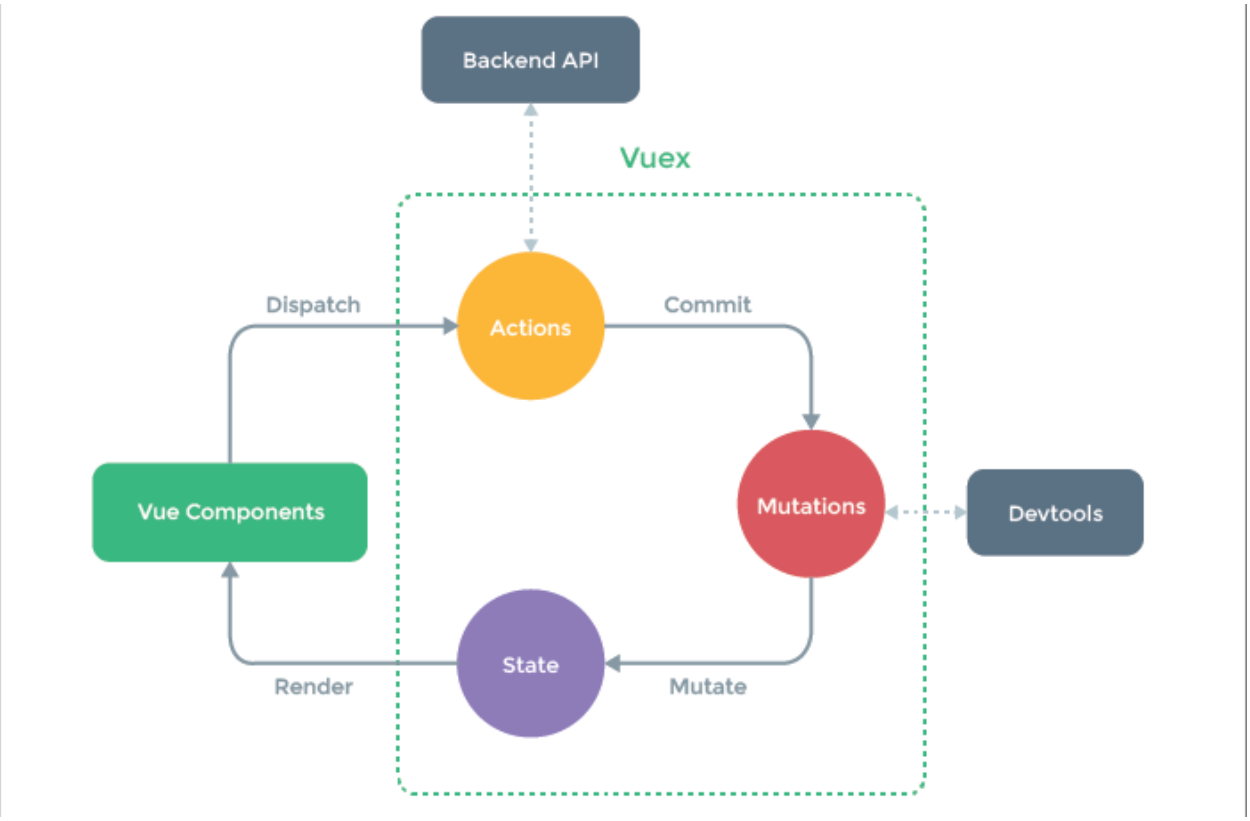
Recitation 9 - Vuex and Vue Router

Introduction

- In the previous recitation, you were introduced to Vue, a reactive framework for the web.
- In today's recitation, we'll be reviewing auxiliary libraries of Vue that help us accomplish more; specifically, ones that enable us to **create "single-page" applications** and **simplify state management**



State management with Vuex

- As you may have realized from a previous recitation and working on your assignment, it can get very cumbersome passing data between the props of different components.
    - Passing props can be tedious for deeply nested components, and simply doesn't work for sibling components.

