6.1040 Rec 9

Vuex and Vue Router

What we'll be covering today

- State management with Vuex
    - Mutations
    - Getters
- Server-side vs. client-side routing
- Vue Router
    - Dynamic routing
    - Programmatic navigation
    - Navigation guards

Store & Vuex

Motivation for Vuex

- Passing data between different components can be difficult
  - Tedious for deeply nested components
  - Straight up doesn’t work for sibling components
- Unsustainable workarounds:
  - Reaching for direct parent/child instance references
  - Trying to mutate and synchronize multiple copies of the state via events

Motivation for Vuex

- Instead, extract the shared state out of the components into a centralized **store**
  - Ensures the state can only be mutated in a predictable fashion
  - Any component can access the state or trigger actions, no matter where they are in the tree!
- We can accomplish this in Vue through the **Vuex** library
  - Vuex uses a **single state tree**
  - Great for keeping track of variables "globally"

Basic Vuex usage

- In the store, define:
  - Initial values of the **state**
  - Synchronous **mutations** to modify the **state**
- Inject store into root component
- Accessing store from child components: **this.$store**
  - Call **commit()** with mutation name to trigger mutation
  - Access state values from **.state**

```
// store.ts
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

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

new Vue({ store }).$mount('#app');

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

Exercise: Todo List

https://github.com/61040-fa22/rec9

Demo at /public/demo.html

1. npm install
2. npm run serve

Exercise 1

Refactor todo items to use the Vuex store instead of events

1. Add a new state variable **items**
2. Add a new mutation **addItem**
3. Update **TodoInputForm** and **TodoListPage** to use the store instead of keeping data and emitting events to **TodoListPage**

```
// store.ts
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

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

new Vue({ store }).$mount('#app');

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

Store getters

- Help dynamically compute values from store values
- In the store:
  - Getters receive the **state** as their 1st argument for you to compute values off of
  - You can also pass arguments to getters by returning a function
- The getters will be exposed on the **store.getters** for you to access

```
// store.ts
const store = new Vuex.Store({
  state: {
    products: [
      { name: 'coffee', inStock: true },
      { name: 'tea', inStock: false }
    ]
  },
  getters: {
    inStockProducts: state => {
      return state.products
        .filter(p => p.inStock)
    },
    getProductByName: (state) => (name) => {
      return state.products
        .find(p => p.name === name)
    }
  }
});

// ChildComponent.vue:
computed: {
  itemsInStock() {
    return this.$store.getters
      .inStockProducts.join(', ');
  },
  isTeaInStock() {
    return this.$store.getters
      .getProductByName('tea').inStock;
  }
}
```

Exercise 2a

Make a new page **TodoStatsPage** that displays:

1. The total number of items in the todo list
2. The number of items that include the keyword “important”. (To do this, implement a new store getter that filters the items in the store.)

