PREGRADO

UNIDAD 2 | FRONTEND WEB APPLICATIONS

WEB CLIENT SECURITY

SI730 | Aplicaciones Web

Al finalizar la unidad de aprendizaje, el estudiante ejecuta un proceso de ingenieríade software para crear una solución de aplicación frontend, utilizando el lenguaje JavaScript y un framework de actualidad, integrada con RESTful web services, con una Arquitectura Orientada a Servicios, en un ambiente de desarrollo ágil y colaborativo, comunicando resultados y proceso de ingeniería aplicado, bajo un enfoque dirigido por la innovación e inclusión.

AGENDA

TEMPORARY STORAGE

STATE MANAGEMENT

SIGN-IN, SIGN-UP & SIGN-OUT

Window.localStorage and sessionStorage

La propiedad **localStorage** permite acceder al objeto Storage para el Document's origin. La data almacenada se preserva entre las sesiones del browser y ésta no expira.

localStorage es similar a **sessionStorage**, pero la data en **sessionStorage** se limpia cuando finaliza la sesión de la página (cuando la página se cierra).

La data en **localStorage** o **sessionStorage** es específica del protocolo de la página, es decir hay un objeto localStorage para **http**://example.com y otro para **https**://example.com.

Window.localStorage

Syntax

```
myStorage = window.localStorage;
```

Properties

```
Storage.length
```

Methods:

```
Storage.key()
Storage.getItem()
Storage.setItem()
Storage.removeItem()
Storage.clear()
```

Using localStorage

Usando localStorage para almacenar información.

```
import axios from 'axios';

const API_URL = 'https://localhost:5001/api/users/authenticate/';

class AuthService {
    login(user) {
        return axios.post(API_URL, {
            username: user.username,
            password: user.password
        })
            .then(response => {
                if (response.data.token) {
                    console.log("user:" + response.data);
                    localStorage.setItem('user', JSON.stringify(response.data));
                }
                return response.data;
            });
    }
    logout() {
        localStorage.removeItem('user');
    }
    register(user) {
        return axios.post(API_URL + 'sign-up', {
            firstName: user.firstName,
            lastName: user.lastName,
            username: user.username,
            password: user.password
        });
    }
}

export default new AuthService();
```

Using localStorage

Usando localStorage para recuperar información.

```
export default function authHeader() {
    let user = JSON.parse(localStorage.getItem('user'));

    if (user && user.token) {
        console.log(`Bearer ${user.token}`);
        return {'Authorization': 'Bearer ' + user.token};
    } else {
        console.log("No token available");
        return {};
    }
}
```

Making requests with JSON Web Token

Estableciendo token en request header.

```
import axios from 'axios';
import authHeader from "@/services/auth-header";

const API_URL = 'https://localhost:5001/api/users/';


class UserService {
    getAll() {
        console.log(authHeader())
        return axios.get(API_URL, { headers: authHeader()});
    }
}


export default new UserService();
```

State Management

Cuando las aplicaciones crecen en objetos de programación, pueden crecer en complejidad.

Múltiples piezas de state (estado) pueden estar diseminadas por muchos componentes que requieren interactuar entre sí.

Para resolver dicho problema, Vue ofrece

Pinia, un state management **library** para Vue 2.X y Vue 3.X

vuex, un *state management library para Vue 2.X*.

Install Pinia

Vite

```
npm install pinia
```

Install Pinia

Create pinia instance

```
import { createApp } from 'vue'
import { createPinia } from 'pinia'
import App from './App.vue'

const pinia = createPinia()
const app = createApp(App)


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

Pinia Store

Store es una entidad que tiene state y business logic que no está vinculada a su Component tree - Alberga un global state. Es un poco como un componente que siempre está ahí y que todos pueden leer y escribir.

Tiene tres conceptos, el *state*, *getters* y *actions* y es seguro asumir que estos conceptos son el equivalente de data, computed y methods en components.

Options Stores

Pasar un options object

```
export const useCounterStore = defineStore('counter', {
  state: () => ({ count: 0, name: 'Eduardo' }),
  getters: {
    doubleCount: (state) => state.count * 2,
  },
  actions: {
    increment() {
      this.count++
    },
  },
})
```

Setup Stores

Pasar una function