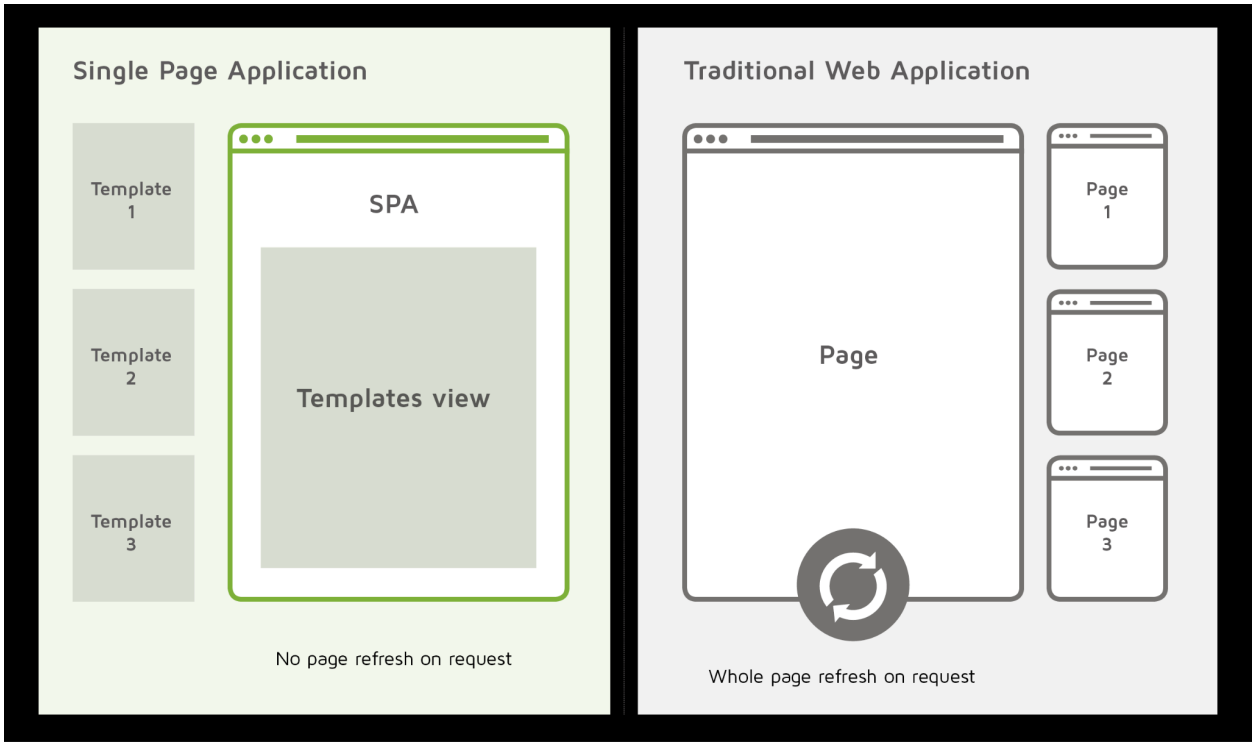
    - We often find ourselves resorting to solutions such as reaching for direct parent/child instance references or trying to mutate and synchronize multiple copies of the state via events.
    - Both of these patterns are brittle and quickly lead to unmaintainable code.
- Instead, we can extract the shared state out of the components into a centralized **store** (think: storage) for all the components to access, with rules ensuring that the state can only be mutated in a predictable fashion.
    - With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!
    - By defining and separating the concepts involved in state management and enforcing rules that maintain independence between views and states, we give our code more structure and maintainability.
- We can accomplish this in Vue through the **Vuex** library
    - Vuex uses a **single state tree** - that is, this single object contains all your application level state and serves as the "single source of truth."
        - Great for keeping track of variables "globally"
    - We can inject a **store** containing an initial **state** and **mutations** into the root app so that we can access it through **this.$store**
        - To modify a value in the state, trigger a state change with the **store.commit** method with the name of the mutation as the parameter
        - Note that **mutations** must be **synchronous**; should you rely on a API call to mutate your store, consider using **actions** instead
    - **Getters** can help you dynamically compute values from store values despite not explicitly storing them (similar to Mongoose getters)
        - For example, consider a list of freets in the state. You might want to filter the stored freets by their author.
        - Instead of having another store for the freets of each author, just make a method style getter that filters the states freets by their author.
        - Getters will receive the state as their 1st argument for you to compute values off of
        - The getters will be exposed on the store.getters object, and you access values as properties
        - You can also pass arguments to getters by returning a function.

Server-side routing

- When you click on a link on a website, your browser communicates with a server to request new content to display for you based on the URL you're visiting.
- When the server responds with HTML content, **the entire page reloads to render the new content**, and the URL in your address bar changes.
- You can save this new URL and come back to the page later on, or share it with others so they can easily find the same page. Additionally, your browser remembers your navigation history and allows you to navigate back and forth.
- This server-delegated process of allowing users to navigate between pages is called **server-side routing**. This is the traditional way of routing for most web applications.

Client-side routing

- On the other hand, modern websites often utilize **client-side routing.**
- In client-side routing, client-side JavaScript can continuously do the following:
    - Intercept user navigation to a different page on the website
    - Dynamically fetch template data accordingly to fill a **view** with
    - Update the page's HTML DOM with the view **without reloading the page!**
- Since no reloads happen, this gives the illusion to users that they are browsing the same page despite performing many interactions over a long period of time, ultimately creating a **more streamlined browsing experience**; there's no awkward page transitions or reloads to deal with!
- However, user behavior associated with browser navigation is broken through this philosophy:
    - Users often link to specific pages in an application; if everything is rendered on the same page, how can they link to different parts of the website?
    - Users may want to move back and forth between pages according to their **browser history;** if we treat everything as the same page, how can we keep track of which part of the website the user just viewed?



