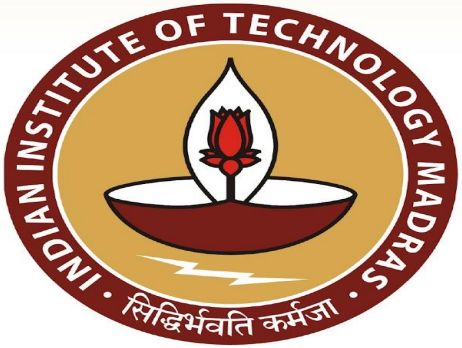


IIT Madras
BSc Degree

Still More Vue

- State Management
- Routes
- SPAs etc.

State Management

UI State

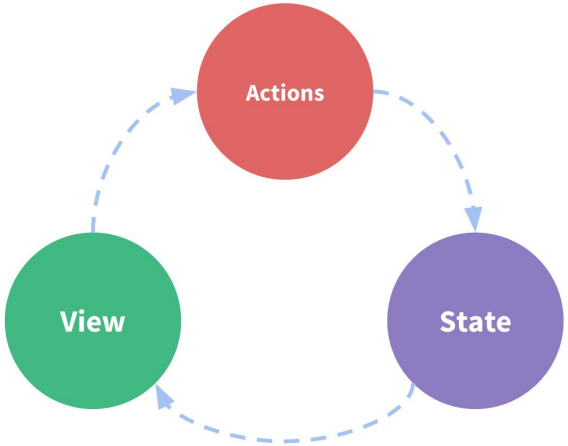
Core idea of declarative programming:

$$UI = f(State)$$

State management pattern

- State
  - “Source of truth” about internals of the app
- View
  - function of State - declarative mapping
- Actions
  - view provides input: action
  - state changes in response to action

One-way data flow



Src: Vuex documentation

Contrast with MVC

- Here we are only looking at UI state: not system state
- MVC can still be used on server to update system state
- Not either/or

Hierarchy - multiple components

- Parent -> child
  - pass information through props
- Child -> parent
  - pass information through events
  - can directly invoke parent functions or modify parent data
    - not desirable: breaks clean separation of code
    - harder to debug

Problem: multiple components

- Multiple views may depend on same piece of state
- Actions from different views may try to modify state
- “Sibling” components
  - At same or similar levels of hierarchy
  - Pass events up from source until common parent
  - Pass props back down to destination

Solution - Global variables?

- Directly accessible from all components
- All components can modify a state variable
- All components can read a state variable for updating views

Problem:

- Keeping track of who modified what is difficult!
- Harder to debug/maintain

Solution - Restricted Global access

- Global still required so all components can update their views easily
- But changes should be constrained
  - No direct modification of state variable
  - Only through special mutation actions

Vuex - state management library for Vue.js

Similar ideas - Flux

- From Facebook - primarily meant for React
- Unidirectional data flow
  - store - maintains the state variables
  - dispatcher - sends action messages
  - view - React components that update based on state

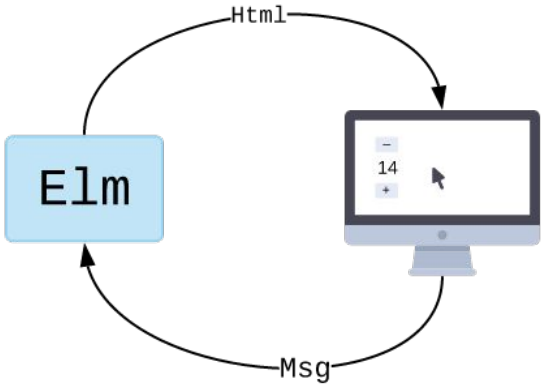
Similar ideas - Redux

- Three principles of Redux:
  - Single source of truth
  - State is read-only: explicitly return a new state object - easier to trace
  - Changes made by “pure” functions - no side effects: changes easy to trace

Similar ideas - Elm architecture

Elm: Functional language designed for web application development

- **Model** — the state of your application
- **View** — a way to turn your state into HTML
- **Update** — a way to update your state based on messages