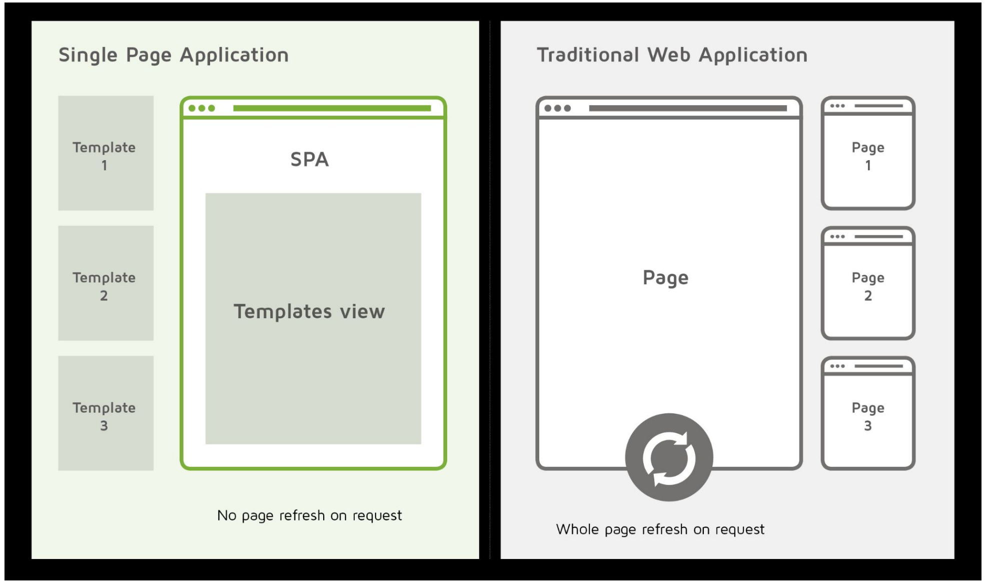
Your answer should be just a few lines! No need for imports, data(), methods() etc. Also, we won’t have a way to view the new page yet.

```
// store.ts
const store = new Vuex.Store({
  state: {
    products: [
      { name: 'coffee', inStock: true },
      { name: 'tea', inStock: false }
    ]
  },
  getters: {
    inStockProducts: state => {
      return state.products
        .filter(p => p.inStock)
    },
    getProductByName: (state) => (name) => {
      return state.products
        .find(p => p.name === name)
    }
  }
});

// ChildComponent.vue:
computed: {
  itemsInStock() {
    return this.$store.getters
      .inStockProducts.join(', ');
  },
  isTeaInStock() {
    return this.$store.getters
      .getProductByName('tea').inStock;
  }
}
```

Routing & Vue Router

Client vs. server-side routing

- Server-side routing (traditional method)
  - Browser requests new page content from web server
  - When the server responds with HTML content, **the entire page reloads to render the new content**

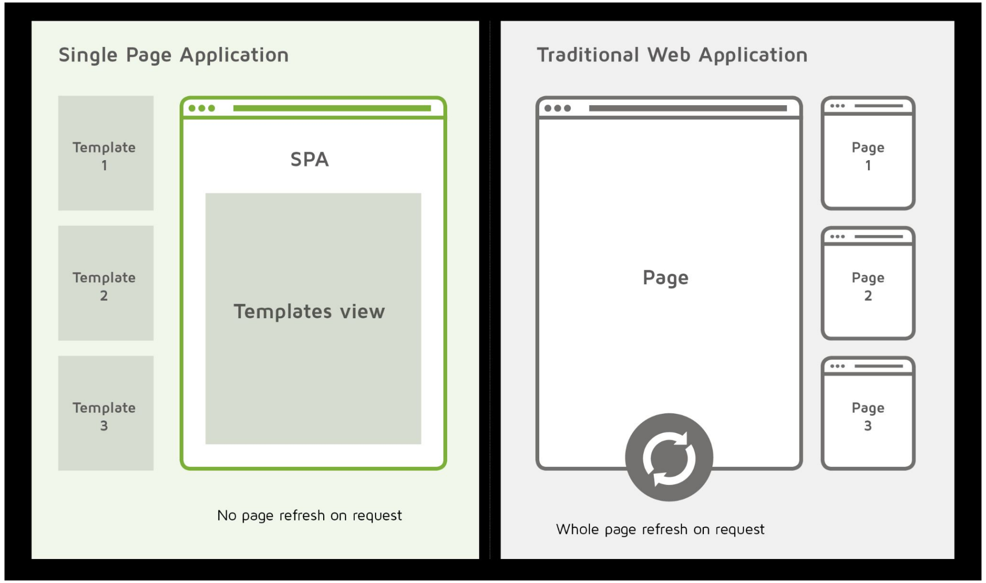


Client-side routing

Server-side routing

Client vs. server-side routing

- Client-side routing (new method)
  - Intercept user navigation to a different page on the website
  - Dynamically fetch template data to update the view **without reloading page**
  - **Problem:** How do we update browser history if we technically stay on the same page the whole time?



