Project 6 - State Management

LATE 20/20 Points

7/13/2021

Attempt 1

REVIEW FEEDBACK
7/18/2021

Attempt 1 Score: **20/20**

Add Comment

Unlimited Attempts Allowed

Details

Due: Tuesday, July 13, 2021
Points: 20 points
Deliverables: Upload [Name]BookstoreState.war to https://cs5244.cs.vt.edu:8443/ArchiveUpload/ (https://cs5244.cs.vt.edu:8443/ArchiveUpload/) and submit the URL to this assignment
Resources:

- ShoppingCartResources.zip (https://canvas.vt.edu/courses/132380/files/18823378/download?wrap=1)

Project 6: State Management

Our bookstores currently ask the network for categories and books inside categories repeatedly. In this project we introduce the State Management pattern. A central store object is used to hold state (such as the categories, currently selected category and books for the selected category) centrally in the browser; the central store shares the state efficiently with all page components.

This State Management pattern is so common in Vue applications that there is a standard Javascript library integrated with Vue called "Vuex" (pronounced VIEW-EX). It is named in homage to "Flux" - a pattern created by Facebook engineers at Facebook around 2014 to gather together state and operations in the browser.

Try Again

In this project we transition from our inefficient use of ApiService to the Vuex store to take advantage of the state management pattern. After that we add a ShoppingCart to your Vuex Store. We will use that cart and the store to implement Add-to-Cart buttons and the cart size indicator features of your website.

0 Setup your project

You will setup this project just as you setup Project 5 **except** that in the client you will include the **Vuex plugin** (along with the Linter and the Vue Router).

Create a new (no-plugin) Gradle project in IntelliJ called `[Name]BookstoreState`. Create two modules:

- server - Gradle with Java and web plugins
- client - Vue with Linter and Router **and Vuex**

Here is a summary of the options I chose for my client project:

```
Vue CLI v3.11.0
? Please pick a preset: Manually select features
? Check the features needed for your project: Router, Vuex, Linter
? Use history mode for router? (Requires proper server setup for index fallback in production) Yes
? Pick a linter / formatter config: Prettier
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to invert selection)Lint on save
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? In dedicated config files
? Save this as a preset for future projects? (y/N)
```

Try Again

Once you've done this, I recommend quitting IntelliJ before copying the following files into your new project.

From your Fetch project (P5) copy the following into the client side:

- `src` folder and
- `vue.config.js`.
- If using font-awesome icons, also copy your `public` folder.

Next from your Fetch project (P5) copy the following into the server side:

- `build.gradle` and
- the `src` folder
    - Before copying this, remove the build files and directories from main -> webapp, keeping only `META-INF`, `WEB-INF`, and `index.jsp`.

On the client side, modify the vue.config.js file to change that to `[Name]BookstoreState`.

Set up a Tomcat run configuration for the server (with application context `[Name]BookstoreState`) and run the Tomcat configuration. The REST API should work as in Project 5.

Run the client using "npm run serve". The Home and Category pages should look and behave just as they did in Project 5.

1 Starting your Vuex Store

Create a file underneath your `src` folder called `store.js` to house our Vuex central store. In your store.js we will need to import the Vuex libraries that you should now have in your project. (If you have a store/index.js folder file, please remove

Try Again

Add the following to the top of the store.js file:

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

Let's establish a Vuex store now immediately below the imports.

```
Vue.use(Vuex)

export default new Vuex.Store({
  state: {},
  mutations: {},
  actions: {}
});
```

We are creating a new (default) Vuex.Store object. We can build our new Vuex.Store without changing any components. Later we will convert the components to use the new Vuex.Store object.

Try Again

2 Sharing the Vuex Store with Components Automatically

Using Vuex ensures every component automatically has a reference `$store` to the Vuex store. (similar to how `$route` is on every component).

To ensure this happens, pass in the store to the main Vue component in main.js.

```
import store from "./store";

new Vue({
  router,
  store,  // every component will reference the store now
  render: function(h) {
    return h(App);
  }
}).$mount("#app");
```

Rather than importing using `import { store } from "@/store.js"` here we now use a basic `import store from "./store"`. In JavaScript the ".js" is assumed, and because we now export the Vuex Store by default, that is what we are importing here.

