ADVANCED VUE JS WORKSHOP_II_

ABOUT ME

@Rafael_Casuso

- **CTO @StayApp**
- **CEO @SnowStormIO**
- **Organizer @VueJsMadrid, @BotDevMad**
- **Software Engineer** with +10 years of experience leading teams and developing.
- **Software Architect** looking for revolutionary ways to change the world.
- **Specialties**: JavaScript, NodeJS, Conversational Intelligences.

+ ADVANCED

ROUTING

BASIC SETUP_

- VUE-ROUTER IS THE OFFICIAL CORE PLUGIN
- IT WORKS SEAMLESSLY WITH WEEX AND NATIVESCRIPT

```html
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="app">
  <h1>Hello App!</h1>
  <p>
    <!-- use router-link component for navigation. -->
    <!-- specify the link by passing the `to` prop. -->
    <!-- `<router-link>` will be rendered as an `<a>` tag by default -->
    <router-link to="/foo">Go to Foo</router-link>
    <router-link to="/bar">Go to Bar</router-link>
  </p>
  <!-- route outlet -->
  <!-- component matched by the route will render here -->
  <router-view></router-view>
</div>
```

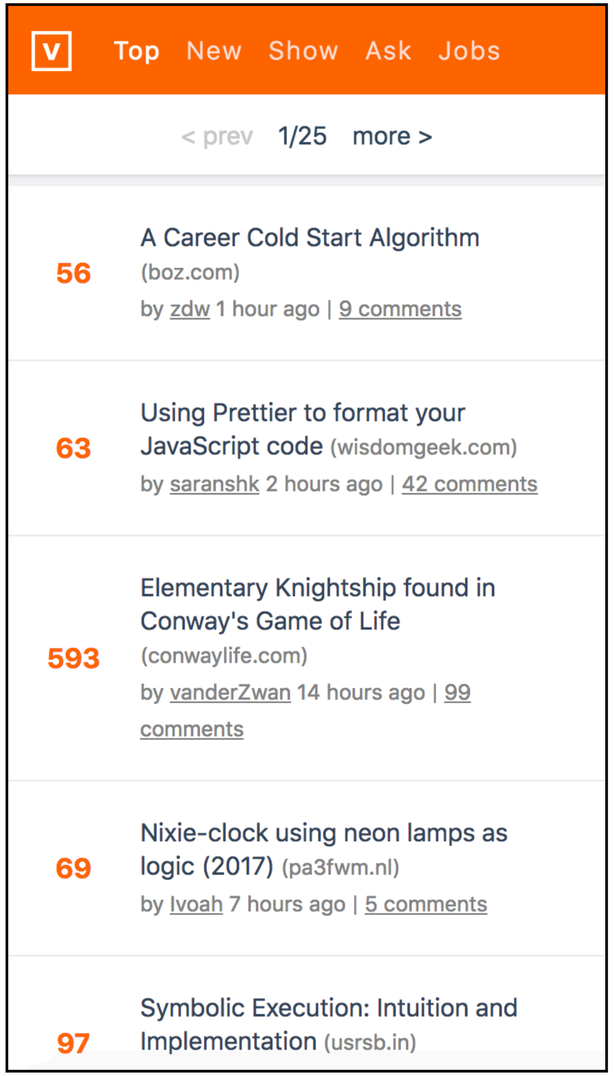
```js
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. Define some routes
// Each route should map to a component. The "component" can
// either be an actual component constructor created via
// `Vue.extend()`, or just a component options object.
// We'll talk about nested routes later.
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]

// 3. Create the router instance and pass the `routes` option
// You can pass in additional options here, but let's
// keep it simple for now.
const router = new VueRouter({
  routes // short for `routes: routes`
```

LEARN BY EXAMPLE_

- Vue Hackernews:
  - git clone https://github.com/vuejs/vue-hackernews-2.0
  - npm i
  - npm run dev



