(activity.name)">?</button>
      </span>
    </li>
    `


  });
```

Final Results

- We start with an empty form
- As new activities are added, new records appear



Thank you!