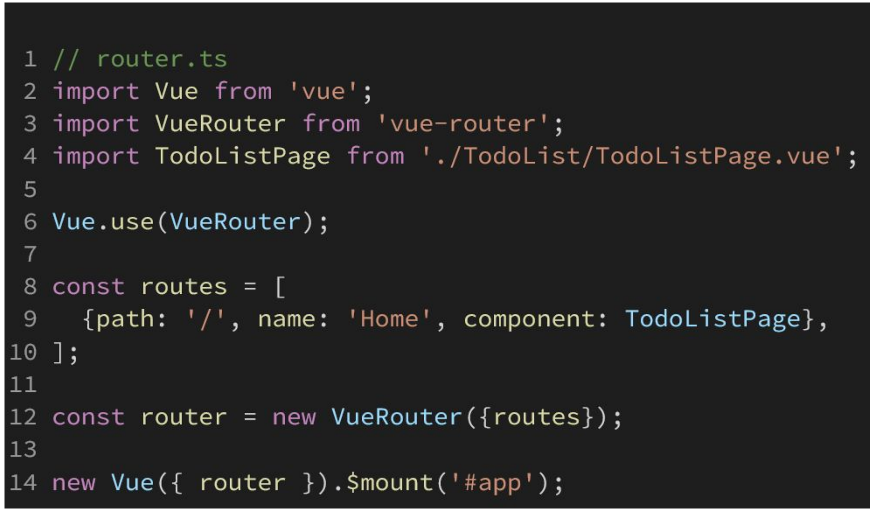
Client-side routing Server-side routing

Vue Router

- 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

Basic router usage

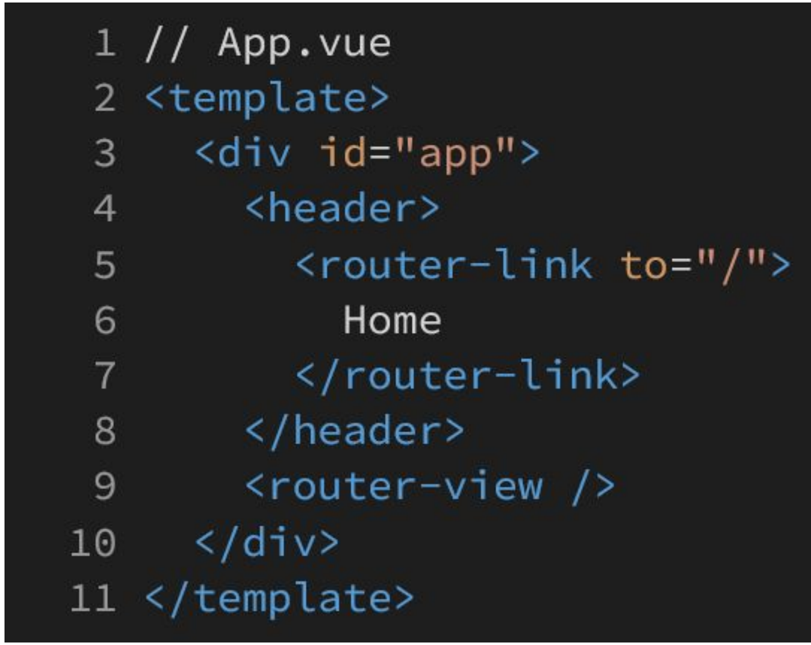
- Define view components, or import them from other files.
- Define **routes**, each an object with the following options:
    - **path** to render the component at
    - **component** to render
    - **name** for the route
- Create a **router** instance
    - Pass in the **routes** as a param option
- Mount the **router** onto the root component



```
// router.ts
import Vue from 'vue';
import VueRouter from 'vue-router';
import TodoListPage from './TodoList/TodoListPage.vue';

Vue.use(VueRouter);

const routes = [
  {path: '/', name: 'Home', component: TodoListPage},
];

const router = new VueRouter({routes});

new Vue({ router }).$mount('#app');
```

