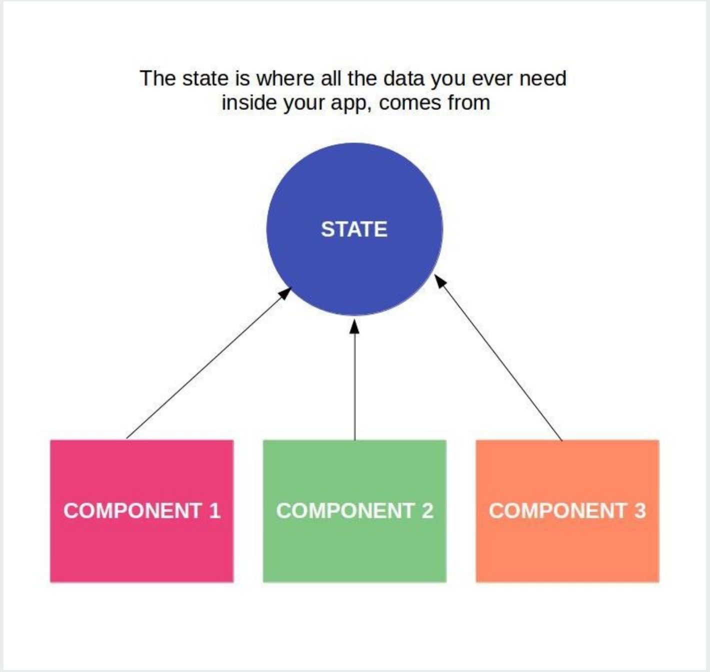
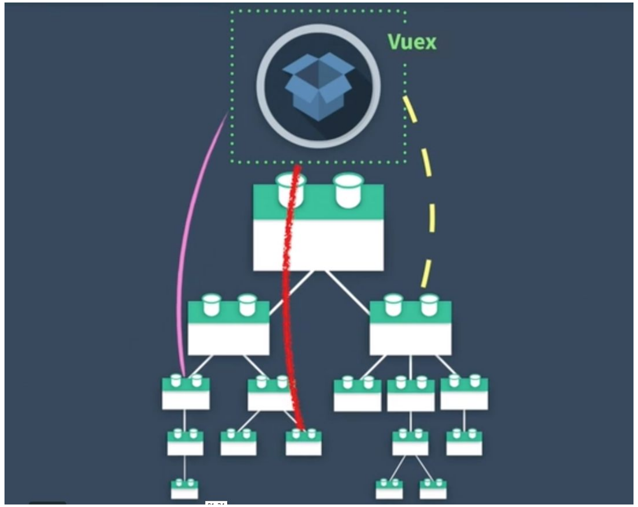
Vue State Management With VueX

A document about agile processes and agile theory


The state is where all the data you ever need inside your app, comes from
STATE
COMPONENT 1
COMPONENT 2
COMPONENT 3

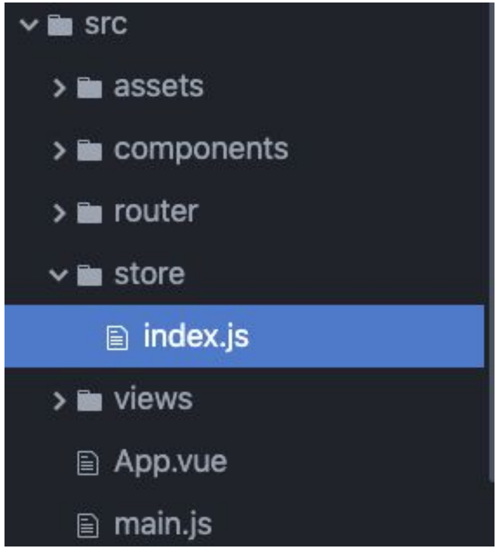
What is state management

State allows for communication between components. State properties can affect single or multiple components using a central 'store' and communication methods (getters, mutations and actions).



State within the Vue cli

When using the vue cli to establish a project, select store/vuex as an option. This will create a 'store' folder within the project. This is where you can set and reference key store properties and methods.