FUNDAMENTALS_

- ROUTER-LINK IS THE BUILT IN COMPONENT FOR LINKS
  - IT AUTOMATICALLY RECEIVES CLASS “ROUTER-LINK-ACTIVE”
- ROUTER-VIEW IS THE BUILT-IN COMPONENT AS PLACEHOLDER
- ROUTER INSTANCE IS INJECTED IN ALL COMPONENTS AS this.$router

```
computed: {
  username () {
    // We will see what `params` is shortly
    return this.$route.params.username
  }
},
methods: {
  goBack () {
    window.history.length > 1
      ? this.$router.go(-1)
      : this.$router.push('/')
```

DYNAMIC ROUTES, PARAMS & QUERIES_

- DYNAMIC ROUTE SECTIONS ARE DEFINED BY :PATH
- THOSE DYNAMIC VALUES ARE AVAILABLE AT this.$route.params
- QUERY PARAMS ARE AVAILABLE AT this.$route.query
- DYNAMIC PATHS ACCEPT REGEXP ADVANCED PATTERNS LIKE :page(\\d+)
- OPTIONAL DYNAMIC PATHS WITH ?

```
const User = {
  template: '<div>User</div>'
}

const router = new VueRouter({
  routes: [
    // los segmentos dinámicos comienzan con dos puntos
    { path: '/user/:id', component: User }
  ]
})
```

NESTED ROUTES_

- **NESTED ROUTES ALLOW COMPONENTS TO BE RENDERED INTO ROOT LEVEL COMPONENTS**

```
const User = {
  template: `
    <div class="user">
      <h2>User {{ $route.params.id }}</h2>
      <router-view></router-view>
    </div>
  `
}
```

```
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User,
      children: [
        {
          // UserProfile will be rendered inside User's <router-view>
          // when /user/:id/profile is matched
          path: 'profile',
          component: UserProfile
        },
        {
          // UserPosts will be rendered inside User's <router-view>
          // when /user/:id/posts is matched
          path: 'posts',
          component: UserPosts
        }
      ]
    }
  ]
})
```

PROGRAMMATIC NAVIGATION_

- Via $router.push(location, onComplete?, onAbort?)
- Without new history entry with $router.replace(location, onComplete?, onAbort?)
- Going n steps forward or backwards with $router.go(n)

```
// literal string path
router.push('home')

// object
router.push({ path: 'home' })

// named route
router.push({ name: 'user', params: { userId: 123 }})

// with query, resulting in /register?plan=private
router.push({ path: 'register', query: { plan: 'private' }})
```

NAMED ROUTES & VIEWS_

- NAMED ROUTES ARE MORE SEMANTIC AND ALLOW DIFFERENT USE
- NAMED VIEWS ALLOW ADVANCED LAYOUTS WITH COMPONENTS PER ROUTE

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
      name: 'user',
      component: User
    }
  ]
})
```

```
<router-view class="view one"></router-view>
<router-view class="view two" name="a"></router-view>
<router-view class="view three" name="b"></router-view>
```

```
<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>
```

```
const router = new VueRouter({
  routes: [
    {
      path: '/',
      components: {
        default: Foo,
        a: Bar,
        b: Baz
      }
    }
  ]
})
```

PASSING PROPS TO ROUTE COMPONENTS_

- **INSTEAD OF $ROUTE.PARAMS WE CAN PASS THIS PARAMS AS PROPS FOR THE COMPONENT**

```
const User = {
  props: ['id'],
  template: '<div>User {{ id }}</div>'
}
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User, props: true },

    // for routes with named views, you have to define the `props` option
    {
      path: '/user/:id',
      components: { default: User, sidebar: Sidebar },
      props: { default: true, sidebar: false }
    }
  ]
})
```

NAVIGATION GUARDS_

- GUARDS CONTROL ACCESS TO ROUTES, REDIRECTING OR CANCELLING IT
- CAN BE GLOBAL, PER-ROUTE OR IN-COMPONENT
- RECEIVE FROM, TO AND NEXT:
  - PASS: NEXT(),
  - CANCEL: NEXT(FALSE)
  - REDIRECT: NEXT('/')