Basic router usage

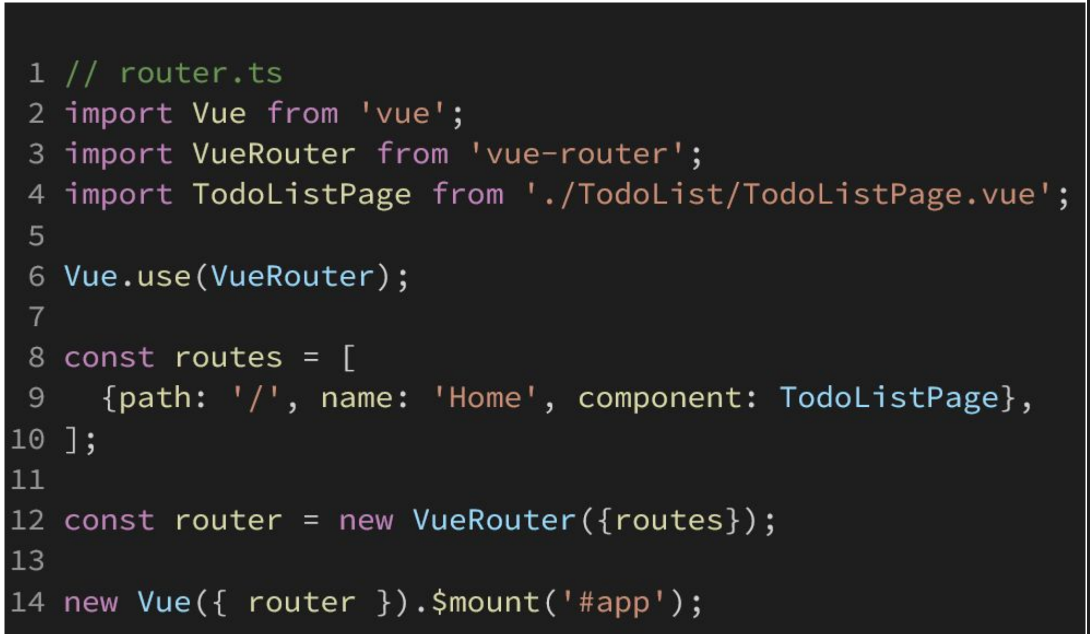
- Page component matching route will replace the **<router-view>** element
- **<router-link>** enables navigation in a router-enabled app
  - It renders as an **<a>** tag with the specified **href** by default
  - Automatically gets an active CSS class when the target route is active
  - The target location is specified with the **to** prop



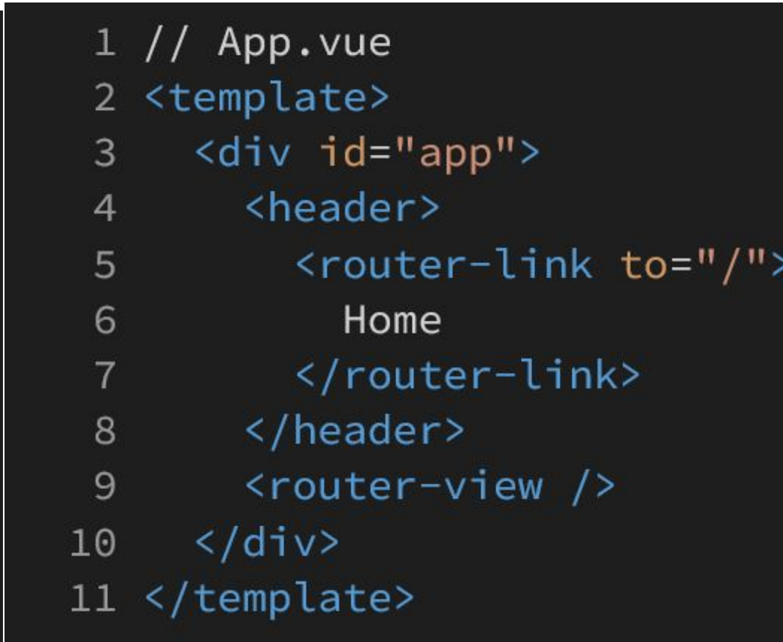
```
// App.vue
<template>
  <div id="app">
    <header>
      <router-link to="/">
        Home
      </router-link>
    </header>
    <router-view />
  </div>
</template>
```

Exercise 2b

1. Modify the router to map the **TodoStatsPage** component you made in 2a to the route “/stats”
2. Add a link to the page in the page header