3 Creating a Central Vuex Store

Try Again

Back in `store.js`, let's build the Vuex store's state, mutation and actions without touching components for now.

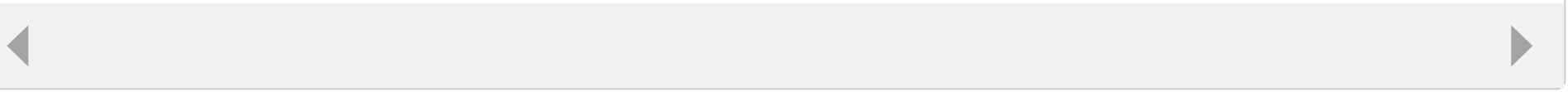
3.1 Adding State to the Store

For our bookstore so far, we can consider three pieces of state: `categories`, `selectedCategoryName`, and `selectedCategoryBooks`. `categories` is clearly a useful shared state variable: we would like to remember the categories and share them among components (e.g. the `CategoryNav`, `HeaderDropdown`, and `HomeCategoryList` (if you have one).

The other elements `selectedCategoryName` and `selectedCategoryBooks` are useful to remember but are less obvious right now as shared elements of state. Their usefulness will be seen when we want to "go back to the selected category page" from other pages (such as a cart page). This avoids having to re-fetch data from the network just to "go back" to a category page.

Let's place the desired state fields into the Vuex Store.

```
export default new Vuex.Store({
  state: {
    categories: [],
    selectedCategoryName: "",
    selectedCategoryBooks: []
  },
  mutations: {},
  actions: {}
});
```



Try Again

3.2 Adding Mutations and Actions to the Store

For each of our `ApiService.js` methods (`fetchCategories` and `fetchSelectedCategoryBooks`), we are going to create one mutation (capturing the ***synchronous*** code that updates the state) and one action (capturing any ***asynchronous*** network calls, and other business logic). The action gathers data and then is said to **commit** mutations into the store to change the store state. This way we can see all state updates (commits) in sequence and can debug our components in any state using the Vue Browser Developer Tool Extensions.

Fetching and storing categories

Let's fetch the categories and add them to the Vuex store. The plan is that our `fetchCategories` **action** first calls the `ApiService.fetchCategories` and then commits the `SET_CATEGORIES` **mutation**, which updates the store.

First, let's add a mutation to set the categories into the store.

Mutations

- The only way to **change state** in a Vuex store is by committing a mutation
- Mutations are **synchronous**; they are tracked by the Vue developer tool
- A mutation handler function takes:
  - the Vuex **state** as its first argument
  - an optional **payload** as its second argument
- By convention, mutations are declared in **ALL_CAPS**
- Mutations are committed: `context.commit("MY_MUT", someValue)`

In the mutations property of the Vuex Store we can add:

```
mutations: {
```

Try Again

```
    }
},
```

Now, let's add an action to load the categories from the ApiService and commit a `SET_CATEGORIES` mutation to the store.

Copy *the* `fetchCategories` *method from the CategoryNav component into the actions property of the Vuex Store.*

Actions

- Instead of mutating state, actions **commit mutations**
- Actions can contain arbitrary **asynchronous** operations (like fetching)
- An action handler function takes:
  - a **context** (or a portion of it) as its first argument
  - an optional **payload** as its second argument
- By convention, actions are styled like other functions, using **camelCase**
- Actions are dispatched: `store.dispatch("myAct", someValue)`

Change the fetchCategories function to take a context parameter:

```
actions: {
  fetchCategories(context) {
    ApiService.fetchCategories()
      ....
```

Try Again

Inside the fetchCategories action we need to change how the categories are stored. In the simple store, we could assign to `vm.categories`. In our Vuex store, we need to commit the new categories into the store using the `SET_CATEGORIES` mutation we just created above. The context contains a commit function to achieve that.