```
export const useCounterStore = defineStore('counter', () =>
{
  const count = ref(0)
  const name = ref('Eduardo')
  const doubleCount = computed(() => count.value * 2)
  function increment() {
    count.value++
  }


  return { count, name, doubleCount, increment }
})
```

Install vuex

Vue CLI

```
npm install vuex
```

Vuex Store

Como centro de toda aplicación con Vuex está el **store**.

Un **store** es un container que almacena el application **state**.

Características:

Vuex stores son reactivos. Cuando los Vue components recuperan un state de él, van a actualizar de forma directa y reactiva si el state del store cambia.

No se pude mutar de forma directa el state del store, solo a través de mutations. Esto asegura la trazabilidad.

Store

Crear un store.

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

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

Store

Accediendo al **state** del **store**.

```
store.commit('increment')

console.log(store.state.count) // -> 1
```

Store

Accediendo al **state** del **store** desde Vue components, se debe proveer el store al Vue instance.

```
new Vue({
  el: '#app',
  store: store,
})
```

Store

Confirmando un **mutation** en un método de componente.

```
methods: {
  increment() {
    this.$store.commit('increment')
    console.log(this.$store.state.count)
  }
}
```

Getters

Vuex permite definir **getters** en el store. Getters reciben state como primer argumento. Pueden recibir otros getters como segundo argumento.

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

Getters

Accediendo a los getters.

```
computed: {
  doneTodos () {
    return this.$store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]
  }
  doneTodosCount () {
    return this.$store.getters.doneTodosCount // -> 1
  }
}
```

Mutations

Permiten modificar el state en un Vuex store.

Similares a events: cada mutation tienen un string **type** y un **handler**.

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

Mutations

Para invocar un mutation handler, se requiere llamar a store.**commit** con el type.

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

Actions

Similares a mutations, pero en vez de mutar un state, los actions hacen commit de mutations. Pueden además contener varias operaciones asíncronas.

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

Actions

Invocar actions usando ES2015 argument destructuring.

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

Actions

Dispatching Actions con el método `store.dispatch`.

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

Actions

Llamar a un async API y hacer **commit** de múltiples **mutations**.

```
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 a success callback and a failure callback
    shop.buyProducts(
      products,
      // handle success
      () => commit(types.CHECKOUT_SUCCESS),
      // handle failure
      () => commit(types.CHECKOUT_FAILURE, savedCartItems)
    )
  }
}
```

Actions

Hacer dispatch de actions en components usando el helper **mapActions**.

```
import { mapActions } from 'vuex'

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


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

Actions

Los actions normalmente son asíncronos. store.dispatch puede manejar un Promise como retorno de un action handler.

```
actions: {
  actionA ({ commit }) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        commit('someMutation')
        resolve()
      }, 1000)
    })
  }
// ...
  actionB ({ dispatch, commit }) {
    return dispatch('actionA').then(() => {
      commit('someOtherMutation')
    })
  }

}
```

Modules

Vuex permite dividir un store en **modules**. Cada uno puede contener state, mutations, actions, getters e incluso nested modules.

```
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
```

Modules

Para que los modules sean más autocontenidos y reusables, pueden marcarse como namedspaced con `namespaced: true`.

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

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

Modules

Cuando se registra el module, todos los getters, actions y mutations son namespaced.

```
// …
// nested modules
modules: {
  // inherits the namespace from parent module
  myPage: {
    state: () => ({ ... }),
    getters: {
      profile () { ... } // -> getters['account/profile']
    }
  },

  // further nest the namespace
  posts: {
    namespaced: true,

    state: () => ({ ... }),
    getters: {
      popular () { ... } // -> getters['account/posts/popular']
    }
  }
}
})
```

Application Structure Principles

- Application-level state is centralized in the store.
- The only way to mutate the state is by committing mutations, which are synchronous transactions.
- Asynchronous logic should be encapsulated in, and can be composed with actions.

Application Structure Principles

Estructura de proyecto con modules.

```
├── 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
```

Implementing state management

Aplicando state management en authentication (auth.module.js).

Estableciendo Initial state.

```
import AuthService from '../services/auth.service';
const user = JSON.parse(localStorage.getItem('user'));


const initialState = user
    ? { status: { loggedIn: true }, user }
    : { status: { loggedIn: false }, user: null };


export const auth = {
    namespaced: true,
    state: initialState,
// …
```

Implementing state management

Aplicando state management en authentication (auth.module.js).