Vue State - 4 Main concepts

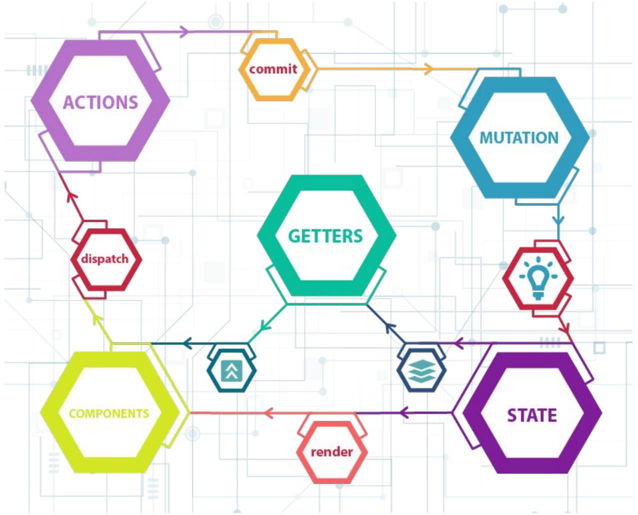
Working with state in a literal and simple way requires 4 concepts:

- Setting **properties** within state

- Creating **getters** to access the properties

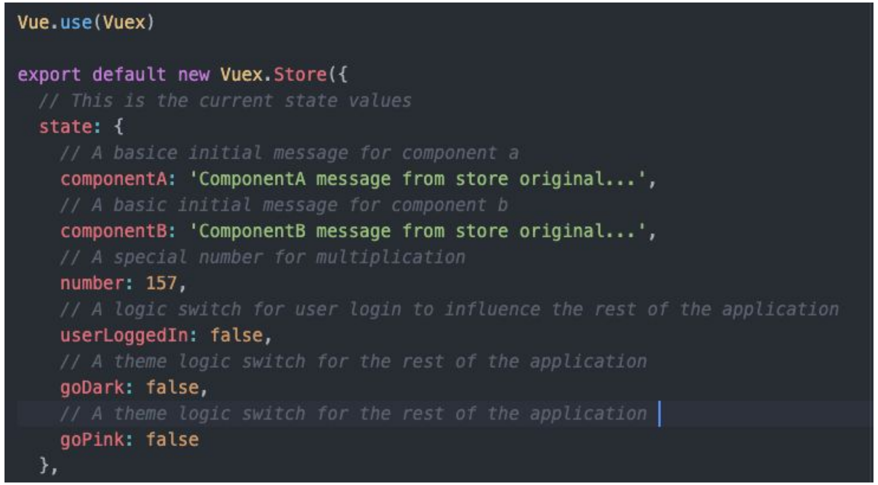
- Creating **mutations** to update and adjust the properties

- Using **watch** to trigger reactivity/changes through the components



1. Setting properties in state

Really simple, just like an object, state has properties with values. You can initialize these in your store file. **state.properties** are good for referencing what state values you have and what they are for.



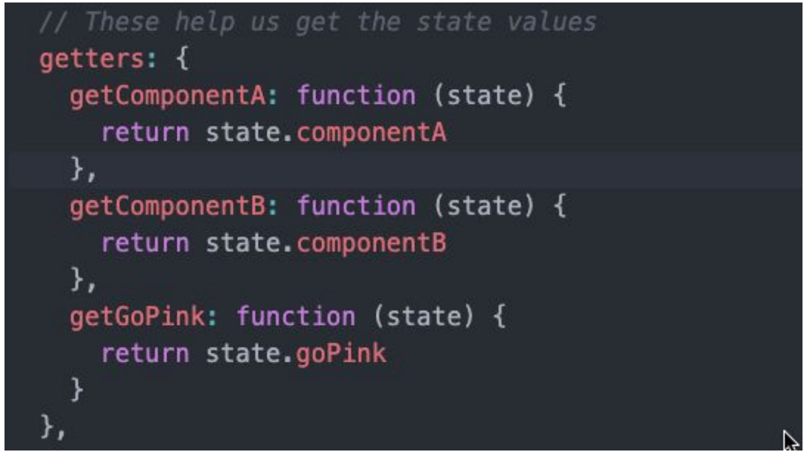
```
Vue.use(Vuex)

export default new Vuex.Store({
  // This is the current state values
  state: {
    // A basice initial message for component a
    componentA: 'ComponentA message from store original...',
    // A basic initial message for component b
    componentB: 'ComponentB message from store original...',
    // A special number for multiplication
    number: 157,
    // A logic switch for user login to influence the rest of the application
    userLoggedIn: false,
    // A theme logic switch for the rest of the application
    goDark: false,
    // A theme logic switch for the rest of the application
    goPink: false
  },
```

2. Creating getters in state

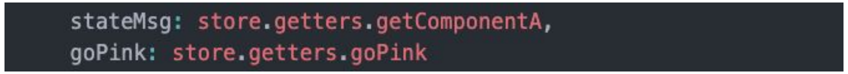
Getters do what they say. A pattern for retrieving values from state/store. Getters can be used from within component javascript.