- ***Refactor*** `data` ***to*** `categories`
- ***Replace*** `vm.categories = categories` ***with*** `context.commit('SET_CATEGORIES', categories)`
- ***Remove*** `const vm = this`

3.3 Complete the Vuex.Store mutations and actions.

Add the following to the Vuex.Store:

- A mutation `SELECT_CATEGORY` that takes the state and a category name and updates the `selectedCategoryName` in the state.
- A similar mutation `SET_SELECTED_CATEGORY_BOOKS` that updates the `selectedCategoryBooks` in the state.
- An action `selectCategory` that takes the context and a name parameter, and that updates the store's `selectedCategoryName` by committing the `SELECT_CATEGORY` mutation to the store.
- An action `fetchSelectedCategoryBooks` that takes the context parameter and first uses ApiService to find the selected books, and then updates the store's `selectedCategoryBooks` by committing the `SET_SELECTED_CATEGORY_BOOKS` mutation to the store. Use `context.state` to read any existing state you may need (such as the current selected category name).

Notice we have built a new Vuex store and we have also kept our existing `ApiService` calls from within components as well.

Run your applications. You should see your site work as in Project 5 still, because we have defined but are not yet using the Vuex.Store.

Try Again

4 Start Using Your Vuex Store

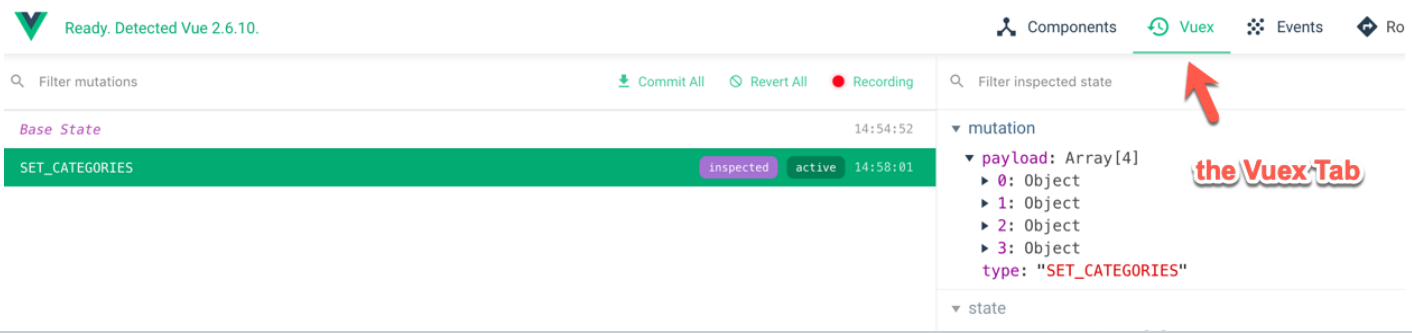
Let us start using the central store by adding calls to the central store mutations and actions from the `App` and `Category` view components - it is when a page is loaded that we will need the categories, and when a category page is loaded we will need books for that category.

Let's add a created method in the App component, using the implicitly available `$store` reference to the Vuex.Store.

```
created: function() {
    this.$store.dispatch("fetchCategories");
}
```

We use the "dispatch" method on the Vuex.Store component to trigger actions using their string names.

Now, in the browser have a look at your components, and the simple store state in the Vue developer tool's "Component" tab. Now go to the "Vuex" tab and you should see a base state and a `SET_CATEGORIES` mutation to inspect.



