Advanced State Management

Created	@February 8, 2022 5:33 PM
Type	Lecture
Week	7
Lecture #	1
Lecture URL	https://youtu.be/MyJe5TNl-rY
Notion URL	https://21f1003586.notion.site/Advanced-State-Management-0c561e21e4764f75ad91fb745c0a26b1

State Mangement

UI State

Core idea of declarative programming

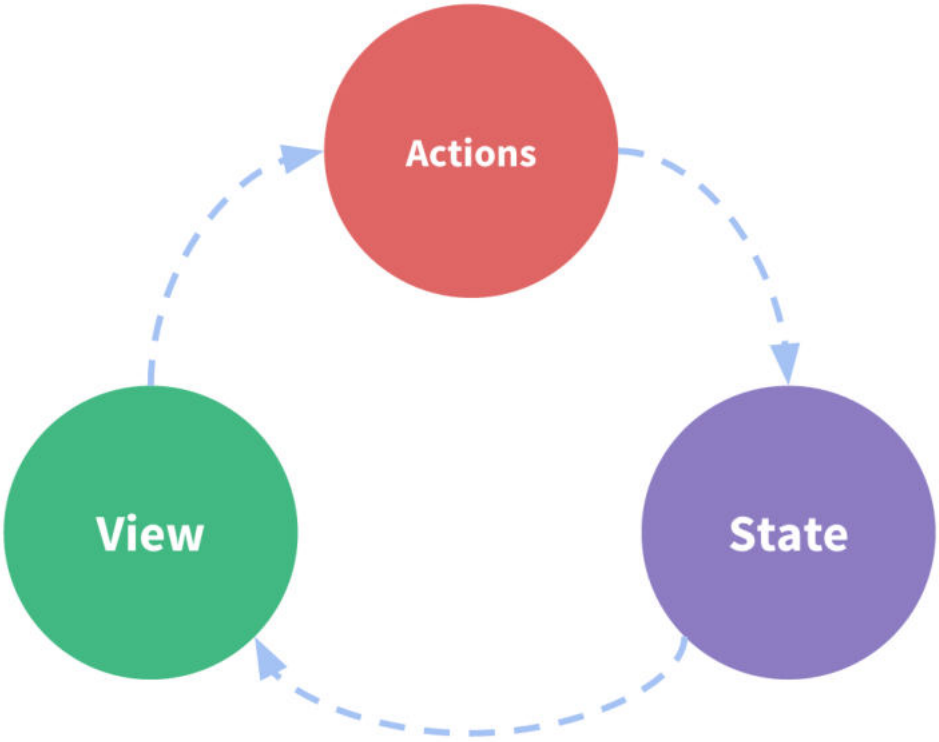
```
UI = f(State)
```

State Management Pattern

- State

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

One-way flow of data



Source: https://vuex.vuejs.org/

Contrast with MVC

- Here, we only look at the UI state
    - Not the system state
- MVC can still be used on server to update the system state
- It’s not like either this or that

Hierarchy - multiple components

- Parent → Child
  - pass information through props
- Child → Parent
  - pass information through events
  - can directly invoke parent functions or modify the parent data
    - it is, however, not desirable
      - it breaks clean separation of code
    - it is harder to debug

Problem → multiple components

- Multiple views may depend on the same piece of state
- Actions from different views may try to modify the 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 the views

Problem:

- Keeping track of which component modified what is difficult
- It is quite a challenge to debug and/or maintain

Solution - Restricted Global access

- Global variables are still required so all the components can update their views easily
- But changes should be constrained
  - No direct modification of the state variable

- Only through special mutation actions

Vuex — state management library for Vue.js

Similar ideas - Flux

- From Facebook — primarily meant for React
- Unidirectional flow of data
  - 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 are easy to trace

Similar ideas - Elm architecture

Elm: Functional language designed for web application development

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

Intro to Vuex

Created	@February 8, 2022 6:21 PM
Type	Lecture
Week	7
Lecture #	2
Lecture URL	https://youtu.be/ml8gFCV7l8s
Notion URL	https://21f1003586.notion.site/Intro-to-Vuex-d9dec616c5ec4c02b5c394abd101c062

Vuex

- It is a state management library for Vue
- Introduces a new “store” that is globally accessible
- It is officially supported by Vue
- Website: https://vuex.vuejs.org/

Example

Vuex Store

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

Use in a 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 the component
- Getter methods
  - Computer 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 the state directly → always 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 that are 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

Created	@February 8, 2022 7:22 PM
Type	Lecture
Week	7
Lecture #	3
Lecture URL	https://youtu.be/7g5T_gKOblU
Notion URL	https://21f1003586.notion.site/Routing-85af2cb5755c4da68178531450ef5b0e

Page composition

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

Vue router Example

```
<div id="app">
  <h1>Hello App!</h1>
  <p>
    <!-- use the router-link component for navigation. -->
    <!-- specify the link by passing the to prop. -->
    <!-- <router-link> will render an <a> tag with correct href attribute -->
    <router-link to="/">Go to Home</router-link>
    <router-link to="/about">Go to About</router-link>
  </p>
  <!-- route outlet -->
  <!-- component matched by the route will render here -->
  <router-view></router-view>
</div>
```

