Vuex

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.
- The basic concept is derived from React’s Redux and Flux library.
- Vuex comes into the picture when our client side application becomes more and more complex.

Why Vuex?

- Imagine you've developed a multi-user chat app. The interface has a user list, private chat windows, an inbox with chat history, and a notification bar to inform users of unread messages from other users they aren't currently viewing.
- Millions of users are chatting to millions of other users through your app daily. However, there are complaints about an annoying problem: the notification bar will occasionally give false notifications. A user will be notified of a new unread message, but when they check to see what it is it's just a message they've already seen.
- Example of real-time scenario that the Facebook developers had with their chat system a few years back. The process of solving this inspired their developers to create an application architecture they named "Flux". Flux forms the basis of Vuex, Redux, and other similar libraries.

- Breaking down larger pieces of an app into smaller components has become really popular thanks to the rise of front-end libraries like React and Vue.
- Components allow you to easily reuse code, reduce architectural complexity, and build impressive user interfaces by logically dividing your code into independent chunks.
- However, despite these advantages, one issue that eventually hits any application that grows large enough is the difficulty of managing its state.
- Facebook's answer to this challenge was Flux, a pattern that later served as the basis for the more popular Redux state management library that is commonly used with React.
- The Vue team, realizing that their library would also need a way to manage state, introduced Vuex, the official state management library for Vue.

What is a "State Management Pattern"?

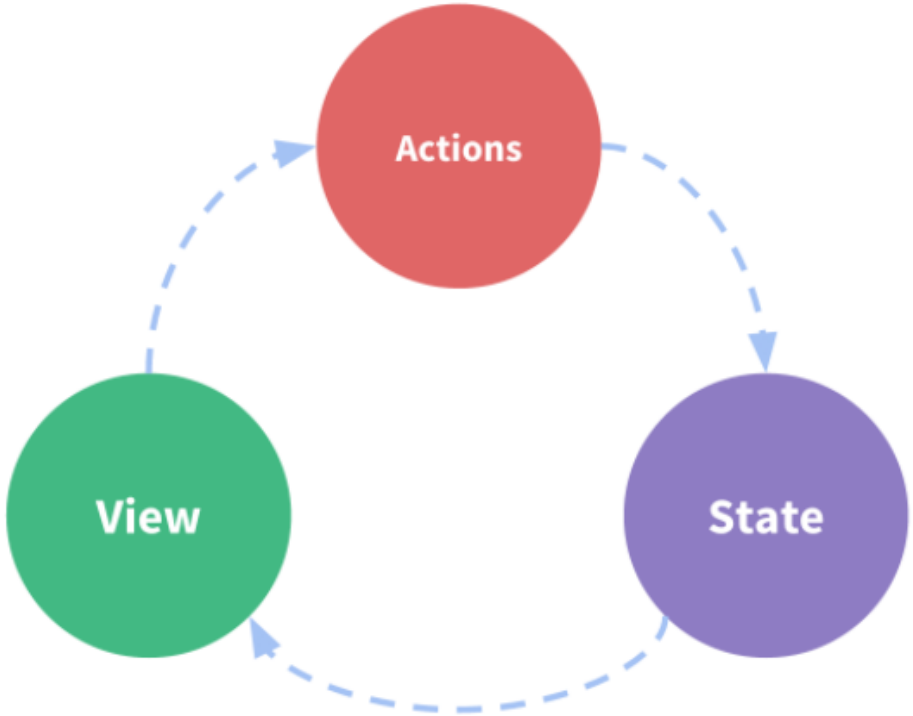
➢ It is a self-contained app with the following parts:

- The **state**, the source of truth that drives our app;
- 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**.