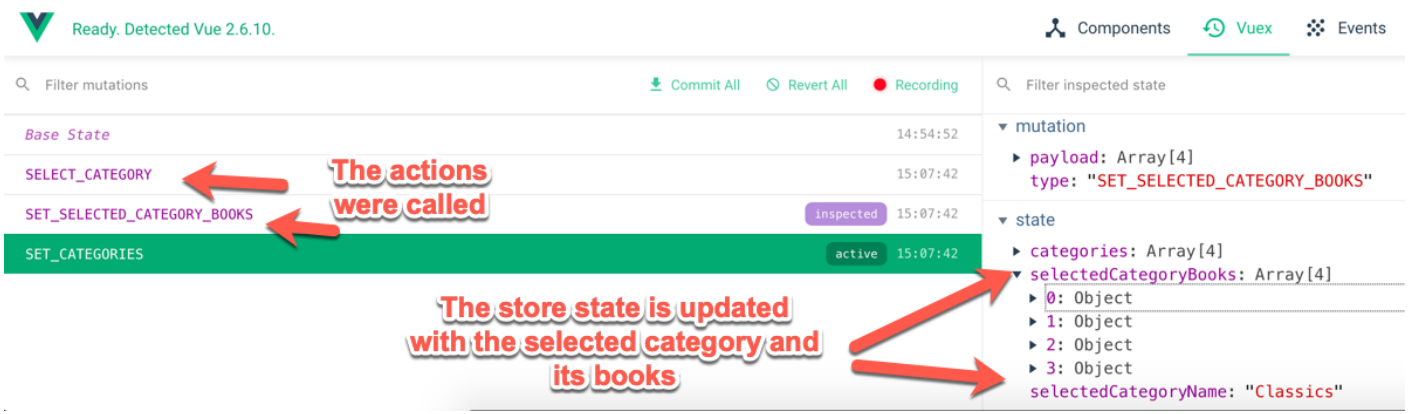
Try Again

The Vuex store tool shows a stream of mutations between states, and allows you to inspect and debug in any state that is visible. Right now only the categories state property has data - as we hoped, the 4 categories in our project!

Let's replace the `created` ***method in the Category.vue component*** to allow the other Vuex.Store state fields to be populated.

```
created: function() {
    this.$store.dispatch("selectCategory", this.$route.params.name);
    this.$store.dispatch("fetchSelectedCategoryBooks");
}
```

When we look in the browser and refresh the category page we can see the actions being called and the state being updated in Vuex.Store.