Setting the getters in store.js



```
// These help us get the state values
getters: {
  getComponentA: function (state) {
    return state.componentA
  },
  getComponentB: function (state) {
    return state.componentB
  },
  getGoPink: function (state) {
    return state.goPink
  }
},
```

Referencing/using the getters within a component

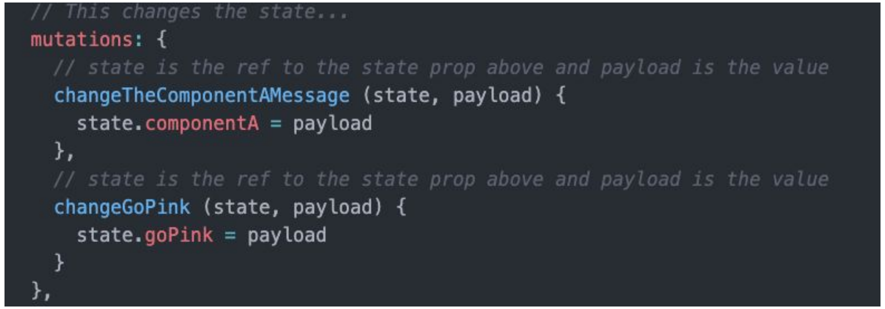


```
    stateMsg: store.getters.getComponentA,
    goPink: store.getters.goPink
```

3. Creating mutations within state

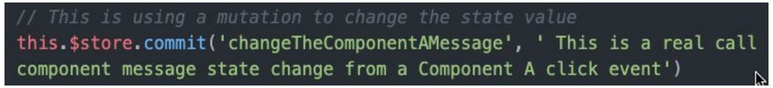
Mutations allow us to change and mutate state from within components. By using a simple pattern within store.js we can create multiple mutation methods

Setting the mutations in store.js



```
// This changes the state...
mutations: {
  // state is the ref to the state prop above and payload is the value
  changeTheComponentAMessage (state, payload) {
    state.componentA = payload
  },
  // state is the ref to the state prop above and payload is the value
  changeGoPink (state, payload) {
    state.goPink = payload
  }
},
```

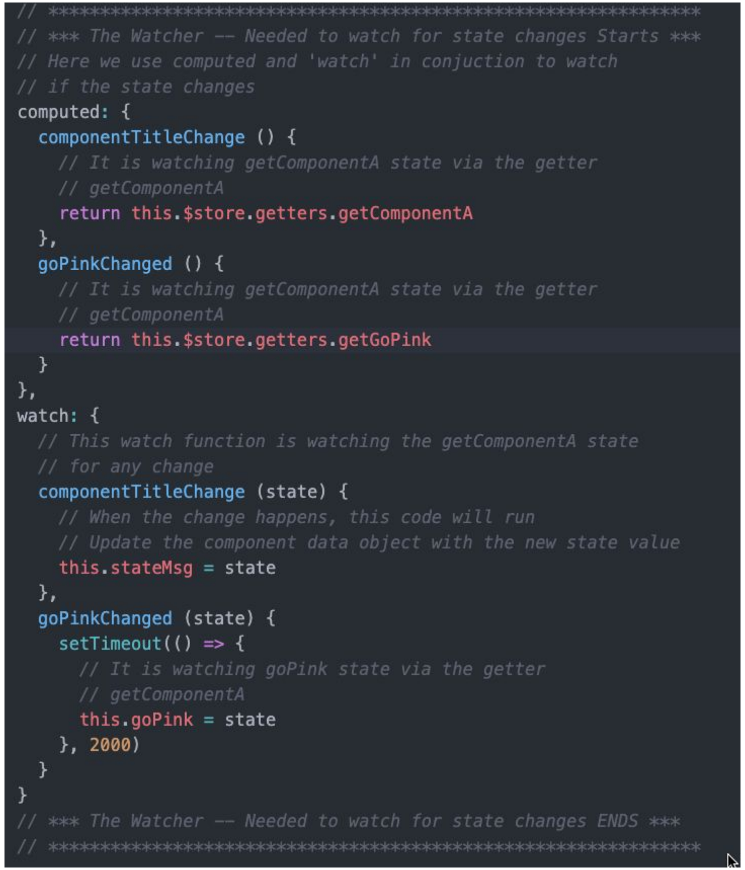
Referencing/using the mutations within a component



```
// This is using a mutation to change the state value
this.$store.commit('changeTheComponentAMessage', ' This is a real call
component message state change from a Component A click event')
```

4. Setting watchers to look for state changes

Watchers or a ‘watch’ method, allows us to make the component react to any state changes. If you want state to apply to a component, set a watcher accordingly. Watchers require a computed function and watch function in combination, placed within the .js of your component.