```
const router = new VueRouter({ ... })

router.beforeEach((to, from, next) => {
  // ...
})
```

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

NAVIGATION GUARDS_

```
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // called before the route that renders this component is confirmed.
    // does NOT have access to `this` component instance,
    // because it has not been created yet when this guard is called!
  },
  beforeRouteUpdate (to, from, next) {
    // called when the route that renders this component has changed,
    // but this component is reused in the new route.
    // For example, for a route with dynamic params `/foo/:id`, when we
    // navigate between `/foo/1` and `/foo/2`, the same `Foo` component instance
    // will be reused, and this hook will be called when that happens.
    // has access to `this` component instance.
  },
  beforeRouteLeave (to, from, next) {
    // called when the route that renders this component is about to
    // be navigated away from.
    // has access to `this` component instance.
```

LAZY LOADING_

- ASYNC COMPONENTS DEFINITION WHICH IS AUTOMATICALLY CREATED WITH VUE-LOADER SINGLE FILE COMPONENTS
- DYNAMIC IMPORT ALLOWED BY WEBPACK 2+
- CHUNK-BASED CODE-SPLITTING ALLOW A DIFFERENT CHUNK FOR EACH COMPONENT THAT IS LOADED WHEN ITS ROUTE IS MATCHED

```
const Foo = () => import('./Foo.vue')
```

+ ADVANCED

STATE MANAGEMENT

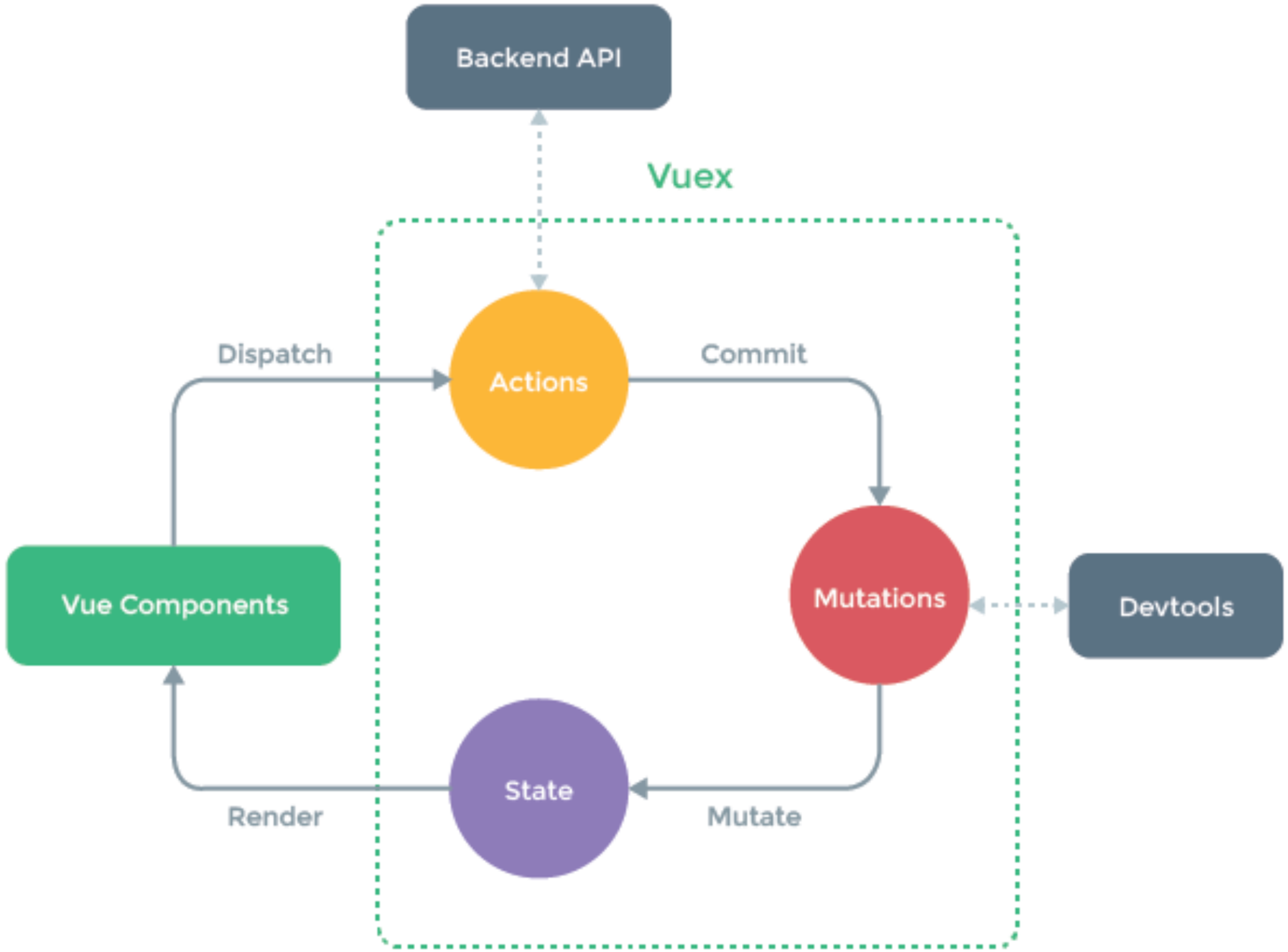
FUNDAMENTALS_

- VUEX IS THE ELM-INSPIRED OFFICIAL FLUX-LIKE IMPLEMENTATION
- VUEX CORE IS THE STORE, A CONTAINER OF APPLICATION STATE
- STORE IS THE SOURCE OF TRUTH

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  }
})
```