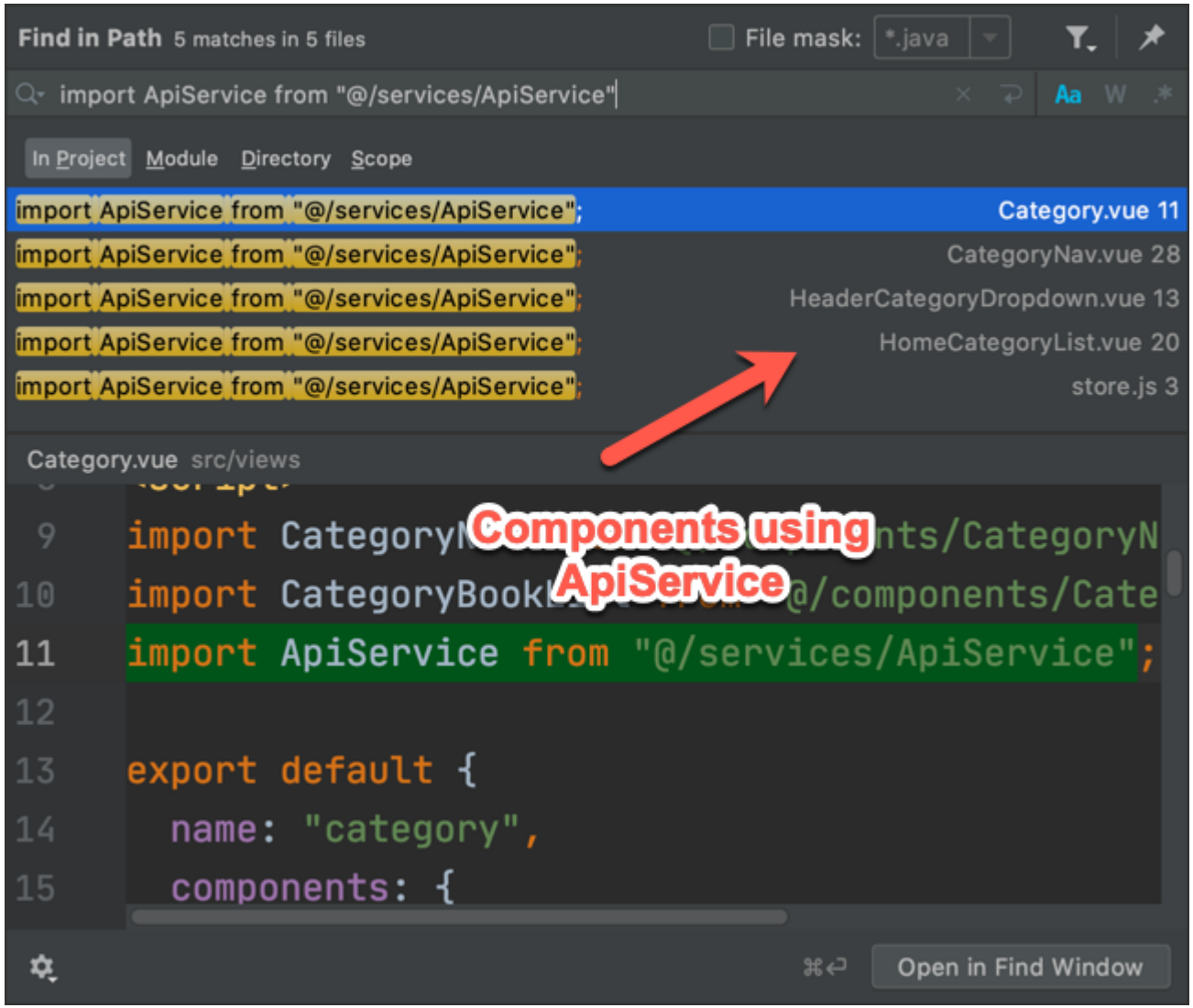
Try Again

5 Stop Using ApiService directly in Components

Since the Vuex store is fully populated, it's time to stop using ApiService directly from components and use dispatched actions instead. Also, by reading the `$store.state` in our templates, we can also reduce the number of network calls being made to display each page.

Let's identify a list of components that used ApiService directly using `import ApiService from "@/services/ApiService"`. We can do that by searching for the import using the Edit | Find | Find in Path | Find in Project dialog box. The `store.js` will use ApiService by design.

Try Again



For each component that uses categories:

- Remove the `categories` data
- Remove `fetchCategories` method

Try Again

- Replace `categories` in the component's template with `$store.state.categories`
- Remove the `ApiService` import
- **Run your code** to check that everything still works as normal

For Category and CategoryBookList:

- Remove the `books` data from the Category component
- Remove the `fetchSelectedCategoryBooks` method
- Remove the call to `fetchSelectedCategoryBooks` from `created` or `mounted`
- Remove the **`props`** from **`CategoryBookList`** and adjust **`Category.vue`** accordingly. Change **`CategoryBookList`** so that it now reads the selected books from the store directly using `$store.state.selectedCategoryBooks` in the template section.
- Remove the `ApiService` import from the **`Category`** component.
- **Run your code** to check that everything still works as normal

Finally, since the Vuex store will act as our single source of truth, use `$store.state.selectedCategoryName` rather than `$route.params.name` in the template section of your projects.

We are now using the Vuex store, allowing us to debug using the Vuex developer tool and have our components all have access to the store without imports and data fields. By loading categories at the App level, and books on a category page basis we are reducing the number of network calls as well.

6 Use the Shopping Cart and the Vuex Store

Try Again

Unzip the provided `ShoppingCartResources.zip` file within your client project. You should unzip `ShoppingCart.js` and `ShoppingCartItem.js` files into a new folder under `src` called `models`.

Do the following in your Vuex store:

- Add a field `cart: new ShoppingCart()`
- Add a mutation `ADD_TO_CART`
- Add an action `addToCart`

The mutation should take the state and a book as parameters, and update the cart using `state.cart.addItem(book, 1)`. The action should take a context and a book parameter, and commit the mutation with the book committed as the payload to the mutation.

In your `AppHeader.vue` component, use the `$store` to show the size of the cart object in your header. (See ShoppingCart's numberOfItems getter method.)

In your `CategoryBookListItem` component, write an `addToCart` method that takes a book parameter, and dispatches the book to your new `addToCart` action in the Vuex store. Finally, add an event listener to your add-to-cart link or button: `@click="addToCart(book)"`.

Run your application. Clicking on your add to cart button should update the header's size counter, and also update the state of your cart in the Vuex store.

---

http://cs5244.cs.vt.edu:8080/GraceBookstoreState/

Try Again