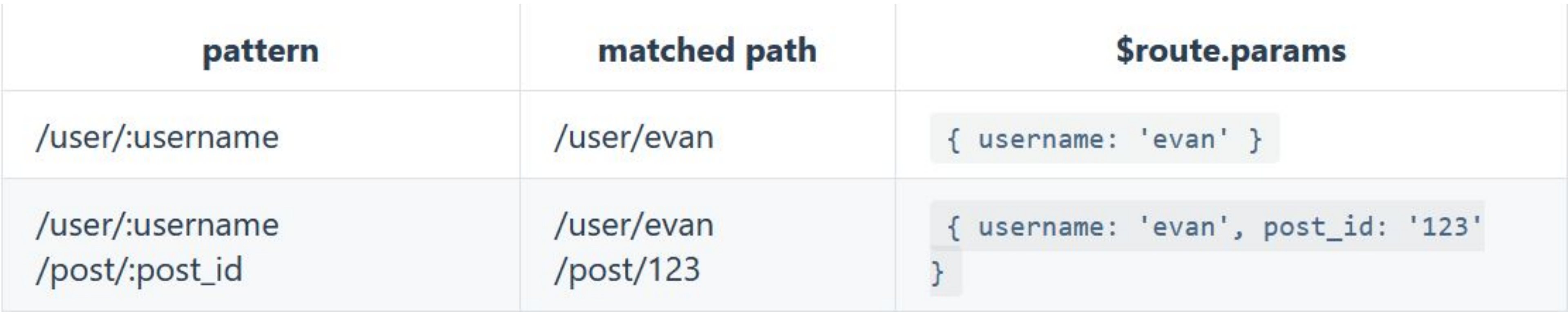
```
// router.ts
import Vue from 'vue';
import VueRouter from 'vue-router';
import TodoListPage from './TodoList/TodoListPage.vue';

Vue.use(VueRouter);

const routes = [
  {path: '/', name: 'Home', component: TodoListPage},
];

const router = new VueRouter({routes});

new Vue({ router }).$mount('#app');
```



```
// App.vue
<template>
  <div id="app">
    <header>
      <router-link to="/"
        Home
      </router-link>
    </header>
    <router-view />
  </div>
</template>
```

Dynamic routing

- Some routes with a common patterns should map to the same component
  - E.g. using the same Profile component to render a user profile for each user, but with different user IDs
- We can use a **dynamic segment** in the path to achieve this
  - Access the named segment from **$route.params**

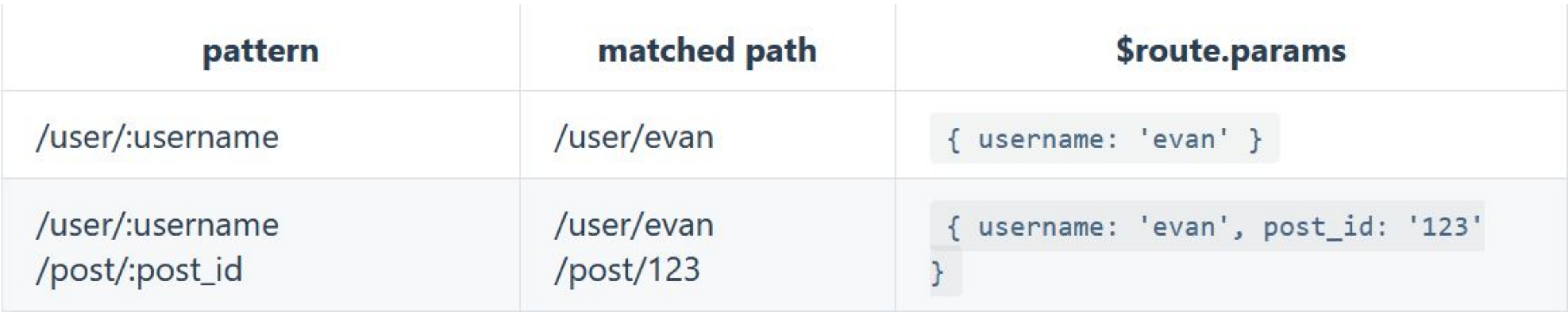


pattern	matched path	$route.params
/user/:username	/user/evan	`{ username: 'evan' }`
/user/:username/post/:post_id	/user/evan/post/123	`{ username: 'evan', post_id: '123' }`