Vuex

Vuex

- State management library for Vue
- Introduces a new “store” that is globally accessible
- Officially supported by Vue

Example

Vuex Store

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

Use in Component

```
const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return store.state.count
    }
  }
 }
```

Vuex concepts

- Single shared state object
  - Tree structure to capture component nesting
  - Similar constraints on data to Vue data object
- Components can still have local state
  - Not to be seen/used outside component
- Getter methods:
  - computed properties on shared state objects
- Access within components
  - `this.$store` available within all components

Mutations

- To change state: “commit” a mutation
- Never directly update a variable
  - Always call a method that updates
  - Explicitly “commit” this action - ensure it can be tracked and recorded
- Must be **synchronous**

Debugging support

- Recorded in devtools
  - Allows “time travel” debugging - retrace steps that caused a problem
- List of all mutations requested, who requested, time of request
  - Can play back mutations in order from beginning
  - Reproduce system state at any point - time travel…

Example

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

Usage scenarios:

- Normal
  - `store.commit('increment')`
- With argument
  - `store.commit('increment', 10)`
- Object

```
store.commit({
  type: 'increment',
  amount: 10
})
```

Actions

- Mutations must be synchronous - no async calls permitted
  - Some data updates may not be possible to sync
- Actions can contain async functionality
  - do not change state directly: commit mutations

Example:

```
actions: {
  increment ({ commit }) {
    commit('increment')
  }
}
```

```
store.dispatch('increment')
```

Why double work?

- Actions can contain async calls

```
actions: {
 checkout ({ commit, state }, products) {
   // save the items currently in the cart
   const savedCartItems = [...state.cart.added]
   // send out checkout request, and optimistically
   // clear the cart
   commit(types.CHECKOUT_REQUEST)
   // the shop API accepts callbacks
   shop.buyProducts(
     products,
     // handle success callback
     () => commit(types.CHECKOUT_SUCCESS),
     // handle failure callback
     () => commit(types.CHECKOUT_FAILURE, savedCartItems)
   )
 }
}
```

Composing actions

```
// assuming `getData()` and `getOtherData()` return Promises

actions: {
   async actionA ({ commit }) {
     commit('gotData', await getData())
   },
   async actionB ({ dispatch, commit }) {
     await dispatch('actionA') // wait for `actionA` to finish
     commit('gotOtherData', await getOtherData())
   }
 }
```

Summary

- State management is complex when dealing with multiple components
- Some kind of globally accessible state required
- Controlled mutation important to allow maintainability

Routing

Page composition

- Original:
  - all pages are HTML from server
- Vue-like frameworks:
  - components
  - parts of app can correspond to components instead of HTML pages
  - application - not just sequence of pages?

Example

```
<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>
```

Src: https://router.vuejs.org/guide/ #html

Example

```
// 1. Define route components.
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. Each route should map to a component.
const routes = [
 { path: '/foo', component: Foo },
 { path: '/bar', component: Bar }
]

// 3. Create the router instance and pass the `routes` option
const router = new VueRouter({
 routes // short for `routes: routes`
})

// 4. Create and mount the root instance.
const app = new Vue({
 router
}).$mount('#app')
```

Src: https://router.vuejs.org/guide/#javascript

Advantages

- Clickable links to transition between components
  - No need of actual HTML page
- Clicks handled by client JS, no need to hit server
- Can replace parts of existing page - limit refreshes

Dynamic routes

```
// User component definition
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
 // Dynamic route
const router = new VueRouter({
  routes: [
    // dynamic segments start with a colon
    { path: '/user/:id', component: User }
  ]
})
```

Impact on reactivity

- Navigate from /user/one to /user/two reuses same component
  - May not trigger reactive updates
- Install a watcher on $route object

```
const User = {
   template: '...',
   watch: {
     $route(to, from) {
       // react to route changes...
     }
   }
 }
```

More features

- Nested routes
  - router-view inside a component
- Named routes
  - readability and maintainability
- Named views
  - associate multiple components with different router-view by name
- HTML5 history mode
  - push URLs into the history of the browser
  - allow more natural navigation
  - Better user experience - not fundamentally different concept

Why router?