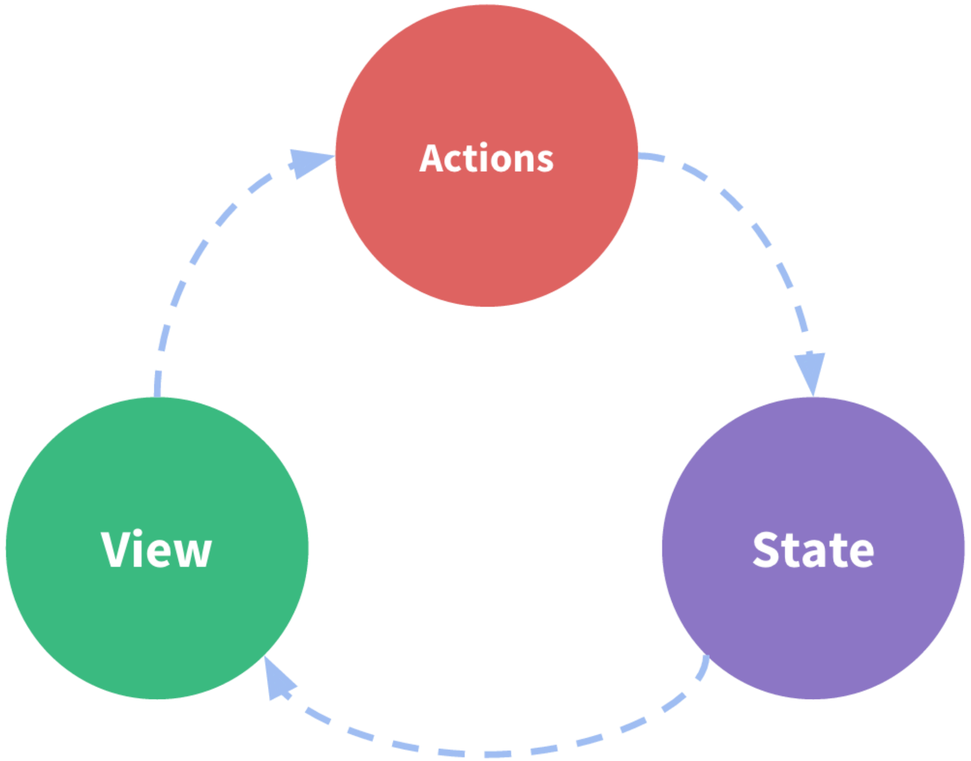
```
const app = new Vue({
  el: '#app',
  // provide the store using the "store" option.
  // this will inject the store instance to all child components.
  store,
  components: { Counter },
  template: `
    <div class="app">
      <counter></counter>
    </div>
  `
})
```