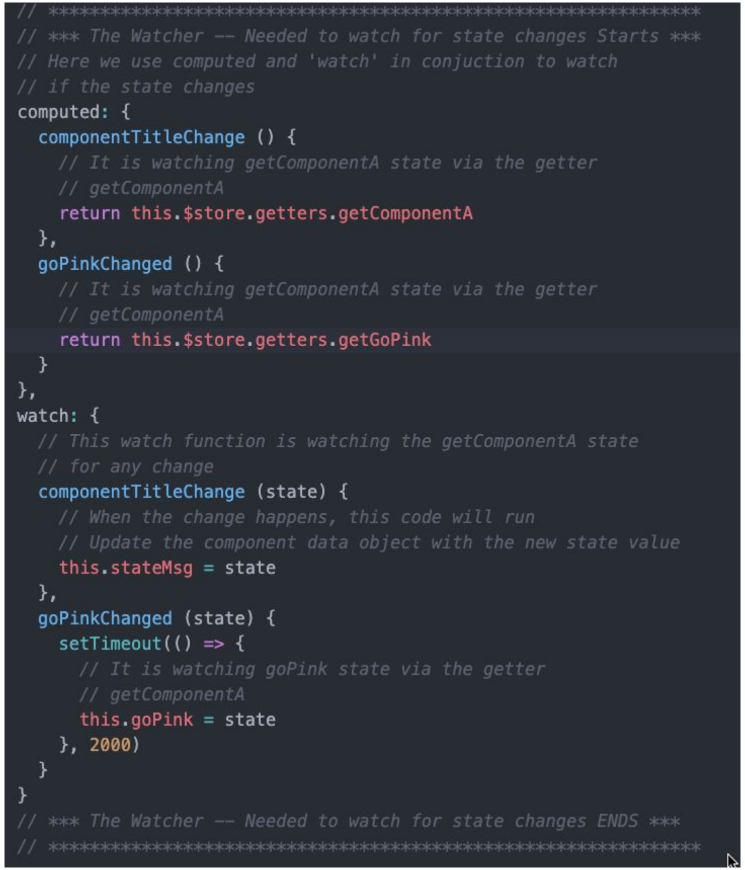
```
// ***************************************************************
// *** The Watcher -- Needed to watch for state changes Starts ***
// Here we use computed and 'watch' in conjuction to watch
// if the state changes
computed: {
  componentTitleChange () {
    // It is watching getComponentA state via the getter
    // getComponentA
    return this.$store.getters.getComponentA
  },
  goPinkChanged () {
    // It is watching getComponentA state via the getter
    // getComponentA
    return this.$store.getters.getGoPink
  }
},
watch: {
  // This watch function is watching the getComponentA state
  // for any change
  componentTitleChange (state) {
    // When the change happens, this code will run
    // Update the component data object with the new state value
    this.stateMsg = state
  },
  goPinkChanged (state) {
    setTimeout(() => {
      // It is watching goPink state via the getter
      // getComponentA
      this.goPink = state
    }, 2000)
  }
}
// *** The Watcher -- Needed to watch for state changes ENDS ***
// ***************************************************************
```

The watchers above will update the component’s data

For more info...

To see a working demo of state in action, reference the following repository:

https://github.com/veratechnz/state-demo



```
// ************************************************************
// *** The Watcher -- Needed to watch for state changes Starts ***
// Here we use computed and 'watch' in conjuction to watch
// if the state changes
computed: {
  componentTitleChange () {
    // It is watching getComponentA state via the getter
    // getComponentA
    return this.$store.getters.getComponentA
  },
  goPinkChanged () {
    // It is watching getComponentA state via the getter
    // getComponentA
    return this.$store.getters.getGoPink
  }
},
watch: {
  // This watch function is watching the getComponentA state
  // for any change
  componentTitleChange (state) {
    // When the change happens, this code will run
    // Update the component data object with the new state value
    this.stateMsg = state
  },
  goPinkChanged (state) {
    setTimeout(() => {
      // It is watching goPink state via the getter
      // getComponentA
      this.goPink = state
    }, 2000)
  }
}
// *** The Watcher -- Needed to watch for state changes ENDS ***
// ************************************************************
```

The watchers above will update the component's data