- Routes handled in JS
- Routes associated with components
- Navigation inside a single app - updates from server only on demand

Single Page Applications

SPAs and More

Web Application User Experience

- HTML -> Navigation by clicking links, posting to forms
  - Load new pages: server rendered
  - Form submission processed and rendered on server
- Full back and forth from server: round-trip delays
  - Page loading / transitions

Alternative

- Handle navigation as far as possible on client
- Asynchronous fetch only required data to update parts of page
- Page transitions and history handled through JS
- API + JS

Single page application

- Dynamic website
- Rewrite current page instead of re-rendering with fresh load
- Why?
  - User experience: faster transitions, page loads
  - Feel more like a native app
- Examples?
  - Gmail
  - Facebook
  - Google maps

How?

- Transfer all HTML in one request
  - Use CSS selectors, display controls to selectively display
  - Large load time, memory
- Browser plugins
  - Java applets, Shockwave Flash, Silverlight
  - Significant overhead, compatibility issues
- AJAX, fetch APIs
  - Asynchronous fetch and update parts of DOM
  - Most popular with existing browsers
  - Requires powerful rendering engines
- Async transfer models
  - Websockets, server-sent events
  - more interactive, can be harder to implement

Impact on server

- Thin server
  - Only stateless API responses
  - All state and updates with JS on browser
- Thick stateful server
  - Server maintains complete state
  - Requests from client result in full async load, but only partial page refresh
- Thick stateless server
  - Client sends detailed information to server
  - Server reconstructs state, generates response: only partial page refresh
  - Scales more easily: multiple servers need not sync state

Running locally

- Can be executed from a `file://` URI
- Download from server, save to local filesystem
  - Subsequent requests served locally
  - App update? Reload from server
- Use WebStorage APIs

Challenges

- Search engine optimization
  - Links are often local, or #
- Managing browser history
  - Can confuse users: browser history API changes
- Analytics
  - Tracking popular pages not possible on local load

Single page application with Vue

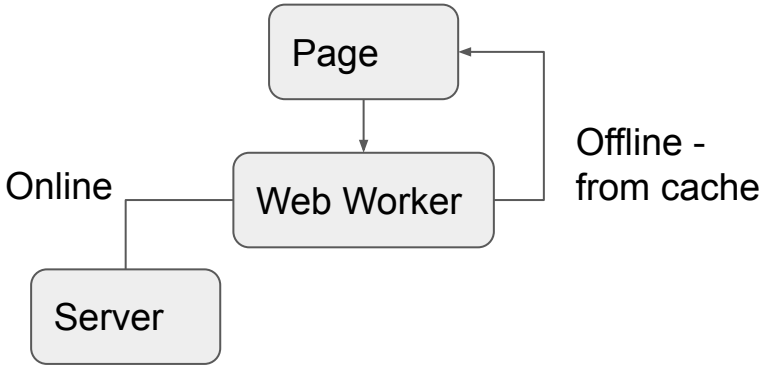
- Complex application logic:
  - Backend on server
- Frontend state variables
  - Vue + Vuex
- Navigation and page updates
  - Vue router
  - Component based

Progressive Web Apps

- Often confused with SPA
  - Very often PWA implemented as an SPA
- Not all SPAs need to be PWAs
  - May be single page but without web workers, offline operation etc.
- Not all PWAs need to be SPAs
  - May have offline and web workers, where rendering is done on server/web worker, not JS

Web Workers

- Script started by web content
  - Runs in background
- Worker thread can perform computations, fetch requests
- Send messages (events) back to origin webcontent



Characteristics

- Installability
- Web Manifest: metadata to identify to operating system
- WebAssembly:
  - faster operation possible - compiled
- Storage
  - Web storage APIs
- Service workers

Example: https://app.diagrams.net/

Web apps vs Native

- Native:
  - Compiled with SDKs like Flutter, Swift SDK
  - Best access to underlying OS
  - Restrictions minimized with OS support
  - Look and Feel of native, but not uniform across devices
- Web apps:
  - write once, run anywhere (original Java slogan)
  - Simple technologies, low barrier to entry
  - Evolving standards