DIAGRAM_


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

WHY_

- WHEN WE HAVE A COMPLEX APPLICATION
- TOO COMPLEX TO PASS PROPS BETWEEN A LARGE HIERARCHY
- SPECIALLY COMPLEX TO PASS DATA BETWEEN SIBLINGS HORIZONTALLY
- WE NEED A DECOUPLED WAY TO SHARE DATA AT AN APPLICATION LEVEL BETWEEN COMPONENTS



STATE_

- SINGLE STATE TREE, SINGLE OBJECT WHICH IS THE SOURCE OF TRUTH
- AVAILABLE AT this.$store.state
- CAN BE MAPPED AS COMPUTED PROPERTIES with ...MapState

```
import { mapState } from 'vuex'

export default {
  // ...
  computed: mapState({
    // arrow functions can make the code very succinct!
    count: state => state.count,

    // passing the string value 'count' is same as `state => state.count`
    countAlias: 'count',

    // to access local state with `this`, a normal function must be used
    countPlusLocalState (state) {
      return state.count + this.localCount
```

GETTERS_

- COMPUTED DERIVED STATE FROM STATE PROPERTIES
- DECOUPLED COMPUTED PROPERTIES
- RECEIVE OTHER GETTERS AS SECOND ARGUMENT

```
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: '...', done: true },
      { id: 2, text: '...', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
})
```

```
import { mapGetters } from 'vuex'

export default {
  // ...
  computed: {
    // mix the getters into computed with object spread operator
    ...mapGetters([
      'doneTodosCount',
      'anotherGetter',
      // ...
```

```
getters: {
  // ...
  doneTodosCount: (state, getters) => {
    return getters.doneTodos.length
  }
}
```

MUTATIONS_

- THE ONLY FUNCTIONS THAT CAN CHANGE THE STATE TREE
- THEY ARE NOT INVOKED DIRECTLY BUT THROUGH A COMMIT, AN EVENT-LIKE API THAT ALLOWS TRACKING AND LOGGING
- COMMIT ALLOWS PAYLOAD, WHICH IS RECEIVED BY MUTATION HANDLER
- ALL INVOLVED COMPONENTS ARE UPDATED WHEN THE STATE IS MUTATED
- FOLLOWING REACTIVITY PATTERN STATE PROPERTIES MUST BE DECLARED UPFRONT
- MUTATION TYPES ARE RECOMMENDED TO BE DEFINED AS CONSTANTS
- MUTATIONS MUST BE SYNCHRONOUS

MUTATIONS_

```
const store = new Vuex.Store({
  state: {
    count: 1
  },
  mutations: {
    increment (state) {
      // mutate state
      state.count++
    }
  }
})
```

```
mutations: {
  increment (state, n) {
    state.count += n
  }
}


store.commit('increment', 10)
```

```
import { mapMutations } from 'vuex'

export default {
  // ...
  methods: {
    ...mapMutations([
      'increment', // map `this.increment()` to `this.$store.commit('increment')`

      // `mapMutations` also supports payloads:
      'incrementBy' // map `this.incrementBy(amount)` to `this.$store.commit('increm
    ]),
    ...mapMutations({
      add: 'increment' // map `this.add()` to `this.$store.commit('increment')`
    })
```