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

```
// 1. Define route components.
// These can be imported from other files
const Home = { template: '<div>Home</div>' }
const About = { template: '<div>About</div>' }

// 2. Define some routes
// Each route should map to a component.
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
]

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

// 4. Create and mount the root instance.
const app = Vue.createApp({})
// Make sure to _use_ the router instance to make the
// whole app router-aware.
app.use(router)

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

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

Advantages

- Clickable links to transition between the components
    - No need to fetch the actual HTML pages from the server

- Clicks are handled by the JS, no need to hit the server
- It can replace the parts of the existing web page — limit refreshes

Dynamic routes

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>',
}

// these are passed to `createRouter`
const routes = new VueRoute({
  routes: [
    // dynamic segments start with a colon
    { path: '/users/:id', component: User },
  ]
})
```

Impact on reactivity

- If the user navigates from `/user/one` to `/user/two`, Vue re-uses the same component
  - It may not trigger reactive updates
- Install a watcher on the `$route` object

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

Other notable features

- Nested routes
  - router-view inside a component
- Named routes
  - For readability and maintainability
- Named views
  - Associated multiple components with different router-view by a name

- HTML5 history mode
  - Push URLs to the browser history
  - Allows for more natural navigation
  - Better use experience

SPAs and more

Created	@February 8, 2022 8:19 PM
Type	Lecture
Week	7
Lecture #	4
Lecture URL	https://youtu.be/CYcSh23IFbk
Notion URL	https://21f1003586.notion.site/SPAs-and-more-328710b75d9e4acb8de1929ba358adb9

Web Application User Experience

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

Alternative

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

Single Page Applications

- Dynamic website
- Re-write current page instead of re-rendering with fresh load
- Why?
  - User experience: faster transitions, page loads
  - Feels 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 the DOM
  - Most popular with existing browsers
  - Requires powerful rendering engine
- Async transfer models

    - Websockets, server-sent events
        - More interactive, can be difficult to implement

Impact on the server

- Thin server
    - Only stateless API responses
    - All state and updates with JS on the browser
- Thick stateful server
    - Server maintains complete state
    - Requests from the client result in full async load, but only partial page refresh
- Thick stateless server
    - Client sends detailed information to the 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 the server
- Use WebStorage APIs

Challenges

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

Single Page Application with Vue

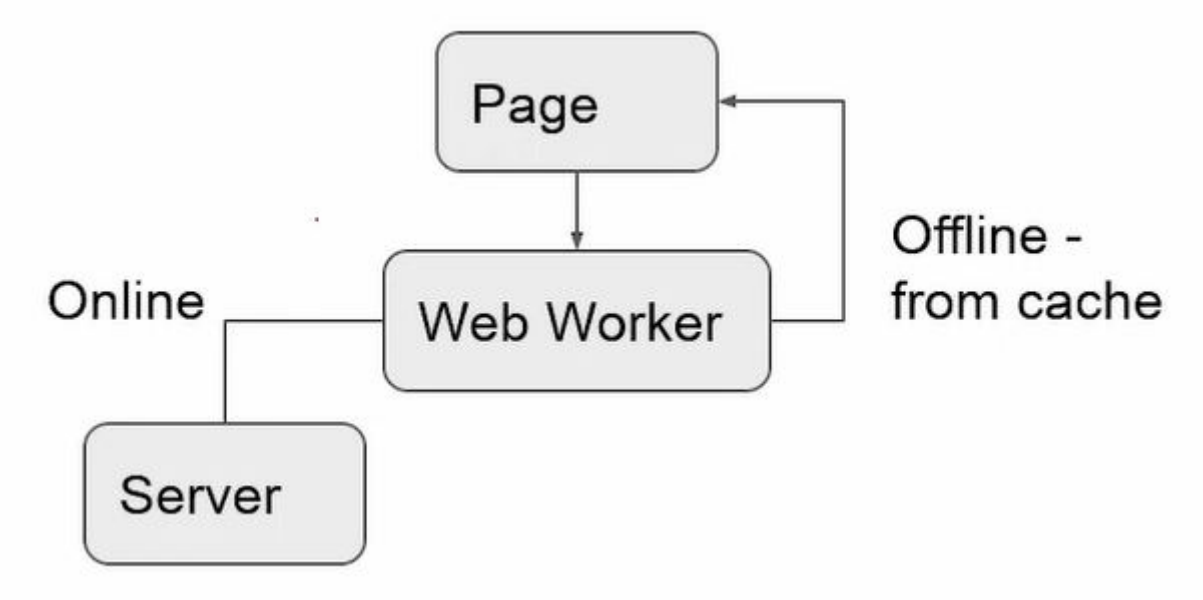
- Complex application logic
  - Backend on server
- Frontend state variables
  - Vue + Vuex
- Navigation and page updates
  - Vue router
  - Component based

Progressive Web Apps

- Often confused with SPA
  - Very often PWA implemented as an SPA
- Not all SPAs need to be PWA
  - 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 the background
- Worker thread can perform computations, fetch requests
- Send messages (event) back to origin webcontent



Characteristics

- Instability
- 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 the underlying OS
  - Restrictions minimized with OS support
  - Look and Feel of native, but not uniform across devices
- Web apps:
  - Write once, Run anywhere

- Simple technologies, low barrier to entry
- Evolving standards