Vue Router

- Enter Vue's solution to this problem: managing client-side routing through their **Vue Router** library
    - Maps specific components to be displayed in the router-view depending on the path you're visiting
        - For example, a component mapped to route /docs would be displayed vuejs.org/#/docs
    - Provides convenient ways to change the user's navigation programmatically
    - Tracks browser navigation and history under the hood

```
import Vue from 'vue';
import VueRouter from 'vue-router';
// component imports not shown

Vue.use(VueRouter);

const routes = [
  {path: '/', name: 'Home', component: FreetsPage},
  {path: '/account', name: 'Account', component: AccountPage},
  {path: '/login', name: 'Login', component: LoginPage},
  {path: '*', name: 'Not Found', component: NotFound}
];

const router = new VueRouter({routes});

new Vue({
  router,
  render: h => h(App)
}).$mount('#app');
```

- **Basic usage**
  - Define view components, or import them from other files.
  - Define **routes**, each an object with the following options:
    - **path** to render the component at (e.g. '/', '/register')
    - **component** to represent the rendered component
    - **name** for the route, so that you can reference it the same way even if the path changes
  - Create a **router** instance and pass in the **routes** as a parameter option
  - Similar to how we inject the store into the root app, mount the **router** onto the root instance
- **<router-link>** is the component for enabling user navigation in a router-enabled app, serving as links that users can click on to trigger the re-rendering of the page.
  - It renders as an **<a>** tag with the specified **href** by default, and automatically gets an active CSS class when the target route is active.
  - The target location is specified with the **to** prop (in contrast to **href** for the **<a>** element)!
  - **Always use <router-link> to link within different parts of your SPA!**
  - **Check your understanding:** Where in the starter code do we use **<router-link>?**

        - Select/highlight text to reveal answer: [redacted]
- **Dynamic routing**
    - Often times, we will need to map routes with the given pattern to the same component (e.g. using the same Profile component to render a user profile for each user, but with different user IDs)
    - In vue-router, we can use a dynamic segment in the path to achieve that, very similar to Express
        - You can access the named parameter by accessing the **this.$route.params**; for example, you can access figure out which user you should render a profile for route **/profiles/:userId this.$route.params.userId**
- **Programmatic navigation**: To programmatically change what page gets rendered at any time, access the router through **this.$router**; the methods of the router closely reflect the History browser API
    - To navigate to a different URL, use **router.push()**; this adds the page to the browser history
    - To navigate to a different URL without changing browser history, use **router.replace()**
    - To navigate to a previous page in your browser history, you can use **router.go(-1)**; similarly, you can move to the previous n-th page with **router.go(-n)**, or move to the next n-th page with **router.go(n)**
    - **Check your understanding:** Where in the starter code do we use the router to programmatically change the route? (Hint: authentication)
        - Select/highlight text to reveal answer: [redacted]
    - **Check your understanding:** Why don't we use **<router-link>** in authentication forms and instead rely on programmatic navigation?
        - Select/highlight text to reveal answer: [redacted]
- **Navigation guards**
    - We can prevent users from accessing pages they're not supposed to visit with **navigation guards**
    - Use **router.beforeEach** to check if some page the user is attempting to navigate **to**, **from** a different page, is acceptable, and specify the **next()**

function to control if they can proceed as desired, or redirect them to an entirely different page (similar to Express!)

```
// lines 22-36 in router.ts
router.beforeEach((to, from, next) => {
  if (router.app.$store) {
    if (to.name === 'Login' && router.app.$store.state.username) {
      next({name: 'Account'}); // Go to Account page if user navigates to Login and are signed in
      return;
    }

    if (to.name === 'Account' && !router.app.$store.state.username) {
      next({name: 'Login'}); // Go to Login page if user navigates to Account and are not signed in
      return;
    }
  }

  next();
});
```

    - Example in [Fritter frontend starter code]:
        - If we are navigating to the Login page but are already signed in, redirect to the Account page instead
        - If we are navigating to the Account page but are signed out, redirect to the Login page instead
    - Like Express, ensure that **next()** is called exactly once in any given pass through the navigation guard!
- There's a lot more you can do with Vue Router! Review [its documentation] for more details.

Exercises: [https://github.com/61040-fa22/rec9]

1. Move todo items into the Vuex store
    a. Add a new state variable items
    b. Add a new mutation addItem
    c. Update TodoInputForm and TodoInputPage to use the store
2. Add a new TodoStatsPage with the following details:
    a. the total number of items in the todo list
    b. the total number of items in the todo list containing the word "important" (store getter)
3. Add a new TodoFilterStatsPage with the following details:
    a. Given route /stats/SOMEKEYWORDHERE, shows the total number of items in the todo list containing the keyword (store getter)
4. Add some tools to help user navigate to and from the keyword stats page:

a. Add a new InlineForm programmatically navigating to the corresponding filter page when they submit the form
b. Add a Back button on the TodoFilterStatsPage to allow user to return to their previous page