ACTIONS_

- FUNCTIONS THAT COMMIT MUTATIONS
- CAN PERFORM ASYNC OPERATIONS
- RECEIVE A CONTEXT OBJECT WITH SAME METHODS AND PROPERTIES OF THE STORE INSTANCE
- FROM COMPONENT THEY ARE DISPATCHED AND ACCEPT PAYLOAD
- CAN BE MAPPED AS METHODS
- CAN BE COMPOSED WITH ANOTHER ACTIONS

ACTIONS_

```
export default {
  // ...
  methods: {
    ...mapActions([
      'increment', // map `this.increment()` to `this.$store.dispatch('increme

      // `mapActions` also supports payloads:
      'incrementBy' // map `this.incrementBy(amount)` to `this.$store.dispatch
    ]),
    ...mapActions({
      add: 'increment' // map `this.add()` to `this.$store.dispatch('increment
    })
```

```
actions: {
  actionA ({ commit }) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        commit('someMutation')
        resolve()
      }, 1000)
    })
  }
```

```
actions: {
  // ...
  actionB ({ dispatch, commit }) {
    return dispatch('actionA').then(() => {
      commit('someOtherMutation')
    })
  }
```

STORE MODULARITY_

- LARGE APPLICATIONS MAY REQUIRE TO BREAK STORE INTO SEPARATED MODULES
- EACH MODULE CAN HAVE ITS OWN ACTIONS
- MUTATIONS ACCESS LOCAL STATE AND GETTERS ALSO ROOTSTATE AS THIRD ARGUMENT, ACTIONS HAVE ALSO ROOTSTATE INTO CONTEXT
- BY DEFAULT THEY ARE GLOBAL, BUT CAN BE NAMESPACED WITH MODULE PATH
- DON’T MODULARIZE IN EARLY STAGES

STORE MODULARITY_

```
const moduleA = {
  state: { ... },
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: { ... },
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> `moduleA`'s state
store.state.b // -> `moduleB`'s state
```

```
const store = new Vuex.Store({
  modules: {
    account: {
      namespaced: true,

      // module assets
      state: { ... }, // module state is already nested and not affected by
      getters: {
        isAdmin () { ... } // -> getters['account/isAdmin']
      },
      actions: {
        login () { ... } // -> dispatch('account/login')
      },
      mutations: {
        login () { ... } // -> commit('account/login')
      },
```

STORE STRUCTURE_

- **APPLICATION STRUCTURE IS KEY TO DEVELOPMENT SCALABILITY**

```
├── index.html
├── main.js
├── api
|   └── ... # abstractions for making API requests
├── components
|   ├── App.vue
|   └── ...
└── store
    ├── index.js          # where we assemble modules and export the store
    ├── actions.js        # root actions
    ├── mutations.js      # root mutations
    └── modules
        ├── cart.js       # cart module
        └── products.js   # products module
```

STORE WITH SERVICES_

- A GOOD PATTERN IS TO DECOUPLE ASYNC OPERATIONS INTO SERVICES
- SERVICES RESPONSIBILITY IS TO PERFORM AN OPERATION
- THEY PROVIDE REUSABILITY AND IMPLEMENTATION DECOUPLING
- THEY CAN BE USED ACROSS ACTIONS OF THE STORE
- SERVICES CAN BE GROUPED BY BOUNDED CONTEXTS
- THEY USUALLY RECEIVE STATE, SUCCESS CALLBACK, ERROR CALLBACK AND OPTIONAL PAYLOAD

STORE PLUGINS_

- YOU CAN EXTEND STORE FUNCTIONALITY SUBSCRIBING TO MUTATIONS
- FOR EXAMPLE STORE PERSISTENCE OR BUILT-IN LOGGING

```
const myPlugin = store => {
  // called when the store is initialized
  store.subscribe((mutation, state) => {
    // called after every mutation.
    // The mutation comes in the format of `{ type, payload }`.
  })
}
```

```
const store = new Vuex.Store({
  // ...
  plugins: [myPlugin]
})
```

THANK YOU