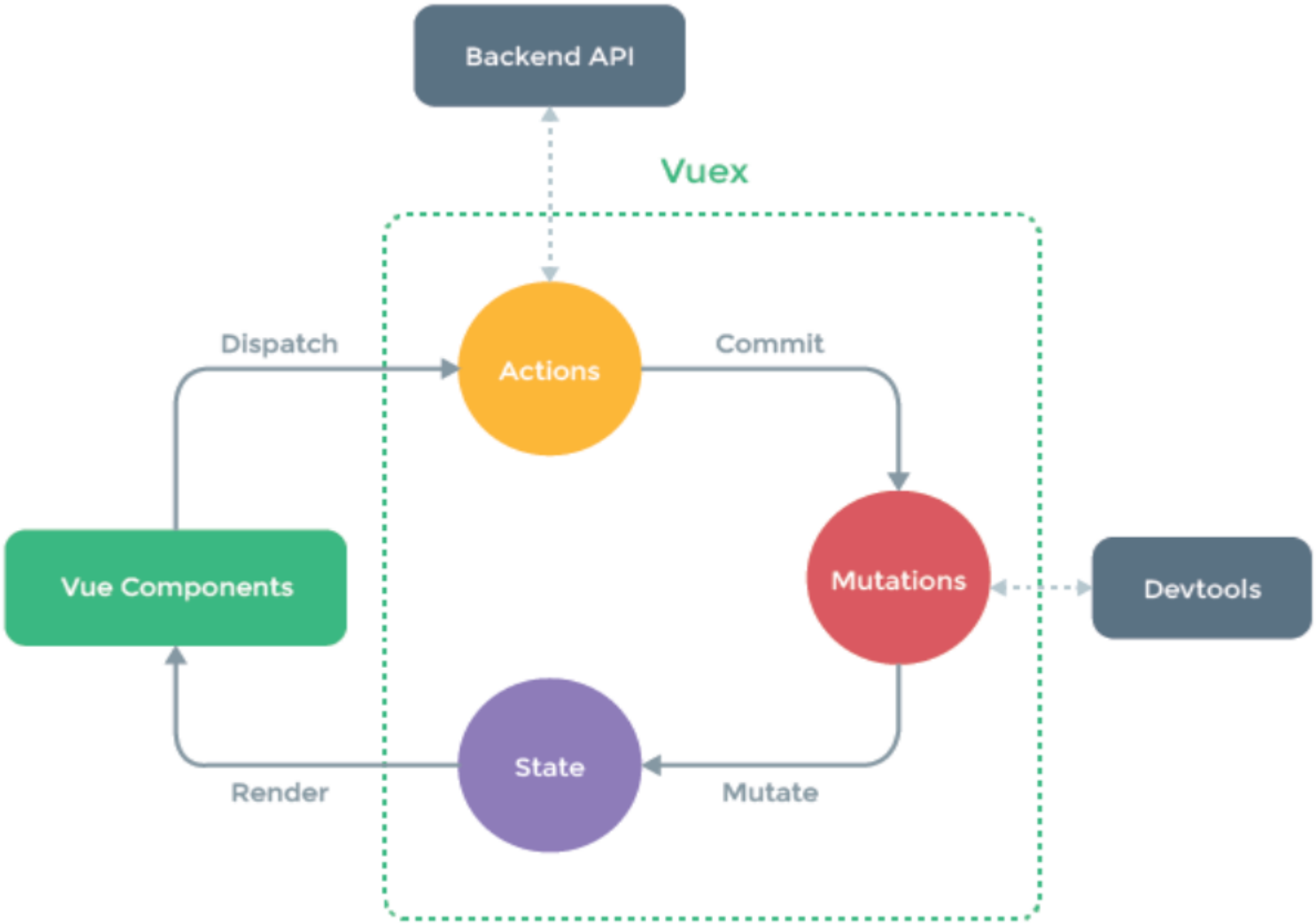
Simple Vue Counter Example

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

This is a simple representation of the concept of "one-way data flow"



- The simplicity quickly breaks down when we have **multiple components that share a common state**:
  - Multiple views may depend on the same piece of state.
  - Actions from different views may need to mutate the same piece of state.
- Problem one, passing props can be tedious for deeply nested components, and simply doesn't work for sibling components.
- Problem two, we often find ourselves resorting to solutions such as reaching for direct parent/child instance references or trying to mutate and synchronize multiple copies of the state via events.
- Both of these patterns are brittle and quickly lead to unmaintainable code.


Backend API
Vuex
Dispatch
Actions
Commit
Vue Components
Mutations
Devtools
State
Render
Mutate

- **State:** If you are using Vuex, then you will have only one Single store for each VueJS powered application. Vuex Store is Singleton source of State. Our store contains application state. State management in VueJS with Vuex is straightforward. You just need to modify the state via dispatching the action and not directly because we want to predict the future states. Our store contains application state.
- **Mutations:** The only way to change the state in a Vuex store is by committing a mutation. We can directly change the state, but we will not do it because we need a snap shot for every step of our project. We need to predict the next state. For the debugging purpose, we will not mutate the state directly, but via mutations.
- **Actions:** Actions are similar to mutations, the differences being that:
    - Instead of mutating the state, actions commit mutations.
    - Actions can contain arbitrary asynchronous operations.