Exercise 3

- Add a new **TodoFilterStatsPage** with the following details:
  - Maps to route **/stats/SOMEKEYWORDHERE** (you'll need to add a *dynamic route* to router.ts)
  - Given the keyword, shows the total number of items in the todo list containing the keyword (you'll need to write a new store getter)

Again, the new page should only be a few lines



pattern	matched path	$route.params
/user/:username	/user/evan	`{ username: 'evan' }`
/user/:username/post/:post_id	/user/evan/post/123	`{ username: 'evan', post_id: '123' }`

Programmatic Navigation

- To programmatically change what page gets rendered at any time, access the router through **this.$router**
  - Navigate to different URL: **router.push()**
  - Navigate to different URL without updating history: **router.replace()**
  - Navigate to previous page in history: **router.go(-1)**
    - Can move to the previous n-th page with **router.go(-n)**
    - Can move to the next n-th page with **router.go(n)**
- **Check your understanding:**
  - Why don't we use **<router-link>** in authentication forms and instead rely on programmatic navigation?
  - Where in the starter code do we use the router to programmatically change the route?

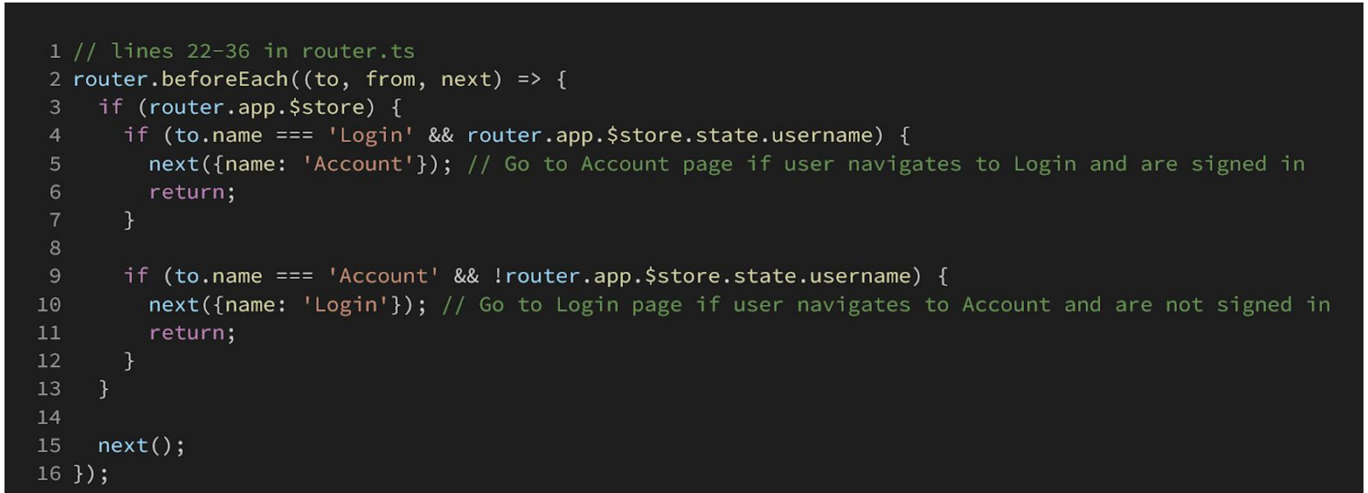
Exercise 4

- Add a tool to help user navigate to and from the keyword stats page:

1. On TodoPage, add a new **TodoFilterForm** that programmatically navigates to the corresponding filter page when they submit the form
   a. **Hint:** Make a copy of **TodoInputForm** and modify accordingly
2. Add a Back button on the **TodoFilterStatsPage** to allow user to return to their previous page

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
- 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();
});
```