Definiendo actions.

```
actions: {
    login({ commit }, user) {
        return AuthService.login(user).then(
            user => {
                commit('loginSuccess', user);
                return Promise.resolve(user);
            },
            error => {
                commit('loginFailure');
                return Promise.reject(error);
            });
    },
    logout({ commit }) {
        AuthService.logout();
        commit('logout');
    },
    register({ commit }, user) {
        return AuthService.register(user).then(
            response => {
                commit('registerSuccess');
                return Promise.resolve(response.data);
            },
            error => {
                commit('registerFailure');
                return Promise.reject(error);
            }
        )
    }
},
```

Implementing state management

Aplicando state management en authentication (auth.module.js).

Definiendo mutations.

```
    mutations: {
        loginSuccess(state, user) {
            state.status.loggedIn = true;
            state.user = user;
        },
        loginFailure(state) {
            state.status.loggedIn = false;
            state.user = null;
        },
        logout(state) {
            state.status.loggedIn = false;
            state.user = null;
        },
        registerSuccess(state) {
            state.status.loggedIn = false;
        },
        registerFailure(state) {
            state.status.loggedIn = false;
        }
    }
}
```

Using state management

Aplicando state management en authentication.

Implementando container store.

```
import Vue from 'vue';
import Vuex from 'vuex';
import { auth } from "@/store/auth.module";

Vue.use(Vuex);


export default new Vuex.Store({
    modules: {
        auth
    }
});
```

Implementing Sign-In Feature

Usar services en login feature. Accediendo al store state.

```
import User from "@/models/user";

export default {
    name: "Login",
    data() {
        return {
            user: new User('', '', '', ''),
            loading: false,
            message: ''
        };
    },
    computed: {
        loggedIn() {
            return this.$store.state.auth.status.loggedIn;
        }
    },
    created() {
        if (this.loggedIn) {
            this.$router.push('/profile');
        }
    },
```

Implementing Sign-In Feature

Usar services en login feature. Invocando action.

```
    methods: {
        handleLogin() {
            this.loading = true;
            console.log('Starting Login handling');
            this.$validator.validateAll().then(isValid => {
                if (!isValid) {
                    this.loading = false;
                    return;
                }
                if (this.user.username && this.user.password) {
                    this.$store.dispatch('auth/login', this.user).then(
                        () => { this.$router.push('/profile'); },
                        error => {
                            this.loading = false;
                            this.message = (error.response && error.response.data)
                            || error.message || error.toString();
                        });
                }
            });
        }
    }
};
```

Implementing Sign-Up Feature

Usar services en register feature. Accediendo al store state.

```
import User from "@/models/user";

export default {
    name: "Register",
    data() {
        return {
            user: new User('', '', ''),
            submitted: false,
            successful: false,
            message: ''
        };
    },
    computed: {
        loggedIn() {
            return this.$store.state.auth.status.loggedIn;
        }
    },
    mounted() {
        if (this.loggedIn) {
            this.$router.push('/profile');
        }
    },
```

Implementing Sign-Up Feature

Usar services en register feature. Invocando action.

```
methods: {
    handleRegister() {
        this.message = '';
        this.submitted = true;
        this.$validator.validate().then(isValid => {
            if (isValid) {
                this.$store.dispatch('auth/register', this.user).then(
                    data => {
                        this.message = data.message;
                        this.successful = true;
                    },
                    error => {
                        this.message = (error.response && error.response.data)
                        || error.message || error.toString();
                        this.successful = false;
                    }
                )
            }
        })
    }
}
}
```

Implementing Sign-Out Feature

Usar services en logout feature. Accediendo al store state e invocando action.

```
export default {
  name: 'App',
  components: {},
  computed: {
    currentUser() {
      return this.$store.state.auth.user;
    }
  },
  methods: {
    logout() {
      this.$store.dispatch('auth/logout');
      this.$router.push('/login');
    }
  }
}
```

RESUMEN

Recordemos

localStorage y sessionStorage son alternativas para almacenamiento temporal de información.

Vue ofrece Vuex como alternativa para State Management, permitiendo establecer states, actions y mutations, así como la organización en modules.

REFERENCIAS

Para profundizar

https://jwt.io/

https://vuex.vuejs.org/

PREGRADO

Ingeniería de Software

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova